Paulo Sérgio dos Santos Júnior

From Continuous Software Engineering Reference Ontologies to the Integration of Data for Data-Driven Software Development

Vitória, ES

2023

Paulo Sérgio dos Santos Júnior

From Continuous Software Engineering Reference Ontologies to the Integration of Data for Data-Driven Software Development

Tese de Doutorado submetida ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Doutor em Ciência da Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Programa de Pós-Graduação em Informática

Supervisor: Profa. Dra. Monalessa Perini Barcellos

Co-supervisor: Prof. Dr. João Paulo Andrade Almeida

Vitória, ES

2023

Ficha catalográfica disponibilizada pelo Sistema Integrado de
Bibliotecas - SIBI/UFES e elaborada pelo autor

Santos Junior, Paulo Sérgio, 1983-
S237f From Continuous Software Engineering Reference Ontologies to the Integration of Data for Data-Driven Software Development / Paulo Sérgio Santos Junior. - 2023.
210 f. : il.

Orientadora: Monalessa Perini Barcellos.
Coorientador: João Paulo Andrade Almeida.
Tese (Doutorado em Informática) - Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Engenharia de Software Continua. 2. Ontologias. 3. Rede de Ontologias. 4. Interoperabilidade Semântica. 5. Integração de Dados. I. Barcellos, Monalessa Perini. II. Almeida, João Paulo Andrade. III. Universidade Federal do Espírito Santo. Centro Tecnológico. IV. Título.

CDU: 004

From Continuous Software Engineering Reference Ontologies to the Integration of Data for Data-Driven Software Development

Paulo Sérgio dos Santos Júnior

Tese de Doutorado submetida ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Aprovada em 20 de dezembro de 2023.

Profa. Dra. Monalessa Perini Barcellos
Orientador, participação remota

Prof. Dr. João Paulo Andrade Almeida
Coorientador, participação remota

Prof. Dr. Breno Bernard Nicolau de França
Membro Externo, participação remota

Prof. Dr. Gleison dos Santos Souza
Membro Externo, participação remota

Prof. Dr. José Maria Parente de Oliveira
Membro Externo, participação remota

Prof. Dr. Vitor Estêvão Silva Souza
Membro Interno, participação remota

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
Vitória/ES, 20 de dezembro de 2023

Documento assinado eletronicamente nos moldes do art. 10 da MP 2200/01 e Lei 14063/20
[Hash SHA256] 548786fcfe873ba1ec3e471ecf7bb83f17d93bd12a3184294ddb5617612c1210

Datas e horários baseados em Brasília, Brasil
Sincronizado com o NTP.br e Observatório Nacional (ON) em 21/12/2023 às 06:29:54

SantosJr.DefesaDoutoradoAprovacao

Data e Hora de Criação: 20/12/2023 às 16:43:47

Documentos que originaram esse envelope:
- SantosJr.DefesaDoutoradoAprovacao.pdf (Arquivo PDF) - 1 página(s)

Hashs únicas referente à esse envelope de documentos

[SHA256]: 548786fcfe873ba1ec3e471ecf7bb83f17d93bd12a3184294ddb5617612c1210
[SHA512]: 2796058bc314864abcb002d71cc5edec475ac89b9ed86861b6019a59550ad4078232d18ad7a63740a087dfcfd4bf63350430e86688c3509fa954e4c3e5c5b3e6

Lista de assinaturas solicitadas e associadas à esse envelope

ASSINADO - Breno Bernard Nicolau de França (bfranca@unicamp.br)
Data/Hora: 20/12/2023 - 16:47:48, IP: 143.106.58.11, Geolocalização: [-22.814949, -47.064614]
[SHA256]: 56978c57348227d44758aad595a7f5d64b57f87dcdf1ea7d815ddf15ee17196a

ASSINADO - Gleison Santos (gleison.santos@uniriotec.br)
Data/Hora: 20/12/2023 - 17:23:07, IP: 191.162.171.42
[SHA256]: 3ad00ca0161b307b12c630877ba8a2dfd5fabdc2d96eb02afe069d9f5840d03e

ASSINADO - João Paulo Andrade Almeida (joao.p.almeida@ufes.br)
Data/Hora: 20/12/2023 - 17:35:16, IP: 187.36.219.45, Geolocalização: [-20.257740, -40.271672]
[SHA256]: 21c4171f622eef1e1750dd43089188705ad92f3542ed1f15494a05afa03e5d35

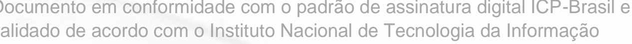
ASSINADO - Monalessa Perini Barcellos (monalessa.barcellos@ufes.br)
Data/Hora: 20/12/2023 - 16:47:39, IP: 200.137.65.103
[SHA256]: c9ca8e5e05faaad5bd7cfcebf6eb9a8c5d1821488c15977fe52cb6687fd9bff5

ASSINADO - José Maria Parente de Oliveira (parente@ita.br)
Data/Hora: 20/12/2023 - 17:44:40, IP: 191.19.195.214, Geolocalização: [-22.725266, -45.128404]
[SHA256]: 556c7e6471fa2584192273e9c51a9d79e916de2ea4509d768565d1130c133763

ASSINADO - Vítor Estêvão Silva Souza (vitor.souza@ufes.br)
Data/Hora: 21/12/2023 - 06:29:54, IP: 187.36.171.219, Geolocalização: [-20.289238, -40.292067]
[SHA256]: 58ffda908ebdea0d504656cf2afa1960c72c9351e28c166d6cdd51cf86f56887

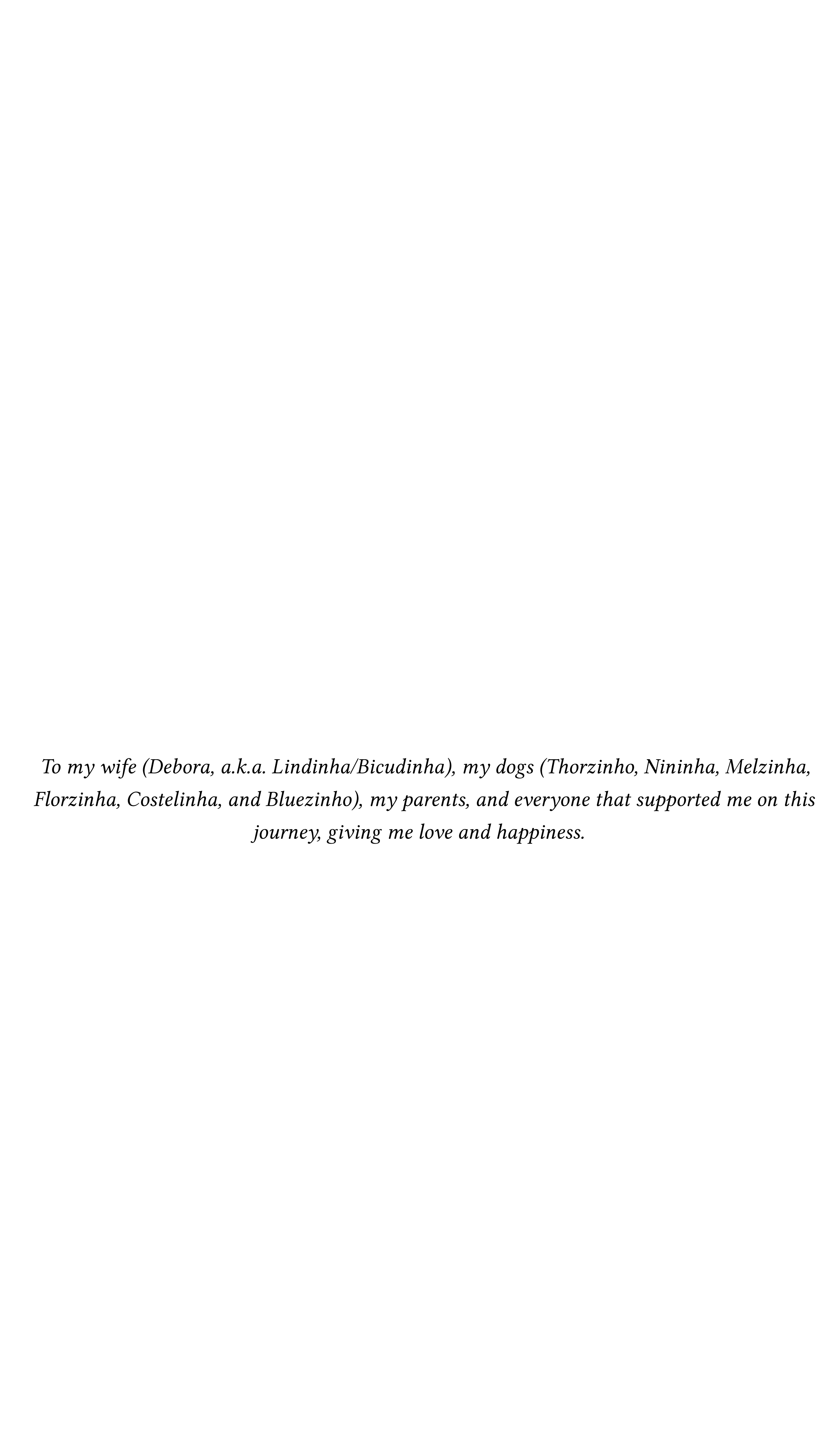
Histórico de eventos registrados neste envelope

21/12/2023 06:29:54 - Envelope finalizado por vitor.souza@ufes.br, IP 187.36.171.219
21/12/2023 06:29:54 - Assinatura realizada por vitor.souza@ufes.br, IP 187.36.171.219
21/12/2023 06:29:50 - Envelope visualizado por vitor.souza@ufes.br, IP 187.36.171.219
20/12/2023 17:44:40 - Assinatura realizada por parente@ita.br, IP 191.19.195.214
20/12/2023 17:44:26 - Envelope visualizado por parente@ita.br, IP 191.19.195.214
20/12/2023 17:35:16 - Assinatura realizada por joao.p.almeida@ufes.br, IP 187.36.219.45
20/12/2023 17:23:07 - Assinatura realizada por gleison.santos@uniriotec.br, IP 191.162.171.42
20/12/2023 17:22:59 - Envelope visualizado por gleison.santos@uniriotec.br, IP 191.162.171.42
20/12/2023 16:47:48 - Assinatura realizada por bfranca@unicamp.br, IP 143.106.58.11
20/12/2023 16:47:43 - Envelope visualizado por bfranca@unicamp.br, IP 143.106.58.11
20/12/2023 16:47:39 - Assinatura realizada por monalessa.barcellos@ufes.br, IP 200.137.65.103
20/12/2023 16:45:08 - Envelope registrado na Blockchain por monalessa.barcellos@ufes.br, IP 200.137.65.103
20/12/2023 16:45:08 - Envelope encaminhado para assinaturas por monalessa.barcellos@ufes.br, IP 200.137.65.103
20/12/2023 16:43:48 - Envelope criado por monalessa.barcellos@ufes.br, IP 200.137.65.103



Documento em conformidade com o padrão de assinatura digital ICP-Brasil e validado de acordo com o Instituto Nacional de Tecnologia da Informação

Os registros de assinatura presentes nesse documento pertencem única e exclusivamente a esse envelope.
Documento final gerado e certificado por **Universidade Federal do Espírito Santo**



To my wife (Debora, a.k.a. Lindinha/Bicudinha), my dogs (Thorzinho, Nininha, Melzinha, Florzinha, Costelinha, and Bluezinho), my parents, and everyone that supported me on this journey, giving me love and happiness.

Acknowledgements

When I stopped to reflect on what I would like to express in this section, I found myself pondering which words I could use to convey my gratitude to the people who have helped me reach this moment. The word that came to mind was “**opportunity**”.

At the beginning of my university journey, Crediné and Dedê provided me with the **opportunity** to join LIED (Computer in Education Laboratory). For nearly four years, I benefited from their scholarships and came to understand how education can transform reality.

Giancarlo, João Paulo, and Renata graciously gave me the **opportunity** to join NEMO (Ontology and Conceptual Modeling Research Group) and collaborate on a project with them. It was through this **opportunity** that I gained the valuable experience of applying research knowledge in practical, real-world situations. I genuinely believe that this **opportunity** played a pivotal role in shaping my academic career and had a significant impact on my future endeavors, as well as my perspective on how academia contributes to the progress of society[1].

Years later, when I passed the exam to become a professor at the *Federal Institute of Education, Science and Technology of Espírito Santo* (IFES), Vítor Souza assisted me through the process and offered me several insights into the IFES admission process. When I went to thank him and asked how I could repay his kindness, he shared with me the following wisdom: “*Do good for someone else.*” Since then, I have tried to follow this principle. Thank you for the **opportunity** and the valuable lesson.

Monalessa gave me the **opportunity** to work put into practice a crazy idea that became the focus of my doctoral research. Working with her all these years, Mona gave me the chance to learn what the process, vision, and significance of being a top-notch researcher and making an impact on society truly entails. She did this without uttering a single word, simply by setting an example. Furthermore, she showed me through numerous examples how a good mentor should motivate and treat everyone around them with respect and honesty. Finally, she demonstrated that hard work, a bit of chocolate, and unwavering focus always yield positive results on the journey.

João Paulo gave me the **opportunity** to work with him on my doctorate, years after I had completed my master’s degree. Like Monalessa, he showed me how a quality researcher should produce impactful results and develop people to reach their potential.

During my doctorate, I had the **opportunity** to work with several friends who helped me at different points in my journey. Among these friends, Rodrigo Fernandes Calhau, and

[1] João Paulo got me hooked on a heavy addiction: *Model-Driven Design*. To this day, I’ve been trying to break free from that habit, but I can’t.

Fabiano Borges Ruy contributed to works related to my thesis. While other friends (Cadu, Simone, Glaice, Jordana, Alexandre and others) gave me the **opportunity** to learn in other areas of computing that also helped me on my journey.

I thank all the members of the doctoral committee for evaluating my work and contributing to this journey of mine. Lastly, I want to express my gratitude to everyone who has been part of my journey, which was made in a quality public institution (*Federal University of Espírito Santo* - UFES).

I firmly believe that the person I am today is a result of the **opportunities** and valuable lessons provided to me over these years by all those mentioned above. Thus, I would like to say, "*Thank you for the **opportunities** you've given me. I hope to do the same for someone someday.*"

"We have to dream, otherwise things don't happen." (Oscar Niemeyer)

Resumo

Contexto: As organizações de software têm enfrentado vários desafios, como a necessidade de entregas mais rápidas, mudanças frequentes nos requisitos, menor tolerância a falhas e a necessidade de se adaptar aos modelos de negócios contemporâneos. As organizações devem evoluir para o desenvolvimento contínuo e orientado por dados em uma abordagem de engenharia de software contínua. A Engenharia de Software Contínua (ESC) consiste em um conjunto de práticas e ferramentas que suportam uma visão holística do desenvolvimento de software com o propósito de torná-lo mais rápido, iterativo, integrado, contínuo e alinhado com o negócio. As organizações de software frequentemente utilizam diferentes aplicativos para apoiar a CSE (e.g., ferramentas de gerenciamento de projetos, repositórios de código e ferramentas de avaliação de qualidade), que armazenam dados úteis para um processo orientado a dados. No entanto, os dados muitas vezes permanecem dispersos em diferentes aplicativos, cada um adotando modelos de dados e comportamentais diferentes, representando uma barreira para o uso integrado de dados. Como consequência, o desenvolvimento de software orientado por dados é incomum, perdendo oportunidades valiosas para melhoria de produto e processo, bem como identificação de novas oportunidades de negócios. *Objetivo:* Considerando a necessidade de possibilitar o desenvolvimento de software orientado por dados no contexto da CSE, nosso objetivo é fornecer uma abordagem baseada em ontologia que possa auxiliar na identificação das necessidades de informação da organização, recuperar dados de aplicativos e fornecer dados integrados que atendam às necessidades de informação. *Método:* Seguindo o paradigma da *Design Science Research* organizando estudos experimentais como *learning iterations*, desenvolvemos a abordagem *Immigrant*, que contém três componentes: *California* (um processo baseado em Pensamento Sistêmico), *Zeppelin* (um instrumento diagnóstico de CSE) e *The Band* (uma solução de integração baseada em ontologia que integra semanticamente dados de aplicativos). *The Band* é baseado em *Continuum*, uma (sub)rede ontológica desenvolvida neste trabalho para abordar aspectos da CSE (particularmente, desenvolvimento ágil, integração contínua e implantação contínua) e que é usada como um modelo de referência para construir artefatos de software na arquitetura de integração. *Resultados:* Estudos realizados em organizações de software avaliaram cada componente separadamente. Os resultados demonstram a utilidade de *California*, *Zeppelin* e *The Band* fornecendo dados que ajudaram identificar problemas relacionados à alocação de equipes, gerenciamento de produtividade de equipe e desempenho do projeto. A proposta completa *Immigrant* foi avaliada em um estudo de caso, identificando problemas relacionados à alocação de tarefas, sobrecarga de papéis e qualidade do código. *Conclusão:* Os resultados obtidos até agora sugerem que *Immigrant* é uma abordagem útil para possibilitar o desenvolvimento de software orientado por dados na CSE.

Palavras-chaves: Engenharia de Software Contínua, Rede de Ontologias, Interoperabilidade Semântica, Integração de Dados.

Abstract

Context: Software organizations face several challenges, such as the need for faster deliveries, frequent changes in requirements, lower tolerance to failures, and the need to adapt to contemporary business models. Agile practices have allowed organizations to shorten development cycles and increase customer collaboration. However, this has not been enough. Organizations should evolve to continuous and data-driven development in a continuous software engineering approach. Continuous Software Engineering (CSE) consists of a set of practices and tools that support a holistic view of software development with the purpose of making it faster, iterative, integrated, continuous, and aligned with the business. Software organizations often use different applications to support CSE (e.g., project management tools, source repositories, and quality assessment tools). These applications store useful data to enable a data-driven software development process. However, data items often remain spread in different applications, each adopting different data and behavioral models, posing a barrier to integrated data usage. As a consequence, data-driven software development is uncommon, missing valuable opportunities for product and process improvement as well as new business opportunities identification. *Objective:* Considering the need to enable data-driven software development in the CSE context, we aim to provide an ontology-based approach that can aid in: identifying the organization's information needs, retrieving data from applications, and providing integrated data that meets the information needs. *Method:* By following the Design Science paradigm and organizing experimental studies as learning iterations, we developed the *Immigrant* approach, which contains three components: *California* (a System-Thinking-based process), *Zeppelin* (a CSE diagnostic instrument, which helps identify the organization information needs), and *The Band* (an ontology-based integration solution that semantically integrates data from applications and, thus, provides integrated data to support data-driven software development). *The Band* is based on *Continuum*, an ontology (sub)network developed in this work to address CSE aspects (particularly, agile development, continuous integration, and continuous deployment) and that is used as a reference model to build software artifacts in the integration architecture. *Results:* Studies performed in software organizations evaluated each component separately. Results demonstrate *California* and *Zeppelin*'s usefulness and show that the integrated solution (*The Band*) contributed to improving estimates, provided data that helped allocate teams, manage team productivity and project performance, and allowed to identify and fix problems in the software process execution. The complete proposal *Immigrant* was evaluated in a case study. As a result, it was possible to identify problems related to the allocation of tasks, role overload, and code quality. *Conclusion*: The results obtained so far suggest that *Immigrant* is a useful approach to enable data-driven software development in CSE.

Keywords: Continuous Software Engineering, Ontology Network, Semantic Interoperability, Data Integration.

List of Figures

Figure 1 – Overview of the Design Science cycles in this research (based on (BARCELLOS et al., 2022)). 29
Figure 2 – Stairway to Heaven Model (StH) (OLSSON; ALAHYARI; BOSCH, 2012) . . 32
Figure 3 – Continuous * (FITZGERALD; STOL, 2017). 33
Figure 4 – The Eye of CSE (JOHANSSEN et al., 2019). 33
Figure 5 – Framework for Continuous Software Engineering (BARCELLOS, 2020). . . 34
Figure 6 – Continuous Integration based on (DUVALL; MATYAS; GLOVER, 2007). . . 36
Figure 7 – Remote and Local Source Repository. 38
Figure 8 – Staging Area and Source Repository, based on (CONSERVANCY, 2023). . . 39
Figure 9 – Commits and their parents (CONSERVANCY, 2023). 39
Figure 10 – Relation between branch and commits (CONSERVANCY, 2023). 39
Figure 11 – Branches, based on (ATLASSIAN, 2023). 40
Figure 12 – Merge Commit, based on (CONSERVANCY, 2023). 40
Figure 13 – The relationship between CI, CDE, and CD, based on (SHAHIN; BABAR; ZHU, 2017). 42
Figure 14 – A UFO fragment. 45
Figure 15 – SEON's Architecture (RUY et al., 2016) 48
Figure 16 – A EO, SPO, and SysSWO fragment. 49
Figure 17 – A SPO fragment focuses on Resource concepts. 51
Figure 18 – CMPO fragment. 52
Figure 19 – ROoST fragment. 54
Figure 20 – A QAPO fragment. 55
Figure 21 – A OSDEF fragment. 56
Figure 22 – A RSRO fragment. 57
Figure 23 – A general three-tier architecture of a FIS (BUSSE et al., 1999). 59
Figure 24 – *Continuum*'s architecture. 62
Figure 25 – SRO's architecture. 64
Figure 26 – Scrum Process Subontology. 65
Figure 27 – Scrum Stakeholders Subontology. 66
Figure 28 – Scrum Stakeholders Participation Subontology. 68
Figure 29 – Product and Sprint Backlog Subontology. 70
Figure 30 – Scrum Deliverables Subontology. 72
Figure 31 – CIRO's architecture. 80
Figure 32 – CMPO fragment focusing on Checkout. 80
Figure 33 – CMPO fragment focusing on Checkin. 81
Figure 34 – Continuous Integration Process Subontology. 83

Figure 35 – Continuous Build Process Subontology Modularization. 85
Figure 36 – Continuous Build Environment model from CI Building Environment package. 85
Figure 37 – Continuous Build Process model from Continuous Build Process package. . 86
Figure 38 – Continuous Test Process Subontology. 88
Figure 39 – Continuous Inspection Process Subontology Modularization. 89
Figure 40 – Continuous Inspection Environment model from CI Inspection Environment package. 90
Figure 41 – Continuous Inspection Process Subontology. 91
Figure 42 – CDRO's Architecture. 95
Figure 43 – Continuous Delivery Activity Subontology. 96
Figure 44 – Continuous Deployment Process Subontology. 98
Figure 45 – Learning Iteration (BARCELLOS et al., 2022). 107
Figure 46 – Overview of the LIs performed to develop *Immigrant*. 108
Figure 47 – Process followed to develop the integration solution (SANTOS et al., 2021). 110
Figure 48 – Fragment of the dashboard showing integrated data (SANTOS et al., 2021). 111
Figure 49 – Fragment of Systemic map (SANTOS; BARCELLOS; CALHAU, 2020). . . . 116
Figure 50 – California: A System Theory-Based Process (SANTOS; BARCELLOS; CALHAU, 2020). 122
Figure 51 – Fragment of the Diagnosis Questionnaire with practices related to Continuous Integration (SANTOS; BARCELLOS; RUY, 2021). 127
Figure 52 – Fragment of Analytic Report (SANTOS; BARCELLOS; RUY, 2021). 127
Figure 53 – Immigrant overview. 133
Figure 54 – Transformation of ON into FIS. 134
Figure 55 – *Journey* overview. 136
Figure 56 – Fragment of SRO Information Model (SANTOS et al., 2021). 137
Figure 57 – Fragment of Microsoft Azure DevOps data model (SANTOS et al., 2021). . . 137
Figure 58 – Fragment of the class diagram of a lib to access Microsoft Azure DevOps data. 139
Figure 59 – Fragment of SRO OBDR. 139
Figure 60 – SRO's OBS using a REST interface. 140
Figure 61 – SRO's OBS using a GraphQL interface. 140
Figure 62 – Concept added to OBDR to allow data traceability. 141
Figure 63 – Example of ETL components and OBDRs. 141
Figure 64 – Example of an interface provided by a data access component. 143
Figure 65 – *The Band* architecture overview. 143
Figure 66 – Application Integration Layer architecture. 144
Figure 67 – Internal Data Communication Layer architecture. 145
Figure 68 – Federated Ontology-based Service Layer architecture. 146
Figure 69 – Federated Data Access Layer Architecture. 147
Figure 70 – SRO Data View. 148

Figure 71 – Fragment of systemic map. 166
Figure 72 – Fragment of the dashboard focusing on User Story Wait Time. 170
Figure 73 – Fragment of the dashboard showing number of US (per type) defined over time. 171
Figure 74 – Fragment of the dashboard showing the *Task Cycle Time* by the projects. . 171
Figure 75 – Fragment of the dashboard showing the Number of Code Smells (per type and artifact). 172
Figure 76 – Fragment of the dashboard showing data regrading CI Pipeline Success Rate in a project. 172
Figure 77 – Fragment of the dashboard focusing on CI Pipeline Success Rate in all projects. 172
Figure 78 – Diagnostic Questionnaire - Context Form. 204
Figure 79 – Diagnostic Questionnaire - Instruction Form. 204
Figure 80 – Diagnostic Questionnaire - Organization Profile Form. 204
Figure 81 – Diagnostic Questionnaire - Participant Profile Form. 205
Figure 82 – Diagnostic Questionnaire - Agile Organization Form. 205
Figure 83 – Diagnostic Questionnaire - Continuous Integration Form. 206
Figure 84 – Diagnostic Questionnaire - Continuous Deployment Form. 206
Figure 85 – Diagnostic Questionnaire - R&D as Innovation System Form. 206

List of Tables

Table 1 – SRO Verification. 74
Table 2 – SRO Validation. 77
Table 3 – Verification of CIRO. 92
Table 4 – Validation of CIRO. 93
Table 5 – Verification of CDRO. 99
Table 6 – Validation of CDRO. 101
Table 7 – Fragment of GUT Matrix. 118
Table 8 – Causes of Undesirable Behaviors. 119
Table 9 – Strategies, Causes, and Agile Concepts. 120
Table 10 – Some terms present in Dictionary based on SRO. 120
Table 11 – Effort spent on development and bug-fixing tasks in different projects. . . . 123
Table 12 – Effort spent on development and bug-fixing tasks before and after applying the strategies in the project. 123
Table 13 – Examples of Semantic Mappings between SRO Information Model and Microsoft Azure DevOps Data Model (SANTOS et al., 2021). 137
Table 14 – Federated Information System's Criteria on The Band. 149
Table 15 – Example of the Information Needs. 151
Table 16 – Examples of measures defined to meet the information needs. 151
Table 17 – Measurement formulas defined based on *Continuum* concepts 152
Table 18 – Adoption Degree by StH stage. 161
Table 19 – Adoption Degree by Eye of CSE category. 161
Table 20 – CSE Practices Adoption Degree by StH' Stages, before and after interview. . 163
Table 21 – CSE Practices Adoption Degree by Eye of CSE, before and after interview. . 163
Table 22 – Information needs identified based on the use of Zeppelin. 164
Table 23 – Some of the identified undesirable behaviors 167
Table 24 – Causes of Undesirable Behaviors. 167
Table 25 – Strategies and Causes. 168
Table 26 – Information needs identified based on the use of *California*. 168
Table 27 – Measures defined to meed the Information Needs. 169
Table 28 – Contributions versus Specific Objectives. 184
Table 29 – Agile Organization Stage‘s Statements. 207
Table 30 – Continuous Integration Stage‘s Statements. 208
Table 31 – Continuous Deployment Stage‘s Statements. 208
Table 32 – R&D as Innovation System Stage‘s Statements. 209

List of abbreviations and acronyms

ASA	Application Software Artifact
CD	Continuous Deployment
CDE	Continuous DElivery
CDDRO	Continuous Deployment and Delivery Reference Ontology
CI	Continuous Integration
CIRO	Continuous Integration Reference Ontology
CMPO	Configuration Management Process Ontology
COM	Core Ontology on Measurements
CSE	Continuous Software Engineering
DaD	Disciplined Agile Delivery
DSR	Design Science Research
EO	Enterprise Ontology
ETL	Extract, Transform, and Load
FCSE	Framework for Continuous Software Engineering
FIS	Federated Information Systems
GUT Matrix	Gravity, Urgency, and Tendency Matrix
LeSS	Large-Scale Scrum
LI	Learning Iteration
NO	Networked Ontology
OBDR	Ontology-Based Data Repository
OBS	Ontology-Based Service
ON	Ontology Network
OSDEF	Reference Ontology of Software Defects, Errors and Failures
OWL	Web Ontology Language

QAPO	Quality Assurance Process Ontology
ROoST	Reference Ontology on Software Testing
RSRO	Reference Software Requirements Ontology
Safe	Scaled Agile Framework
SE	Software Engineering
SEON	Software Engineering Ontology Network
SPO	Software Process Ontology
SRO	Scrum Reference Ontology
StH	Stairway to Heaven
SysSwO	System and Software Ontology
UFO	Unified Foundational Ontology
UML	Unified Modeling Language
XP	Extreme Programming

Contents

1 INTRODUCTION . . . 19
1.1 Context and Motivation . . . 19
1.2 Research Hypothesis . . . 22
1.3 Objectives . . . 23
1.4 Research Method . . . 24
1.4.1 Relevance Cycle . . . 24
1.4.2 Design Cycle . . . 25
1.4.3 Rigor Cycle . . . 28
1.5 Organization of this Thesis . . . 28

2 BACKGROUND . . . 31
2.1 Continuous Software Engineering (CSE) . . . 31
2.1.1 Agile Development and Scrum . . . 34
2.1.2 Continuous Integration (CI) . . . 36
2.1.3 Continuous DElivery (CDE) and Continuous Deployment (CD) . . . 41
2.2 Ontology and Ontology Network . . . 43
2.2.1 Unified Foundational Ontology (UFO) . . . 44
2.2.2 Software Engineering Ontology Network (SEON) . . . 47
2.2.2.1 SPO, EO, and SysSwO . . . 48
2.2.2.2 Configuration Management Process Ontology (CMPO) . . . 51
2.2.2.3 Reference Ontology on Software Testing (ROoST) . . . 53
2.2.2.4 Quality Assurance Process Ontology (QAPO) . . . 54
2.2.2.5 Reference Ontology of Software Defects, Errors, and Failures (OSDEF) . . . 55
2.2.3 Reference Software Requirements Ontology (RSRO) . . . 56
2.3 Semantic Integration . . . 57
2.4 Federated Information Systems . . . 58
2.5 Final Considerations . . . 60

3 CONTINUUM - A CONTINUOUS SOFTWARE ENGINEERING ONTOLOGY (SUB)NETWORK . . . 61
3.1 *Continuum* Overview . . . 61
3.2 Scrum Reference Ontology (SRO) . . . 63
3.2.1 Scrum Process subontology . . . 63
3.2.2 Scrum Stakeholders Subontology . . . 66
3.2.3 Scrum Stakeholders Participation Subontology . . . 67
3.2.4 Product and Sprint Backlog Subontology . . . 69

3.2.5 Scrum Deliverables Subontology . . . 71
3.2.6 Evaluation . . . 73
3.3 Continuous Integration Reference Ontology (CIRO) . . . 79
3.3.1 Extension of the Configuration Management Process Ontology (CMPO) . 79
3.3.2 Continuous Integration Process Subontology . . . 82
3.3.3 Continuous Build Process Subontology . . . 84
3.3.4 Continuous Test Process Subontology . . . 87
3.3.5 Continuous Inspection Process Subontology . . . 89
3.3.6 Evaluation . . . 91
3.4 Continuous Deployment Reference Ontology (CDRO) . . . 95
3.4.1 Continuous Delivery Activity Subontology . . . 95
3.4.2 Continuous Deployment Process Subontology . . . 97
3.4.3 Evaluation . . . 99
3.5 Related Work . . . 101
3.6 Final Considerations . . . 104

4 LEARNING ITERATIONS TOWARDS IMMIGRANT . . . 106
4.1 Learning Iterations . . . 106
4.2 First Learning Iteration: Towards an Ontology-Based Approach to Integrate Data Application . . . 107
4.2.1 Execution and Results . . . 109
4.2.2 What did we learn? . . . 112
4.3 Second Learning Iteration: California . . . 114
4.3.1 Theoretical Background . . . 115
4.3.2 Execution and Results . . . 116
4.3.3 Threats to Validity to the Study Results . . . 123
4.3.4 What did we learn? . . . 124
4.4 Third Learning Iteration: Zeppelin . . . 125
4.4.1 Execution and Results . . . 125
4.4.2 Threats to validity to study results . . . 129
4.4.3 What did we learn? . . . 130
4.5 Final Considerations . . . 131

5 IMMIGRANT . . . 132
5.1 *Immigrant* Overview . . . 132
5.2 The use of an Ontology Network and Federated Information Systems in *The Band* . . . 133
5.3 Journey: *The Band* Development Process . . . 135
5.4 *The Band* Architecture . . . 142
5.4.1 Application Integration Layer . . . 144

5.4.2 Internal Data Communication Layer 144
5.4.3 Federated Ontology-based Service Layer 145
5.4.4 Federated Data Access Layer 146
5.5 Implementing *The Band* 148
5.6 *The Band* as FIS 149
5.7 Using *Immigrant* 151
5.8 Related work 153
5.9 Final Considerations 155

6 FINAL LEARNING ITERATION: APPLYING *IMMIGRANT* IN A SOFTWARE ORGANIZATION 157
6.1 Context 157
6.2 Study Planning 158
6.3 Study Execution, Data Collection, and Results 159
6.3.1 Identifying Information Needs from *Zeppelin* 160
6.3.2 Applying *California* to complement the Information Needs 165
6.3.3 Identifying the Available Sources 169
6.3.4 Defining Measures 169
6.3.5 Providing Integrated Data using *The Band* 170
6.3.6 Getting Feedback about *Immigrant* 173
6.4 Discussion 175
6.5 Threats to Validity 176
6.6 What did we learn? 177
6.7 Final Considerations 178

7 FINAL CONSIDERATIONS 180
7.1 Summary of the Research 180
7.2 Research Contributions 182
7.3 Research Limitations 186
7.4 Perspectives of Future Works 187

BIBLIOGRAPHY 190

APPENDIX 202

APPENDIX A – ZEPPELIN 203
A.1 Diagnostic Questionnaire 203

1 Introduction

Yes, there are two paths you can go by, but in the long run, there's still time to change the road you're on.

Led Zeppelin, Stairway to Heaven

This chapter presents an overview of this thesis and defines the basis for the following chapters. It discusses the research context and motivation, research hypothesis, objectives, and methodological aspects that have guided the work. Last, it presents the structure of this document.

1.1 Context and Motivation

Characteristics and demands of the modern and digital society have transformed the software development scenario and presented new challenges to software developers and engineers, such as the need for faster deliveries, frequent changes in requirements, lower tolerance to failures, and the need to adapt to contemporary business models (BARCELLOS, 2020). Some of the difficulties that need to be overcome when dealing with these challenges are due to the lack of connection between important software development activities such as planning, implementation, and deployment (FITZGERALD; STOL, 2017). These difficulties are usually accentuated when development adopts a traditional sequential approach, prescribed by the waterfall life cycle model.

Agile methods, such as Scrum (SCHWABER; KEN, 2013), XP (KNIBERG, 2015), and Kanban (KNIBERG; SKARIN, 2010), have been increasingly adopted in software development to deal with some of the aforementioned issues, because they enable organizations to deliver valuable products to clients in short iterative cycles, increase customer collaboration, and improve the organization responsiveness to change (JULIAN; NOBLE; ANSLOW, 2019; SCHWABER; KEN, 2013; BARCELLOS, 2020). However, this has not been enough. Continuous actions of planning, construction, operation, deployment, and evaluation are necessary to produce products that properly meet customers' needs, make well-informed decisions, and identify business opportunities. Thus, organizations should evolve from traditional to continuous and data-driven software development in a Continuous Software Engineering (CSE) approach (BOSCH, 2014; FITZGERALD; STOL, 2017; BARCELLOS, 2020).

CSE consists of a set of practices and tools that supports a holistic view of software development to make it faster, iterative, integrated, continuous, and aligned with the business. It understands the development process not as a sequence of discrete activities, performed by distinct and disconnected teams, but as a continuous flow, considering the entire software

life cycle. It is a recent topic that seeks to transform discrete development practices into more iterative, flexible, and continuous alternatives, keeping the goal of building and delivering quality products according to established time and costs (FITZGERALD; STOL, 2017). For that, CSE involves agile practices and goes beyond them by emphasizing the need for continuity, alignment to business and a broader view of software development.

In this context, some initiatives have emerged aiming to speed up the development process and improve the connection between its activities. For example, Continuous Integration (BECK, 2000) seeks to eliminate discontinuities between development and delivery. In a similar approach, DevOps (DEBOIS et al., 2011) recognizes that the integration between software development and system operation must be continuous. Extending the need for integration to other levels, BizDev (FITZGERALD; STOL, 2017) advocates that continuity should exist not only in the software process context, but also between software and strategic processes of the organization.

Software organizations often use different applications to support different aspects of software development (FITZGERALD; STOL, 2017). For example, agile management practices can be supported by project management and time-tracking applications, while continuous development and continuous integration can be supported by integrated development environments, version control, and code quality tools. The intensive use of applications in software development creates opportunities involving the various kinds of data they store, enabling data-driven software development (BRYNJOLFSSON; HITT; KIM, 2011), which is characterized by the use of data to drive software engineering activities and decision-making. For example, data regarding code quality (e.g., number of defects, smells, etc.) and rework (e.g., effort spent fixing errors made during the development process) can provide useful information to support decisions about testing and coding strategies. In the CSE context, the use of development and customer-related data to support daily activities and decision-making is of key relevance (OLSSON; ALAHYARI; BOSCH, 2012; BOSCH, 2014; FITZGERALD; STOL, 2017).

Although applications employed throughout the software lifecycle store useful data to support data-driven software development, data items often remain spread in different applications, each of which adopts different data and behavioral models, posing a barrier to integrated data usage. As a consequence, using data to drive software development has been relatively uncommon, wasting valuable opportunities for informed decision-making. Particularly in agile software development, decisions related to software development have been commonly based on subjective aspects, such as previous experiences of the managers and stakeholders, intuitions, or a combination of these (SVENSSON; FELDT; TORKAR, 2019). A recent study in which we investigated CSE adoption in Brazilian organizations corroborates this perception as it revealed that only 17% of software organizations have used data to drive software development (SANTOS et al., 2022).

One of the reasons organizations fail to leverage data stored in applications is the

difficulty to access, integrate, analyze, and visualize data handled by heterogeneous applications. In general, each application implements its own data and behavioral models and focuses on specific aspects of the software process, with little concern for sharing and integration aspects, leading thus to several conflicts (CALHAU; FALBO, 2010). Particularly in the agile development context, the challenge is to use data to support the development process in such a way that does not represent a bottleneck to process agility. There is a need to extract useful information from data stored in applications and to present it to the team within their development environment, effectively and proactively (WACHE et al., 2001), without requiring extra effort from the development team.

One source of difficulty for data integration is semantic heterogeneity, which can result in conflicts whenever the same information item is given divergent interpretations, a situation that may not even be detected (WACHE et al., 2001). Neglecting these "semantic conflicts" can lead to poorly integrated solutions that fail in achieving their purposes (e.g., providing incorrect information) (POKRAEV, 2009). To reduce these conflicts, integration should address semantic issues (CALHAU; FALBO, 2010; FONSECA; BARCELLOS; FALBO, 2017). A means for that is to employ ontologies to establish a common conceptualization about the applications' subject domains in order to support communication and application integration. An ontology is a formal, explicit specification of a shared conceptualization (GRUBER, 1993). Thus, it can be used as an interlingua to map the concepts used by different applications, enabling data and services understanding (CALHAU; FALBO, 2010). In fact, ontologies have become the predominant way to deal with semantic aspects in semantic integration initiatives (NARDI; FALBO; ALMEIDA, 2013).

For large and complex domains, such as Software Engineering, representing all concepts in a single ontology could result in a large and monolithic artifact that would be hard to manipulate, use, and maintain (RUY et al., 2016). In such cases, using an ontology network (ON) is a better solution (RUY et al., 2016). An ON is a set of ontologies connected to each other through relationships (e.g., dependency and alignment) to provide a comprehensive and consistent conceptualization (SUÁREZ-FIGUEROA; GÓMEZ-PÉREZ; FERNÁNDEZ-LÓPEZ, 2012). The integration of data from applications that address different subdomains of a (super) domain such as Software Engineering (e.g., Software Requirements, Coding, Design, Testing, and Software Quality) has the same problem. Therefore, a Software Engineer can explore a combination or a fragment—but nevertheless aligned— of networked ontologies to integrate data from different applications. This mitigates the risks of developing a single large ontology or integrating ontologies that are not consistent with each other.

In this work, we propose to use ontologies from an ON to integrate application data aiming at data-driven software development. More specifically, we propose to develop ontologies that address CSE aspects (agile development, continuous integration, and continuous deployment), add them to the Software Engineering Ontology Network (SEON) (RUY et al.,

2016) and explore the use of the ON to support application data integration. SEON is an ON that (i) describes various subdomains of the Software Engineering domain (e.g., Software Process, Software Requirements, Software Testing, Software Measurement); (ii) offers mechanisms to facilitate building and integrating new Software Engineering subdomain ontologies into the network; and (iii) promotes integration by keeping a consistent semantics for concepts and relations along the whole network (RUY et al., 2016). Since existing ontologies in SEON describe general concepts in Software Engineering, the ontologies to be added to it in this work can focus on specifics of CSE, while reusing established notions such as "software project", "project team", "requirement", among others.

In the literature, there are some approaches using ontologies to integrate software development data (e.g., tasks, deadlines, and team members' skills), including those of Calhau & Falbo (2010) and Fonseca, Barcellos & Falbo (2017). However, none of them explore ontologies from an ON or address CSE aspects. Our proposal also differs from software repository mining approaches, such as those of Cubranic et al. (2005) and Mattila et al. (2017), because they usually do not consider the use of ontologies (thus, with little attention to data semantics) and their integration solutions are devoted to solving integration problems in a specific context (e.g., using data from a particular software repository). By using ontologies from an ON, our proposal aims to provide an integration solution based on a common domain conceptualization. As a result, it can be used to integrate data from different applications, possibly involving different subdomains covered by the ON.

1.2 Research Hypothesis

Considering, as previously discussed that:

- To perform CSE, organizations use different applications to support different aspects of the software development process;
- A vast amount of data stored in applications is currently not put to use in the support of data-driven software development;
- There is a significant difficulty in accessing, integrating, analyzing, and visualizing data handled by heterogeneous applications;
- One source of difficulty for data integration is semantic heterogeneity, which can result in semantic conflicts;
- It is necessary to integrate applications considering semantic aspects to provide useful information;
- Ontologies can be used to establish a common conceptualization about a domain of interest and have become a way to deal with semantics in integration initiatives;

- Ontology networks allow organizing ontologies, providing comprehensive, consistent, and integrated knowledge;

We formulate the following research hypothesis:

The use of ontologies in an ontology network that addresses Continuous Software Engineering (CSE) aspects facilitates the semantic integration of data stored in diverse applications, and can thereby enable data-driven software development.

1.3 Objectives

The general objective of the work reported here is to *propose an ontology-based approach that uses networked ontologies to integrate application data aiming at enabling data-driven software development in the CSE context*. This general objective is broken down into the following specific objectives (SO):

- **SO1. Establish mechanisms to identify an organization's information needs**: the successful use of data to support software development and decision-making depends on providing useful information corresponding to organizational needs. Therefore, we aim to propose mechanisms to help organizations identify information needs that will guide data integration in the CSE context;
- **SO2. Develop networked ontologies on CSE subdomains**: ontologies on a number of subdomains of CSE such as Agile Development, Continuous Integration, and Continuous Deployment are required to establish a consensual, shared, and comprehensive conceptualization that can be used to assign semantics and facilitate the integration of data from different applications. The various CSE subdomain ontologies should be organized in an overarching ontology network as motivated earlier;
- **SO3. Create an ontology-based approach to integrate application data**: creating an approach based on an ontology network *(SO2)* and that considers the organization's information needs *(SO1)* and the available data helps provide integrated data for enabling data-driven software development. Many of the ontology-based approaches focus on conceptual integration (e.g., (CALHAU; FALBO, 2010; FONSECA; BARCELLOS; FALBO, 2017)), without supporting the integration solution design and implementation. An approach addressing the development of the integration architecture through reusable components can provide a flexible solution to cover a diversity of subdomains considered in the ON;
- **SO4. Apply the proposed approach in a real context**: the use of the proposed approach *(SO3)* in software organizations can offer evidence of the proposal's usefulness and feasibility considering its support to data-driven software development and its

contribution to improving software products (e.g., reducing defects) and processes (e.g., improving estimates).

1.4 Research Method

The research method adopted in this work follows the *Design Science Research* (DSR) paradigm, which concerns extending "*human and organizational capabilities by creating new and innovative artifacts*" (HEVNER, 2007). DSR was selected as the research approach because the object of study is an artifact — specifically, an approach that uses networked ontologies to integrate application data to support data-driven software development in CSE — and its evaluation is possible in a real organizational environment.

DSR comprises the following steps (PEFFERS et al., 2007): (i) *Problem identification and Motivation*, (ii) *Definition of the objectives for a solution*, (iii) *Design and Development*, (iv) *Demonstration*, (v) *Evaluation*, and (vi) *Communication*. These steps are organized in an iterative process, with three cycles: *Relevance Cycle*, *Design Cycle*, and *Rigor Cycle* (HEVNER, 2007).

A DSR project begins with the Relevance Cycle, which involves defining the problem to be addressed, requirements, and criteria for evaluating the results (HEVNER, 2007), including steps (i) and (ii). The Design Cycle involves developing and evaluating artifacts or theories to solve the identified problem (HEVNER, 2007), comprising steps (iii), (iv), and (v). The Rigor Cycle refers to using and generating knowledge (HEVNER, 2007), comprising step (vi), and the use of knowledge and foundations along with the work.

1.4.1 Relevance Cycle

In the *Problem identification and motivation* step, the problem was identified from the literature, e.g., as reported by Karvonen et al. (2015), Svensson, Feldt & Torkar (2019), Kasauli et al. (2020), and was also observed first hand by the doctoral candidate when working as a Scrum master and consultant in some software organizations. The problem refers to the need for integrating application data considering semantic issues to provide useful information that enables data-driven software development in CSE.

Considering the identified problem, in the *Definition of the objectives for a solution* step, we decided to develop an ontology-based approach to integrate application data aiming at supporting data-driven software development in the CSE context. As requirements for the approach, we defined that it must:

- **R1**. Support identifying an organization's information needs to enable data-driven software development in CSE;
- **R2**. Address semantic issues involved in data integration in such a complex domain;

- **R3**. Consider data available in the organization's applications; and
- **R4**. Provide integrated and meaningful data, considering the organization's information needs and available data.

In order to evaluate the proposed approach, we foresee an assessment of the usefulness (**C1**) and feasibility (**C2**) of the solution.

1.4.2 Design Cycle

In the *Design and development* step, we developed the *Immigrant* approach, the main artifact produced in this work. The development involved three empirical studies. As suggested in (BARCELLOS et al., 2022), they were organized as *learning iterations*, i.e., studies performed in iterations that allow the researcher to learn something about the research, by providing useful knowledge to understand the problem, develop the artifact, and evaluate or improve it. Each one of the three learning iterations aimed to answer specific questions and provided knowledge and results that helped us to better understand the problem and contributed to developing the proposed artifact.

The first learning iteration was an exploratory study performed to evaluate our design choice of using networked ontologies to meet requirement *R2*. The study aimed to answer the following question: *Is it useful and feasible to use an ontology network to integrate existing data stored in applications to meet organizations' information needs aiming at data-driven software development in CSE?*

The study was performed in a public software organization that adopted agile practices. We added a new ontology — the Scrum Reference Ontology — to SEON (RUY et al., 2016) and used it and other SEON ontologies as a basis for a solution that integrated data from two applications (Microsoft Azure DevOps[1] and Clockfy[2]), presented it in dashboards and supported data-driven software development. The main results were published in (SANTOS et al., 2021).

In that study, an initial version of the proposed approach was defined and we learned how networked ontologies can be used to integrate, share, and exchange data from different applications to support data-driven software development. Moreover, we confirmed in practice the necessity of properly identifying the organization's information needs to guide data integration, and we noticed that organizations face a number of difficulties to recognize such information needs.

Thus, we performed the second learning iteration, a participative case study that aimed

[1] Microsoft Azure DevOps: is a project management application. It can be accessed at <https://azure.microsoft.com/en-us/products/devops/>.

[2] Clockify: is a time-tracking application. It can be accessed at: <https://clockify.me/>.

at answering the following question: *How to understand the way an organization works and, thus, help identify its information needs relevant to data-driven software development in the CSE context?*

The study was performed in a Brazilian software house interested in implementing CSE practices gradually, by following the Stairway to Heaven model (StH) (OLSSON; ALAHYARI; BOSCH, 2012; KARVONEN et al., 2015), which defines five stages that organizations should follow to implement CSE until achieving data-driven software development. The study resulted in a Systems Theory-based process called *California*[3] (SANTOS; BARCELLOS; CALHAU, 2020) (SANTOS; BARCELLOS; CALHAU, 2022), which helps organizations identify strategies to implement CSE practices, using Systems Theory (MEADOWS, 2008), GUT Matrix (KEPNER; TREGOE, 1981), and Reference Ontologies (GUIZZARDI, 2007).

With that study, we learned that knowing the organization's current state and defining strategies for improving it helps identify information needs that integrated data should meet (e.g., the information needed to evaluate the strategies' effectiveness). We also learned that, although the proposed process (*California*) is useful, it involves a lot of tacit knowledge and judgment as well as knowledge of Systems Thinking tools and GUT matrix. Moreover, it may demand much time to be applied. In a software organization under study (SANTOS; BARCELLOS; CALHAU, 2020; SANTOS; BARCELLOS; CALHAU, 2022), two months were required to understand the CSE process present in that organization.

[3] The name *California* was inspired by the *Going to California* song by the Led Zeppelin band because the process allows organizations to choose which CSE practices (*Made up my mind to make a new start*) to improve aiming at achieving a new quality level (*Going to California*).

Considering the conclusions of the second iteration, and aiming to further examine the question it addressed, we performed a third learning iteration, in the form of a multiple case study in five organizations. The main purpose was to obtain an easier means to establish a panorama of an organization's current state of CSE practices adoption, identify potential improvements and derive information needs. In this iteration, we developed *Zeppelin*[4] (SANTOS; BARCELLOS; RUY, 2021), a diagnostic instrument that provides a panorama of CSE adoption in an organization by identifying the CSE practices it presently performs. Thus, it supports the identification of weaknesses and strengths, and helps identify information needs. From this study, we confirmed that knowing the organization's current state helps identify information needs that integrated data should meet and we learned that a diagnostic instrument like *Zeppelin* is a good way to apply at first hand to obtain an overview of the organization state and needs.

Considering the knowledge and the results obtained from the three learning iterations, we have developed *Immigrant*[5], an ontology-based approach to integrate application data aiming at supporting data-driven software development in CSE. In order to meet requirement *R4*, the approach considers a top-down and a bottom-up perspective. From the top-down perspective, information needs are used to drive data integration. From the bottom-up perspective, data available in the application repositories is considered to identify the information needs that can be met (*R3*). Thus, based on the information needs and available data, *Immigrant* integrates application data and provides meaningful dashboards to enable monitoring and insights aiming at data-driven software development.

To satisfy *R1*, *Califonia* (SANTOS; BARCELLOS; CALHAU, 2020) and *Zeppelin* (SANTOS; BARCELLOS; RUY, 2021) were integrated into *Immigrant*. As for *R2*, the approach uses ontologies from SEON to deal with semantic aspects. For that, we have developed new ontologies addressing CSE (specifically agile development, continuous integration, and continuous deployment) and added them to SEON as a subnetwork called *Continuum*. In order to satisfy *R4*, a data integration solution based on ontology, called *The Band*, was created and integrated into *Immigrant*. It provides integrated data, considering data available in the applications used by the organization to support the software development process.

After developing *Immigrant*, in the *Demonstration* step, a proof of concept was performed to show the feasibility of the proposed approach and, thus, in the *Validation* step, we performed a new learning iteration to answer the question: *Is Immigrant useful and is its use by software organizations feasible?* For that, the approach was applied in a case study in a software organization to evaluate and refine *Immigrant*.

---

[4] The name *Zeppelin* was chosen because the diagnosis instrument allows viewing an organization in a panoramic way, as if we were in a zeppelin seeing a city.

[5] The name *Immigrant* was inspired by the *Immigrant Song* by the Led Zeppelin band because the approach

1.4.3 Rigor Cycle

All the aforementioned steps have been based on the relevant literature, which includes papers, books, theses, standards, and other materials about Ontology, Ontology Network, CSE, Semantic Integration, Federated Information Systems, and Experimental Software Engineering. Therefore, the foundation of this work was based on the areas of knowledge mentioned above. Finally, the *Communication* step involves presenting the research results to the academic and industry communities. Some results have already been published in the following papers: (SANTOS; BARCELLOS; CALHAU, 2020; SANTOS et al., 2021; SANTOS; BARCELLOS; RUY, 2021; SANTOS; BARCELLOS; ALMEIDA, 2021; SANTOS; BARCELLOS; CALHAU, 2022; SANTOS et al., 2022).

Figure 1 summarizes the DSR cycles performed in this work.

1.5 Organization of this Thesis

This chapter presented the Introduction of the work, involving the general aspects, namely: the context and motivation for this research, the research hypothesis and objectives, and the adopted methodological approach. The next chapters are organized as follows:

- **Chapter 2 (Background)** presents the foundations required for grounding this research. It briefly reviews Agile, Continuous Software Engineering, Semantic Integration, Federated Information Systems, and the most relevant ontological notions applied in the work, including ontology classification and ontology networks. An introduction to the Software Engineering Ontology Network (SEON) (RUY et al., 2016) portions used in this work is also presented in this chapter.
- **Chapter 3 (*Continuum*: A Continuous Software Engineering Ontology (Sub) Network)** presents the current version of an ontology (sub)network that describes concepts and relationships of Continuous Software Engineering, called *Continuum*, which is integrated to SEON (RUY et al., 2016). The chapter presents *Continuum*'s architecture and its networked ontologies covering Scrum, Continuous Integration, and Continuous Deployment.
- **Chapter 4 (Learning Iterations Towards *Immigrant*)** presents the three learning iterations performed in the Design Cycle of this work to help us design the proposed approach.
- **Chapter 5 (*Immigrant*)**: presents the ontology-based approach to support data-driven software development proposed in this work. It provides an overview of the approach,

allows organizations to migrate from a non-data-driven software development process (land of ice and snow) to a data-driven software development (western shore).

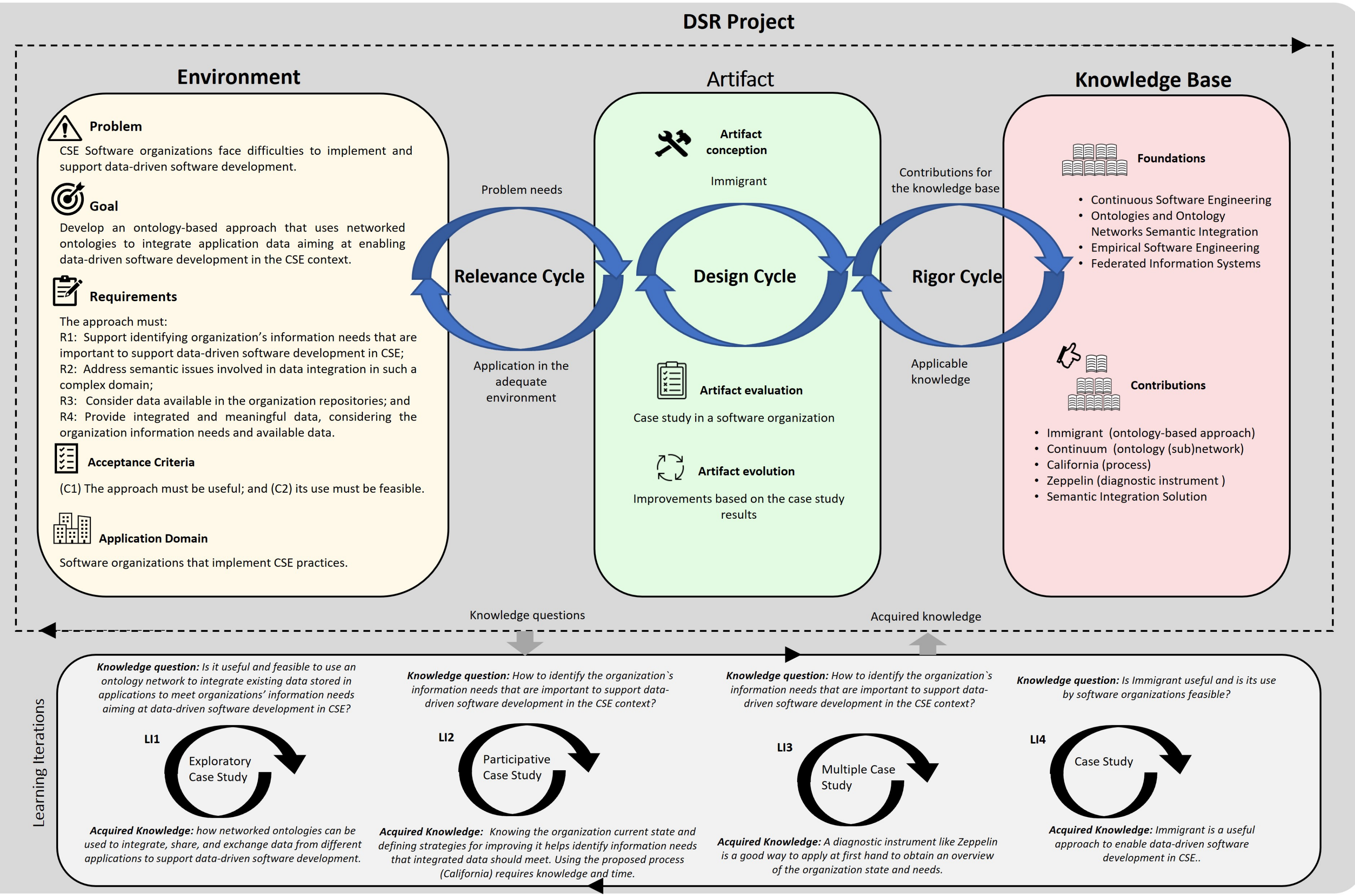


Figure 1 – Overview of the Design Science cycles in this research (based on (BARCELLOS et al., 2022)).

discusses how ON and Federated Information System architectures influence the proposed integration solution architecture, describes the process followed to create the integration solution architecture, and presents its components.

- **Chapter 6 (Final Learning Iteration: Applying *Immigrant* in a Software Organization)** presents the fourth, and last, learning interaction performed in this thesis. Its purpose was to evaluate *Immigrant* in a real-context.
- **Chapter 7 (Final Considerations)** summarizes the general ideas discussed in this thesis and the achieved results. It also describes the limitations and outlines some future work.
- **Appendix A (Zeppelin)** presents details about the *Zeppelin*'s Questionnaire.

2 Background

Oh, let the sun beat down upon my face and stars fill my dream. I'm a traveler of both time and space to be where I have been.

Led Zeppelin, Kashmir

This chapter presents an overview of the background for this work. It refers to the Rigor Cycle in the DSR approach adopted in this work, particularly to the use of existing and consolidated knowledge to ground the research and the proposed artifact. Section 2.1 concerns Continuous Software Engineering concepts and some proposals relevant to this work. Section 2.2 addresses Ontologies and Ontology Networks, presenting the parts of the *Unified Foundation Ontology* (UFO) (GUIZZARDI, 2005) and the *Software Engineering Ontology Network* (SEON) (RUY et al., 2016) used in this work. Section 2.3 regards basic notions of Semantic Integration. Section 2.4 introduces Federated Information Systems concepts considered to define the *Immigrant* architecture. Last, Section 2.5 closes the chapter.

2.1 Continuous Software Engineering (CSE)

Continuous Software Engineering (CSE) involves practices and tools that aim at establishing an end-to-end flow between customer demands and the fast delivery of a product or service. The 'big picture' by which this might be achieved goes beyond agile principles and surfaces a more holistic set of continuous activities (FITZGERALD; STOL, 2017). According to Johanssen et al. (2019), in CSE, customers are proactive, and users and other stakeholders are involved in the process, learning from usage data and feedback. Planning is continuous, so as requirements engineering, which focuses on features, modularized architecture and design, and fast realization of changes. Agile practices are employed, including short development cycles, continuous integration of work, continuous delivery and continuous deployment of releases. It relies on version control of code, branching strategies, fast commit, code coverage, and code reviews. Quality assurance involves automated tests, regular builds, pull requests, and audits. Knowledge is shared and continuous learning happens, capturing decisions and rationale.

In the last years, some works have addressed CSE processes and practices, providing an overview of CSE. We highlight four of them: *Stairway to Heaven Model* (OLSSON; ALAHYARI; BOSCH, 2012; KARVONEN et al., 2015), *Continuous* * (FITZGERALD; STOL, 2017), *Eye of CSE* (JOHANSSEN et al., 2019), and *Framework for Continuous Software Engineering* (FCSE) (BARCELLOS, 2020).

The Stairway to Heaven Model (StH) (OLSSON; ALAHYARI; BOSCH, 2012) describes a five-stage evolution path organizations follow to successfully move from traditional to customer data-driven software development. Figure 2 shows StH's stages. In summary, organizations evolving from traditional development start by experimenting with one or a few agile teams. Once these teams are successful, agile practices are adopted by the organization, turning it into an agile organization. As the organization starts showing the benefits of working agile, system integration and verification become involved and continuous integration is adopted. Once continuous integration runs internally, lead customers often express an interest to receive software functionality earlier than through the normal release cycle. They want continuous deployment of software. The final stage is R&D as innovation system, when the organization collects data from its customers and uses the installed customer base to run frequent feature experiments to support customer data-driven software development.

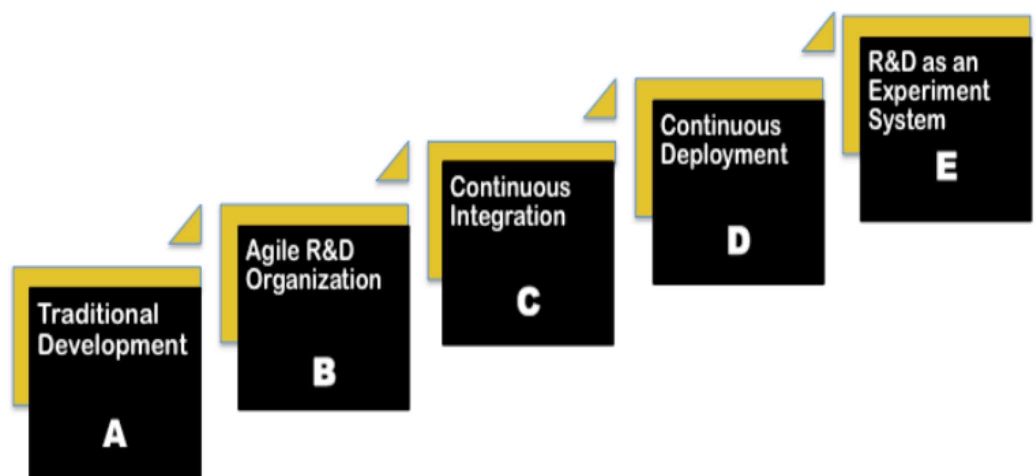


Figure 2 – Stairway to Heaven Model (StH) (OLSSON; ALAHYARI; BOSCH, 2012)

Fitzgerald & Stol (2017) argue that continuous activities go beyond software engineering activities. They introduce *'Continuous *'* as a set of activities from business, development, operations, and innovation that provides a holistic view of the software life cycle. Continuous planning, continuous security, continuous use, continuous trust, and continuous experimentation are some of the considered Continuous * activities. They introduce BizDev, analogous to DevOps, but referring to the continuity and alignment between business strategy and software development. Figure 3 presents Continuous * activities.

From interviews performed with CSE practitioners, Johanssen et al. (2019) defined the *Eye of CSE*, consisting of 33 elements (e.g., practices) organized in nine categories, as presented in Figure 4. According to the authors, the *Eye of CSE* can serve as a checklist for practitioners to tackle the subject of CSE by incrementally applying CSE elements and keeping an eye on potential next steps. The proposal differs from the sequential nature of the StH model (OLSSON; ALAHYARI; BOSCH, 2012) as the authors argue that even if some CSE elements, such as Continuous Integration and Delivery, require a step-wise introduction, CSE should be

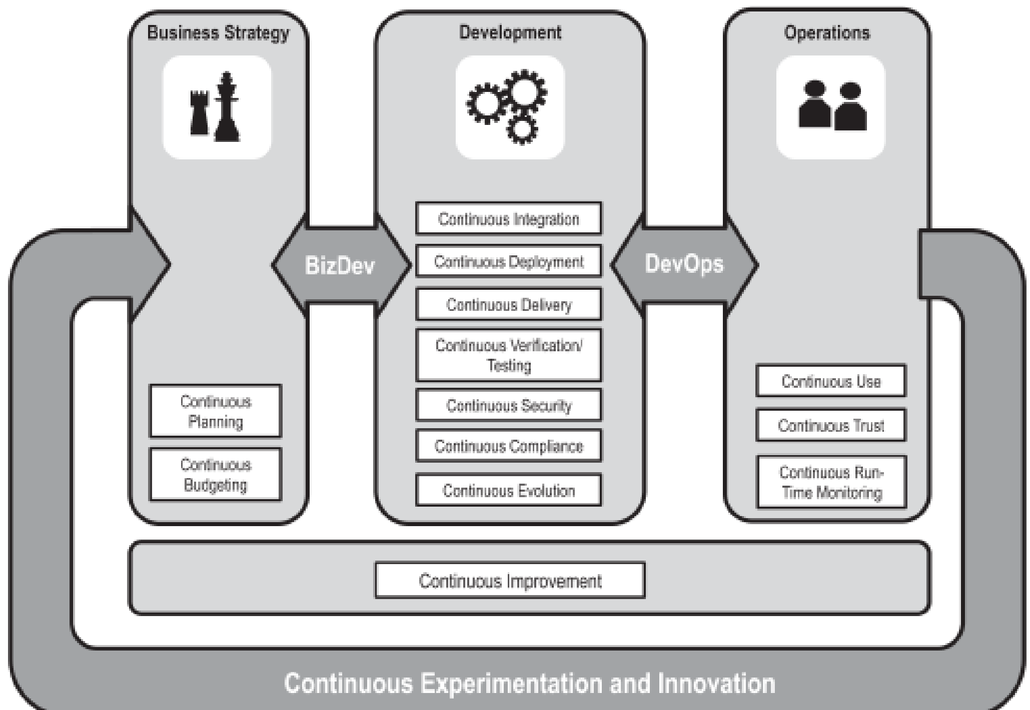


Figure 3 – Continuous * (FITZGERALD; STOL, 2017).

approached from multiple angles simultaneously.

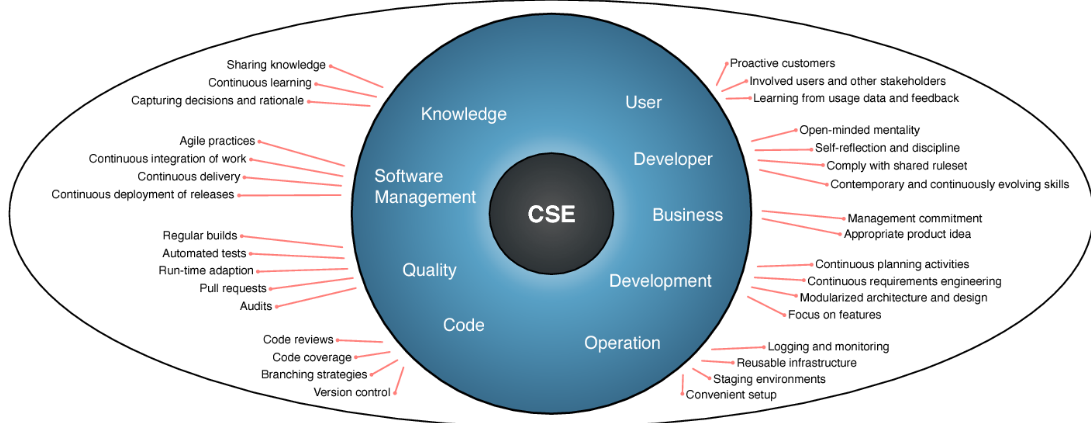


Figure 4 – The Eye of CSE (JOHANSSEN et al., 2019).

Finally, Barcellos (2020) proposed the *Framework for Continuous Software Engineering* (FCSE). FCSE includes a set of processes to be performed in the CSE context (e.g., agile development, continuous integration, continuous deployment, continuous software measurement, continuous knowledge management, and others) and the main relations (information flows and data flows) between them. Activities suggested in FCSE (BARCELLOS, 2020) were based on the elements from the Eye of CSE (JOHANSSEN et al., 2019) and StH stages (OLSSON; ALAHYARI; BOSCH, 2012; KARVONEN et al., 2015). Differently from StH, FCSE considers that processes can be performed simultaneously and gradually. Figure 5 presents FCSE.

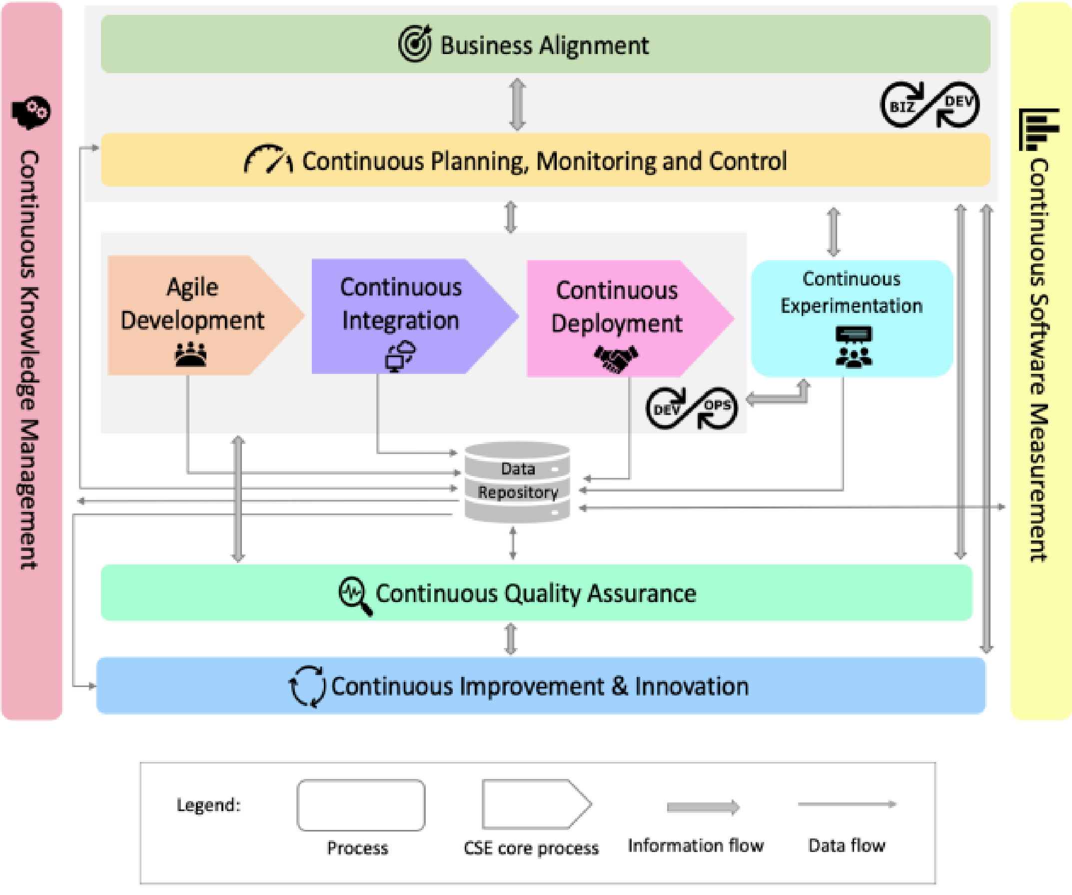


Figure 5 – Framework for Continuous Software Engineering (BARCELLOS, 2020).

In this work, we focus on three CSE processes, namely: Agile Development, Continuous Integration and Continuous Deployment. A brief background about them is presented in the next sections.

2.1.1 Agile Development and Scrum

Different flavors of the agile methods (e.g., Kanban (KNIBERG; SKARIN, 2010), Scrum (SCHWABER; KEN, 2013), Disciplined Agile Delivery (DaD) (AMBLER; LINES, 2012), eXtreme Programming (XP) (KNIBERG, 2015), Scaled Agile Framework (Safe) (PUTTA; PAASIVAARA; LASSENIUS, 2018), and Large-Scale Scrum (LeSS) (LARMAN; VODDE, 2016)) have become the *de facto* way of working in the software industry (RODRÍGUEZ et al., 2012). In allowing for more flexible ways of working with an emphasis on customer collaboration and speed of development, agile methods help organizations address many of the problems associated with traditional development (DYBÅ; DINGSØYR, 2008). The successful adoption of agile methods has also provided evidence of the need for greater flexibility and adaptation in the software development environment (PAPATHEOCHAROUS; ANDREOU, 2014).

In agile software development, the notion of cross-functional, multidisciplinary teams plays a central role. These teams have the different roles necessary to take a customer need all

the way to a delivered solution. Moreover, the notion of small, empowered teams, the backlog, daily stand up meetings, and sprints guide software development through shorter cycles, and bring the software development closer to the client (BOSCH, 2014).

Scrum is a software development process framework created with the assumption that software development is too complex and unpredictable to be fully planned at the beginning of a project. Therefore, it employs an iterative, incremental approach to optimize predictability and control risk (SCHWABER; SUTHERLAND, 2011). A Scrum team is a flexible, adaptive and small team (usually up to 7 people). Scrum teams are self-organized, cross-functional and capable of delivering products iteratively and incrementally, maximizing opportunity for continued feedback. A Scrum team is composed of a product owner, which is the role played by a person acting on behalf of the client and responsible for maximizing the value of the developed product, and a development team, which is responsible for developing the product. The development team, in turn, is composed of developers and a Scrum master. The Scrum master is a facilitator who ensures that the development team is provided with an adequate environment to complete the project successfully (SCHWABER; SUTHERLAND, 2011).

The Scrum process starts with initial planning to establish the product requirements and record them ordered in the product backlog (RISING; JANOFF, 2000). The product backlog is a document that contains the product requirements and it is never complete, and it can constantly change (SCHWABER; SUTHERLAND, 2011). The project is developed through incremental time-boxed cycles (usually lasting one month or less) called *sprints*. For each sprint, there is a sprint planning meeting, when the team selects from the product backlog the items to be addressed in the sprint and plans the work to be done. The planning result is recorded in the sprint backlog. A sprint produces a visible, usable, deliverable product that implements one or more user interactions with the system. The key idea behind a sprint is to deliver valuable functionality. Each product increment builds on previous increments. The goal is to complete the tasks defined in the sprint backlog by the sprint's delivery date and deliver an increment of a done product. An increment is said *done* if it is in conformance to established acceptance criteria and, thus, it can be delivered to the client.

As a time-boxed event, the end date for a sprint does not change. The team can reduce functionality to be delivered at the end of the sprint, but the delivery date cannot change (RISING; JANOFF, 2000). During the sprint, the team holds daily stand-up meetings aiming at optimizing the probability of the development team meeting the sprint goal. Before delivering the increment produced during a sprint, the team performs a sprint review meeting to inspect the increment and adapt the product backlog if needed. At the end of the sprint, there is a sprint retrospective meeting, when the team evaluates and reflects on itself and the project, regarding people, relationships, processes, and tools. As a result, a plan for improvement can be created. The meetings that occur during a sprint are known as *ceremonies* (SCHWABER; SUTHERLAND, 2011).

Agile methods, with their iterative approach, have transformed the way teams develop software, prioritizing continuous delivery of value to the customer. However, it is also necessary to connect different practices in a continuous and holistic software engineering flow. Thus, by integrating other approaches such as Kanban (KNIBERG; SKARIN, 2010) into agile practices, a new perspective focusing on continuity complements the agile view of software development. Kanban emphasizes fluidity and continuous workflow. Thus, it contributes to enhancing the effectiveness of agile practices, allowing for more adaptable and transparent management of the development process, and resulting in higher-quality and more efficient deliveries.

2.1.2 Continuous Integration (CI)

Continuous Integration (CI) is a widely established development practice in software development in which members of a team integrate and merge development work (e.g., code) frequently, for example, multiple times per day (DUVALL; MATYAS; GLOVER, 2007; HUMBLE; FARLEY, 2010; FOWLER, 2011; SHAHIN; BABAR; ZHU, 2017). CI enables software organizations to have shorter and more frequent releases cycle, improve software quality, reduce risk, and increase teams' productivity (FITZGERALD; STOL, 2017; SHAHIN; BABAR; ZHU, 2017). Figure 6 provides an overview of CI. The CI process is composed of the following activities (DUVALL; MATYAS; GLOVER, 2007; SHAHIN; BABAR; ZHU, 2017): Building, Testing, Inspection, and Feedback. Each activity is performed by a CI Server without human interaction.

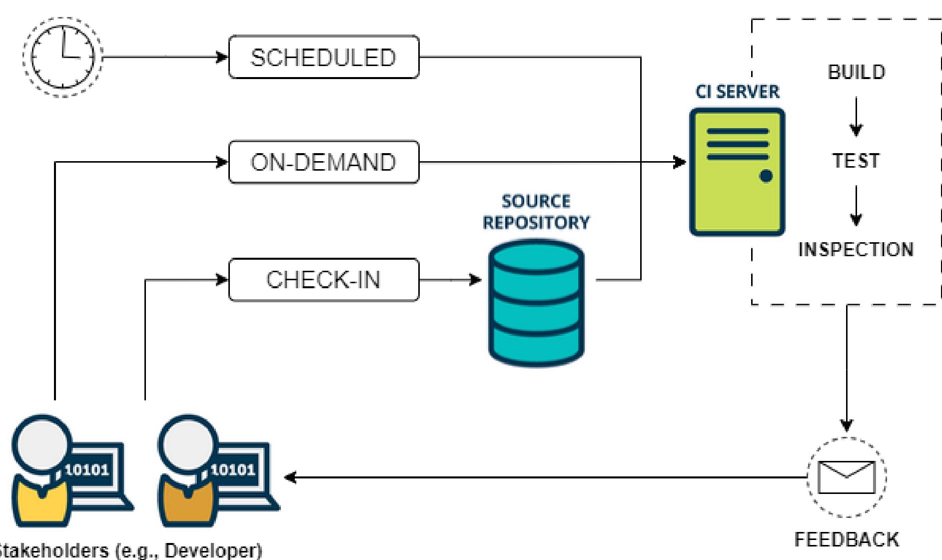


Figure 6 – Continuous Integration based on (DUVALL; MATYAS; GLOVER, 2007).

The CI process can be started by on-demand, scheduled, or check-in events. *On-demand events* happen when a stakeholder (e.g., developer) starts a CI process manually due to project needs. *Scheduled events*, in turn, start the CI process when a specific date or time is achieved. *Check-in* events occur when a developer creates, updates, or deletes a software artifact (e.g., source code of program, scripts, or database's configurations), in a local environment, and

sends a new/modified software artifact to a source repository used by a CI Server (DUVALL; MATYAS; GLOVER, 2007).

After the start event, the CI server executes the Build activity. First, Building creates a candidate version of the software, using the most recent version of the software artifacts (DUVALL; MATYAS; GLOVER, 2007; HUMBLE; FARLEY, 2010) in a branch (e.g., Master or Development). A candidate version is a new software that was built using a new version of software artifacts that were committed in a branch.

The CI server runs a set of automated tests (e.g., Unit Test and Integration Test) on the candidate version. Testing activities are essential for CI because without these activities the stakeholders (e.g., developers, clients, and sponsors) cannot trust the changes made in the software artifacts (DUVALL; MATYAS; GLOVER, 2007; HUMBLE; FARLEY, 2010). If there is no automated, continuous testing, there is no true CI. Without automated tests, it is difficult for developers or other project stakeholders to have confidence in software changes (HUMBLE; FARLEY, 2010). After Testing, Inspection is carried out to perform static or dynamic quality analysis of the software artifacts considering established quality rules/criteria (e.g., a class cannot have more than 300 lines of code without comments (MARTIN, 2009)).

One of the main purposes of CI is to produce feedback for the stakeholders involved in the development process. After all, the stakeholders want to know, as soon as possible, about problems in the build and, thus, fix them quickly. Therefore, along the other CI activities, the Feedback activity aims to inform (e.g., via email) the stakeholders about the status of the CI process. A predominant characteristic of a CI process is its running speed. This is necessary for CI to provide feedback in the shortest possible time (HUMBLE; FARLEY, 2010).

Modern source repositories (e.g., Github[1] and Gitlab[2]) are distributed, which means that a developer locally creates a "clone" of the entire repository (CONSERVANCY, 2023), via a *git pull* command. This means that each developer has a local copy, in the local source repository, of the remote source repository. The local copies can be "pushed" up to replace the remote repository, in the event of changes but also a crash or corruption, using a *git push* command (CONSERVANCY, 2023). Figure 7 illustrates different developers that have a local copy of the remote source repository in their machines.

A *Source Repository*, also called *Version Control Repository*, manages the changes in source code and other software artifacts (e.g., documentation or database scripts). In addition, it defines a single point of access to the project's software artifacts and allows a developer to view different versions of the various artifacts over time (DUVALL; MATYAS; GLOVER, 2007). A *Remote Source Repository* is a source repository that is located in a server while a *Local Source Repository* is a source repository in a developer's computer, which is typically a clone of a remote source repository.

[1] https://github.com/
[2] https://gitlab.com/

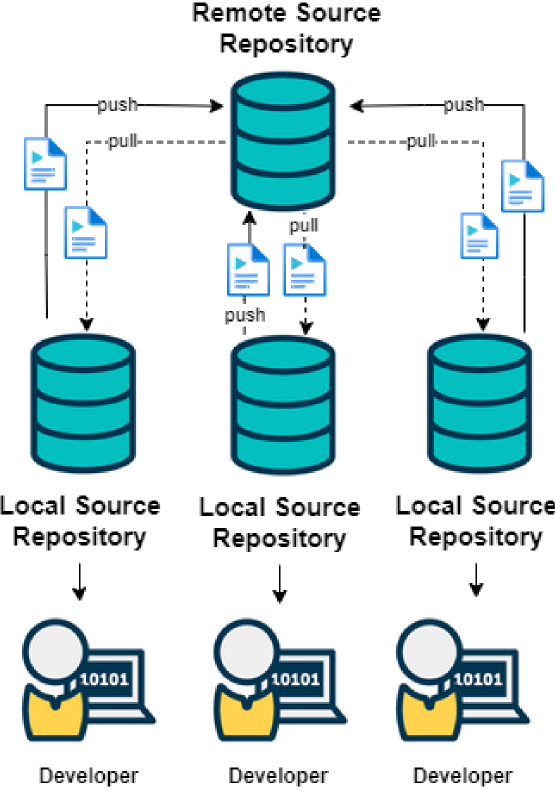


Figure 7 – Remote and Local Source Repository.

A developer executes the *git pull* command when he/she wants to incorporate changes from a remote repository into the local source repository (CONSERVANCY, 2023). After that, a developer creates, updates, or deletes a software artifact of a local branch, in a *working directory*, and takes a *snapshot* of the contents of one or more files under the local branch, executing *git add* command (CONSERVANCY, 2023). The snapshot is stored in a temporary called *Staging area*. This way, it is possible to stage only the change that a developer needs for the current commit and stage the changes in the code, made before the current commit, for the next commit. Staging Area is an intermediate area where commits can be formatted and reviewed before completing the commit (CONSERVANCY, 2023).

A *Commit* represents a snapshot of the source repository (GitHub, 2023). Changes, present in the staging area, are "committed" to the source repository, via *git commit* command. Commits should tell a story of the history of a source repository and how it came to be the way that it currently is. Figure 8 presents the relations between the *add* and *commit* commands with the *staging area* and *source repository*.

A commit contains the author's name and email address, a message the developer uses to explain the commit, and pointers to the commit, or commits, that directly came before this commit (its parent or parents): (i) zero parents for the initial commit, (ii) one parent for a normal commit, and (iii) multiple parents for a commit that results from a merge of two or more commits (CONSERVANCY, 2023). When a developer makes some changes and commits

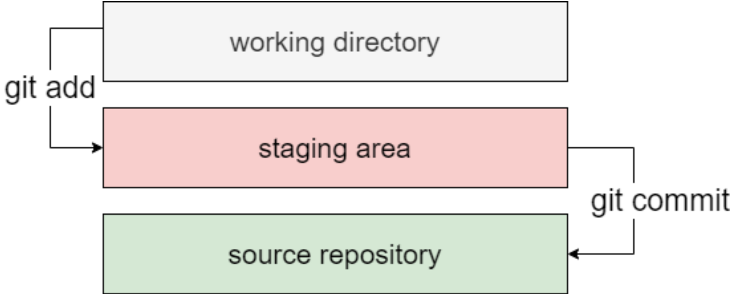


Figure 8 – Staging Area and Source Repository, based on (CONSERVANCY, 2023).

again, the next commit stores a pointer to the commit that came immediately before it. Figure 9 illustrates an example of a commit and its parents.

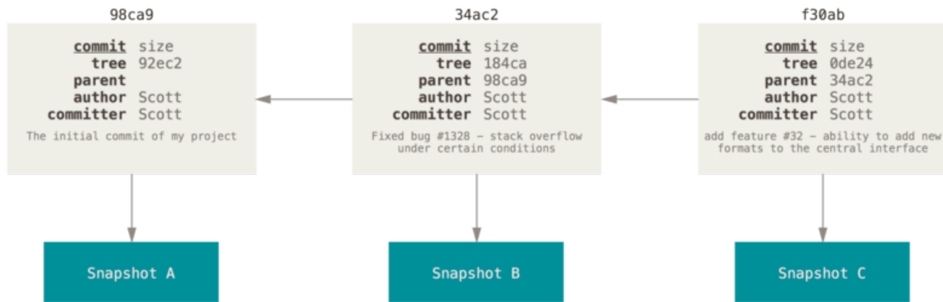


Figure 9 – Commits and their parents (CONSERVANCY, 2023).

A *branch* in the source repository is simply a lightweight movable pointer to one of these commits. The default branch name is usually *master* (or *main*). A developer starts making commits and he/she is given a master branch that points to the last commit he/she made. Every time a developer commits, the master branch pointer moves forward automatically. Figure 10 shows the relation between a branch and commits.

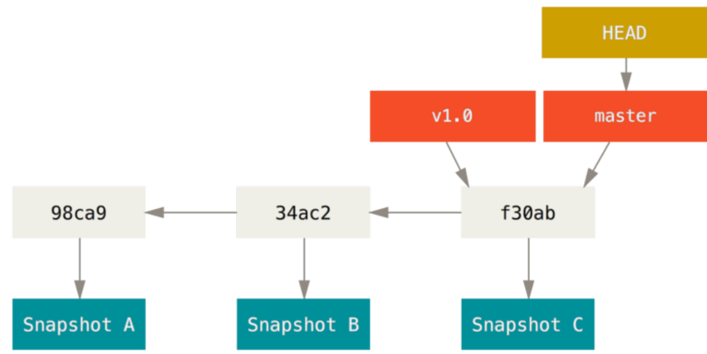


Figure 10 – Relation between branch and commits (CONSERVANCY, 2023).

A developer can create a new local branch from an existing local branch (GitHub, 2023) to work isolated from changes, as shown in Figure 11, using *git checkout* command. For example, a developer can make changes in the software artifacts in the “Little Feature” and “Big Feature” branches without impacting the “main” branch.

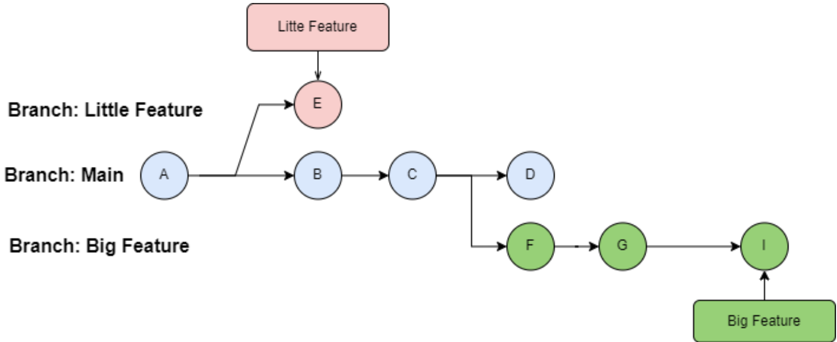


Figure 11 – Branches, based on (ATLASSIAN, 2023).

A developer uses the *git merge* command when he/she wants to apply the changes (e.g., create, delete, or update a source artifact) made in a source branch into a target branch (CONSERVANCY, 2023). For example, the changes made in "Big Feature" branch (source branch) are applied in the "Main" branch (target branch) when a developer uses *git merge* command between "Big Feature" branch (source branch) and "Main" branch (target branch). Observe in Figure 12 that "Big Feature" branch diverges from "Main" branch from since "C" commit until "I" commit. When a *git merge* is performed on these branches, the changes made in "Big Feature" (source branch) are applied in "Main" branch (target branch) then creating a new commit "H".

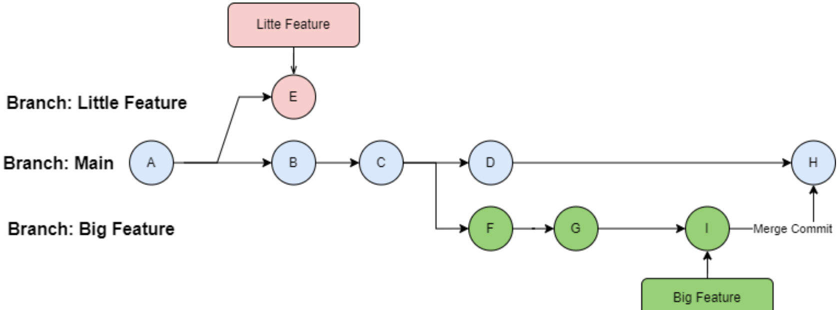


Figure 12 – Merge Commit, based on (CONSERVANCY, 2023).

The source repository verifies whether significant divergences (which prevent an automated merge) exist (e.g., the same software artifact with different content) between the artifacts present in the source and target branches, before executing the merge operation. Thus, the merge is not executed if any divergence is found. Otherwise, if no divergence was found, the source repository executes the merge. For example, if the "Little Feature" branch is behind the "main" branch, then by default it will fast-forward the "Little Feature" branch to match the "main". If the "Little Feature" and the "main" branch have diverged, then the source repository informs the developer about it and does not execute the merge between them.

A developer executes the *git push* command when he/she desires to update a remote branch using information about the local branch's changes (e.g., new branches or delete a software artifact). This way, a developer can synchronize the remote branch of the same local branch's name with local changes, and, thus, a developer shares the branch's changes with other project's developers. The remote source repository executes the *git pull* and *git merge*

commands to update the software artifacts. If the remote source repository finds a divergence between the updated and remote software artifacts, the merge process is stopped, and the developer is informed about the divergences and, thus, the remote software artifact is not updated.

Sometimes a developer desires that other developers evaluate the changes made in a branch, before it is incorporated into a remote branch. Then, a developer starts a process called *Code review*. A Code review is a process in which a developer sends local changes to the remote source repository via a review command[3] and another developer reviews the changes that were sent. A developer that receives and evaluates the updated changes is called *Reviewer*. A reviewer approves[4] the changes, in case he/she did not find any problems. The remote source repository executes *git pull* and *git merge*, after the changes be approved by a reviewer. Otherwise, the changes are not approved and the developer receives a message with reasons for non-approval. Finally, a *Check-in* event happens when a change is integrated into a remote branch without divergence, after a *git push* command or a *code review* process.

2.1.3 Continuous DElivery (CDE) and Continuous Deployment (CD)

Together with CI, Continuous DElivery (CDE) and Continuous Deployment (CD) are the software development industry practices that enable organizations to frequently and reliably release new features and products (SHAHIN; BABAR; ZHU, 2017). Continuous DElivery (CDE) aims at ensuring an application is always in a production-ready state after successfully passing automated tests and quality checks (WEBER; NEPAL; ZHU, 2016; HUMBLE, 2010). CDE employs a set of practices (e.g., CI practices), and deployment automation to deliver software automatically to a production-like environment (PUPPET, 2015). According to Chen (2015) and Humble (2022), this practice offers several benefits such as reduced deployment risk, lower costs, and shorter user feedback cycles. Continuous Deployment (CD), in turn, goes a step further and automatically and continuously deploys the application to production or customer environments (WEBER; NEPAL; ZHU, 2016). Figure 13 illustrates CI, CDE and CD. As shown, continuous delivery practices presuppose continuous integration practices.

There is a robust debate in academic and industrial circles about defining and distinguishing between Continuous Deployment and Continuous Delivery (FITZGERALD; STOL, 2017; WEBER; NEPAL; ZHU, 2016; HUMBLE, 2010). What differentiates Continuous Deployment from Continuous DElivery is the relation to a production environment (i.e., how actual customers are impacted): the goal of continuous deployment practice is to automatically and steadily deploy every change into the production environment (SHAHIN; BABAR; ZHU, 2017).

An other distinction between CD and CDE lies in the decision of when a new candidate

[3] In GitHub, this is called *pull request* while in GitLab this is called *merge request*.

[4] It is possible to configure a remote source repository to automatically approve updated changes, in case no divergence is found between the updated changes and remote software artifacts.

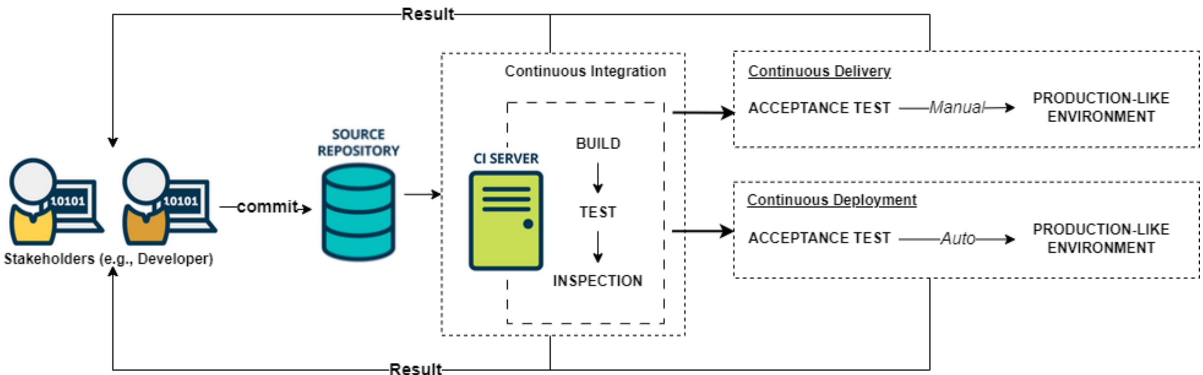


Figure 13 – The relationship between CI, CDE, and CD, based on (SHAHIN; BABAR; ZHU, 2017).

version of an application should be deployed to a production-like environment. A candidate version of an application is a new version of an application that a new code was integrated and approved in a CI process (HUMBLE; FARLEY, 2010). In the CDE, the deployment decision is driven by the needs of the business area. In other words, in the context of CDE, deploying a candidate version of an application is directly tied to decisions that can bring significant benefits to an organization. For example, introducing a new feature related to a marketing campaign of an organization, and launching it outside the campaign period, can have a direct impact on the company's revenue. Therefore, synchronizing the deployment cycle with business demands is a fundamental consideration in CDE. This concern is not present in the CD process, it is addressed to deploy a candidate version of an application in a production-like environment when a new candidate version of an application is approved in a CI process (HUMBLE; FARLEY, 2010; SHAHIN; BABAR; ZHU, 2017).

It is important to note that CD practice implies CDE practice but the inverse is not true (HUMBLE, 2010). Whilst the final deployment in CDE is a manual step, there should be no manual steps in CD. Typically, as soon as developers push commits to a remote source repository, the latest version is deployed to production through a deployment pipeline (SKELTON; O'DELL, 2016). The scope of CDE does not include frequent and automated release, and CD is consequently a continuation of CDE. Whilst CDE practice can be applied for all types of systems and organizations, CD practice may only be suitable for certain types of organizations or systems (HUMBLE, 2010; SKELTON; O'DELL, 2016; MUKHERJEE, 2016).

Finally, as continuous deployment is a critical aspect of software development process, various deployment strategies were developed to achieve the project and business needs:

- **Rolling/Ramped Deployment**: gradually replace instances of the old version with the new one, typically one at a time. This method ensures a smooth transition while monitoring the impact of the update on each instance before moving on to the next (FOWLER, 2014).

- **Blue-Green Deployment**: two identical environments, one "blue" (the current production) and one "green" (the new version), coexist. The new version is deployed to the green environment and tested thoroughly. The switch from blue to green is quick and reversible, allowing for minimal downtime and easy rollback if issues arise (HUMBLE; FARLEY, 2010; FOWLER, 2014).

- **Canary Deployment**: a strategy where a small subset of users or servers is selected to receive the new version of the software. This "canary" group serves as an early indicator of how the new release performs in a production environment. If the canary group experiences no issues, the deployment gradually expands to the entire user base or server pool (HUMBLE; FARLEY, 2010).

- **Feature Toggles (Feature Flags)**: toggles are a way to control specific features or functionalities within the software. These toggles can be turned on or off dynamically, enabling selective release of features without deploying an entirely new version (RAHMAN et al., 2016; FOWLER, 2023).

- **Shadow Deployment**: the new version runs alongside the existing one, with live traffic directed to the current version. The new version is primarily used for testing and comparison, allowing developers to identify issues and ensure compatibility before a full switch (BERCLAZ, 2023).

- **A/B Testing**: involves deploying different variations of a feature or user interface to different user groups. This strategy helps gather user feedback and performance data to make informed decisions about which variation to deploy to all users (HUMBLE; FARLEY, 2010; SHAHIN; BABAR; ZHU, 2017).

2.2 Ontology and Ontology Network

An ontology *"is a formal, explicit specification of a shared conceptualization"* (GRUBER, 1993). Here, "conceptualization" refers to an abstract model of some phenomenon in the real world that identifies the relevant concepts of this phenomenon; "explicit" means that the types of concepts used and the constraints imposed on their use are explicitly defined; "formal" refers to the fact that an ontology should be interpretable by machines; and "shared" reflects that ontologies must capture consensual knowledge accepted by a community (STUDER; BENJAMINS; FENSEL, 1998).

According to Scherp et al. (2011), ontologies can be organized in a three-layered architecture that discriminates among *foundational ontologies*, *core ontologies*, and *domain ontologies*. *Foundational ontologies* aim at modeling the very basic and general concepts and relations that make up the world (e.g., objects, events, participation, and parthood). They are generic across any area and are highly reusable in different modeling scenarios. *Core ontologies* provide a

refinement to foundational ontologies by adding detailed concepts and relations in a specific area (such as service, process, organizational structure) that still spans across various domains. *Domain ontologies* concern a particular domain in reality, such as a domain-specific medical ontology describing the anatomy of the human body. Domain ontologies can make use of, or be based on, foundational ontologies and core ontologies, by specialization of their concepts.

Another important distinction sets apart ontologies as conceptual models, called *reference ontologies*, from ontologies as computational artifacts, called *operational ontologies* (GUIZZARDI, 2007). A *reference ontology* is constructed with the goal of making the best possible description of the domain in reality, representing a model of consensus within a community, regardless of its computational properties. Once users have agreed on a common conceptualization, operational versions (machine-readable ontologies) of a reference ontology can be implemented. Differently from reference ontologies, *operational ontologies* are designed with the focus on guaranteeing desirable computational properties (FALBO, 2014).

For a complex domain, representing its knowledge as a single ontology results in a large and monolithic ontology that is hard to manipulate, use, and maintain (SUÁREZ-FIGUEROA et al., 2012). On the other hand, representing each subdomain in isolation is a costly task that leads to a very fragmented solution that is again hard to handle (RUY et al., 2016). In such cases, building an ontology network is a suitable approach (SUÁREZ-FIGUEROA et al., 2012). An *Ontology Network (ON)* is a collection of ontologies related together through a variety of relationships, such as alignment, modularization, and dependency, sharing concepts and relations with other ontologies (SUÁREZ-FIGUEROA et al., 2012). A *Networked Ontology (NO)*, in turn, is an ontology included in such a network, sharing concepts and relations with other ontologies (SUÁREZ-FIGUEROA et al., 2012).

As discussed in Chapter 1, this works aims to develop an ontology-based approach to integrate application data in the CSE context. For that, we have developed an ontology network called *Continuum*, which will be introduced in Chapter 3. *Continuum* uses concepts present in the *Unified Foundational Ontology* (UFO) (GUIZZARDI, 2005) (GUIZZARDI; FALBO; GUIZZARDI, 2008) (GUIZZARDI et al., 2013) and the *Software Engineering Ontology Network* (SEON[5]) (RUY et al., 2016). The next sections 2.2.1 and 2.2.2 describe, respectively, the fragments of UFO and SEON that were used to develop *Continuum*.

2.2.1 Unified Foundational Ontology (UFO)

The Unified Foundational Ontology (UFO) was developed based on a number of theories from Formal Ontology, Philosophical Logics, Philosophy of Language, Linguistics and Cognitive Psychology (GUIZZARDI, 2005). UFO is divided in three parts: (i) *UFO-A*, an ontology of endurants (objects) (GUIZZARDI, 2005; GUIZZARDI et al., 2015; GUIZZARDI et al., 2022), *UFO-*

[5] SEON specification is available in https://dev.nemo.inf.ufes.br/seon/

B, an ontology of perdurants (events) (GUIZZARDI et al., 2013; ALMEIDA; FALBO; GUIZZARDI, 2019), and *UFO-C*, an ontology of social entities (GUIZZARDI; FALBO; GUIZZARDI, 2008; GUIZZARDI; GUIZZARDI, 2011). The next paragraphs are dedicated to introducing the concepts of UFO that are adopted in this work. Details about UFO and its application are available in (GUIZZARDI, 2005; GUIZZARDI; FALBO; GUIZZARDI, 2008; GUIZZARDI et al., 2013; GUIZZARDI et al., 2022). In the sequel, UFO's concepts are written in *italics*.

UFO presents a fundamental distinction between the ontological categories of *Types* and *Individuals*. *Types* are patterns of features that are repeatable across different entities (GUIZZARDI et al., 2022). *Individuals* are particular entities that cannot be instantiated. Figure 14 shows a fragment of UFO with the concepts used in this work.

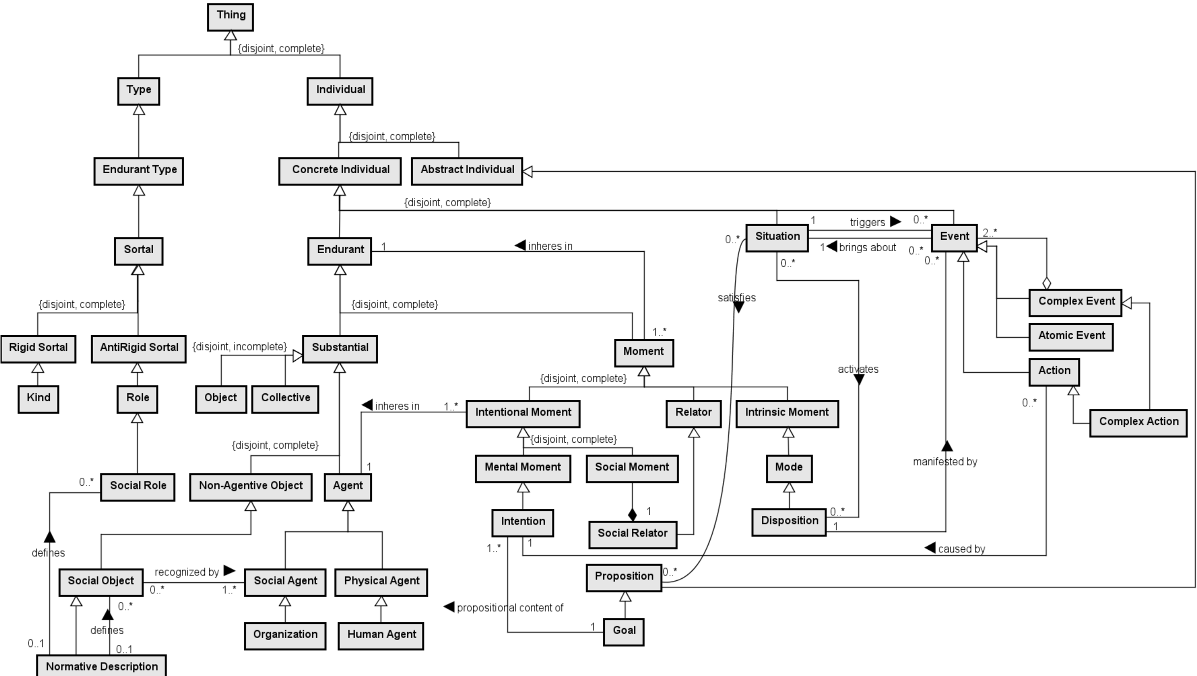


Figure 14 – A UFO fragment.

Endurants (e.g., Mick Jagger, the Moon, John's headache, Mary's marriage to Paul) are individuals that exist in time with all their parts (GUIZZARDI et al., 2022). They have essential and accidental properties and, hence, they can qualitatively change while maintaining their numerical identity (i.e., while remaining the same individual). The sorts of changes an endurant can undergo while maintaining its identity are defined by the unique *Kind* it instantiates (GUIZZARDI et al., 2022).

Substantials are existentially independent individuals. Examples include ordinary meso-scopic objects such as an individual person, a dog, a house, a hammer, a car, Alan Turing, and The Rolling Stones. *Moments* are *Endurants* that existentially depend on other individuals, i.e., *Moments* are 'parasitic' entities and can only exist by inhering in other entities (e.g., color, height, weight, electrical charge) (GUIZZARDI, 2005).

Intrinsic moments are existentially dependent on a single individual. *Intrinsic moments*

also include *modes* (GUIZZARDI, 2005). *Modes* can bear their own moments, including their own qualities, which can vary in independent ways. The category of modes includes *Dispositions* (e.g., functions, capabilities, capacities, vulnerabilities, etc.) as well as externally dependent entities (e.g., the love of John for Mary, the commitment of Paul towards Clara to meet for lunch next Friday). *Dispositions* are moments that may be manifested through the occurrence of events (possibly actions of intentional agents, such as Anna's speaking English) (CALHAU; AZEVEDO; ALMEIDA, 2021).

Perdurants (*Event*s) are individuals that unfold in time accumulating temporal parts (GUIZZARDI et al., 2022). Examples of events are a conversation, a football game, a symphony execution, a birthday party, the Second World War, and a particular business process (GUIZZARDI, 2005). *Events* can be *Atomic* or *Complex*, depending on their mereological structure, i.e., whilst atomic events have no proper parts, complex events are aggregations of at least two events (that can themselves be atomic or complex). *Events* are possible transformations from a portion of reality to another, i.e., they may change reality by changing the state of affairs from one (pre-state) situation to a (post-state) *Situation*. *Situations* can be used to represent certain configurations of entities that can be comprehended as a whole (ALMEIDA et al., 2020).

There are types that a substantial also instantiates in some circumstances, but not in others. This is the case of *Roles* (e.g., student and husband). A *Role* is a type instantiated in the context of a given event participation or of a given relation. A *Relator* is a *Moment* (i.e., an existentially dependent entity) that is an aggregation of *qua individuals*. A *Qua individual* is a *Mode* composed of other externally dependent modes that share the same bearer, the same source of external dependence, and the same foundational event (GUIZZARDI, 2005; GUIZZARDI et al., 2022). For example, the sum of all commitments and claims of John towards Mary form such a complex mode that we could call John-qua-husband-of-Mary, as they all inhere in John, are externally dependent of Mary, and are created in that particular wedding event (GUIZZARDI et al., 2022).

A *Relator* is, hence, existentially dependent on multiple individuals, namely, the bearers of its constituting qua individuals. Examples of relators include marriages, enrollments, employments, contracts, and presidential mandates (GUIZZARDI et al., 2022). Derived from relators we have *material relations*, which have material structure of their own and include examples such as 'working at', 'being enrolled in', and 'being connected to'.

UFO-C makes a distinction between *Agent* and *Non-Agentive Objects*. *Agents* can also be further specialized into *Human Agents* (Person) and *Social Agents*, which can be represented respectively, by *human beings*, computationally-based agents, and organizations or organizational units (departments, areas, and divisions). *Social agents* are composed by a number of other agents, which can themselves be human agents, artificial agents, or other institutional agents.

Agents are *substantials* that can bear special kinds of moments named *Intentional*

Moments. Intentionality should be understood in a much broader context than the notion of "intending something", but as the capacity of some properties of certain individuals to refer to possible situations of reality. Every *Intentional Moment* has a type (e.g., *Belief*, *Desire*, and *Intention*) and a propositional content. The latter being an abstract representation of a class of situations referred by that intentional moment. Thus, "intending something" is a specific type of intentionally termed *Intention*. *Non-Agentive Object* can also be further categorized in *Social Objects* (e.g., money, language, and Normative Descriptions).

Social Moments are *Intentional Moments* that are created by the exchange of communicative acts and the consequences of these exchanges (e.g., goal adoption, and delegation). For instance, suppose that John rents a car at a car rental agency. When signing a business agreement, John performs a communicative act (a promise), thereby creating commitments (social moments) for him and correlative claims (also social moments) for the rental agency.

Socials Relators are those composed of two or more pairs of associated commitments/-claims (*Social Moments*). A *Commitment* (internal or social) is fulfilled by an agent if this agent performs an action such that the post-state of that action is a situation that satisfies the propositional content of that commitment.

Actions are intentional events, i.e., events which instantiate a *Plan* with the specific purpose of satisfying (the propositional content of) some intention. Examples of actions include writing this thesis and the execution of a business process. As *Events*, *Actions* can be *Atomic* or *Complex*.

A *Complex Action* is composed of two or more participations. These participations can themselves be intentional (i.e., be themselves actions) or unintentional events. For example, the stabbing of Caesar by Brutus includes the intentional participation of Brutus and the unintentional participation of the knife. In other words, following philosophical action theories, we take that it is not the case that any participation of an agent is considered an action, but only those intentional participations — termed here *Action Contributions* (GUIZZARDI; FALBO; GUIZZARDI, 2008). Only agents (entities capable of bearing intentional moments) can perform actions, possibly with the partition of *Non-Agentive Objects*.

Finally, a *Complex Action* composed of action contributions of different agents is termed an *Interaction*. Two artists collaborating to create a sculpture is an example of an interaction. In the former case, the sculpture as well as the tools and raw materials used to create it are examples of *Non-Agentive Objects*. *Non-Agentive Objects* can participate in actions in different ways.

2.2.2 Software Engineering Ontology Network (SEON)

The *Software Engineering Ontology Network* (SEON) (RUY et al., 2016) is an ontology network that describes various subdomains of the Software Engineering domain. SEON orga-

nizes its ontologies according to the layers defined by Scherp et al. (2011), namely: foundational, core, and domain layers. Figure 15 presents SEON's Architecture.

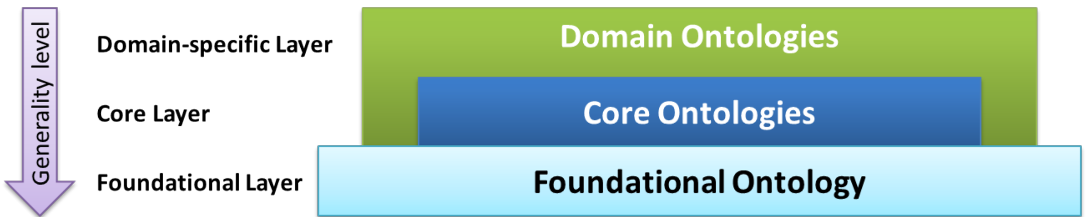


Figure 15 – SEON's Architecture (RUY et al., 2016)

At the *Foundational Layer* there is the *Unified Foundational Ontology (UFO)* (GUIZZARDI, 2005) whose distinctions are used for classifying SEON concepts, e.g., as objects, actions, commitments, agents, roles, goals, and so on. UFO provides the necessary grounding for the concepts and relations of all networked ontologies. At the *Core Layer* providing the SE core knowledge for the network, there are the Software Process Ontology (SPO) (BRINGUENTE; FALBO; GUIZZARDI, 2011; RUY, 2017) and the System and Software Ontology (SysSwO) (DUARTE et al., 2018b) (COSTA et al., 2022). SPO establishes a common conceptualization on software processes while SysSwO is about the nature of system and software, including, software artifacts, software constitution, software execution, computer system and hardware equipment. There are also two more general core ontologies, namely: the Enterprise Ontology (EO) (FALBO, 2014), and the Core Ontology on Measurements (COM) (BARCELLOS; FALBO; FRAUCHES, 2014). EO deals with aspects related to organizations, such as team membership, while COM defines concepts related to the measurement domain. Over the foundational and core layers, *Domain Ontologies* are grounded in core ontologies and in UFO, and encompass several SE subdomains (e.g., software requirements, design, configuration management, and software measurement).

For developing the *Continuous Software Engineering Ontology (sub)Network* (*Continuum*) proposed in this work, we reused concepts from the core ontologies SPO, EO and SysSwO, and also from the following domain ontologies: Reference Software Requirements Ontology (RSRO) (FALBO; NARDI, 2008), Configuration Management Process Ontology (CMPO) (CALHAU; FALBO, 2012), Reference Ontology on Software Testing (ROoST) (SOUZA; FALBO; VIJAYKUMAR, 2017), Quality Assurance Process Ontology (QAPO) (RUY, 2017), and Reference Ontology of Software Defects, Errors, and Failures (OSDEF) (DUARTE et al., 2018a). In the following, we present the SEON fragments reused to develop *Continuum*.

2.2.2.1 SPO, EO, and SysSwO

Figure 16 shows a fragment of SEON representing concepts from the three core ontologies relevant to this work. In the figure and in the following ones referring to SEON fragments, red double-dashed horizontal lines separate the layers of SEON architecture, describing depen-

dency between the layers, while black single-dashed lines separate concepts from different ontologies at the same layer. Concepts represented in soft gray are from UFO, in yellow belong to EO, those in green are from SPO, and in light blue are from SysSWO. In the text, SEON concepts are presented in *underlined italics*, UFO concepts are shown in *italics*.

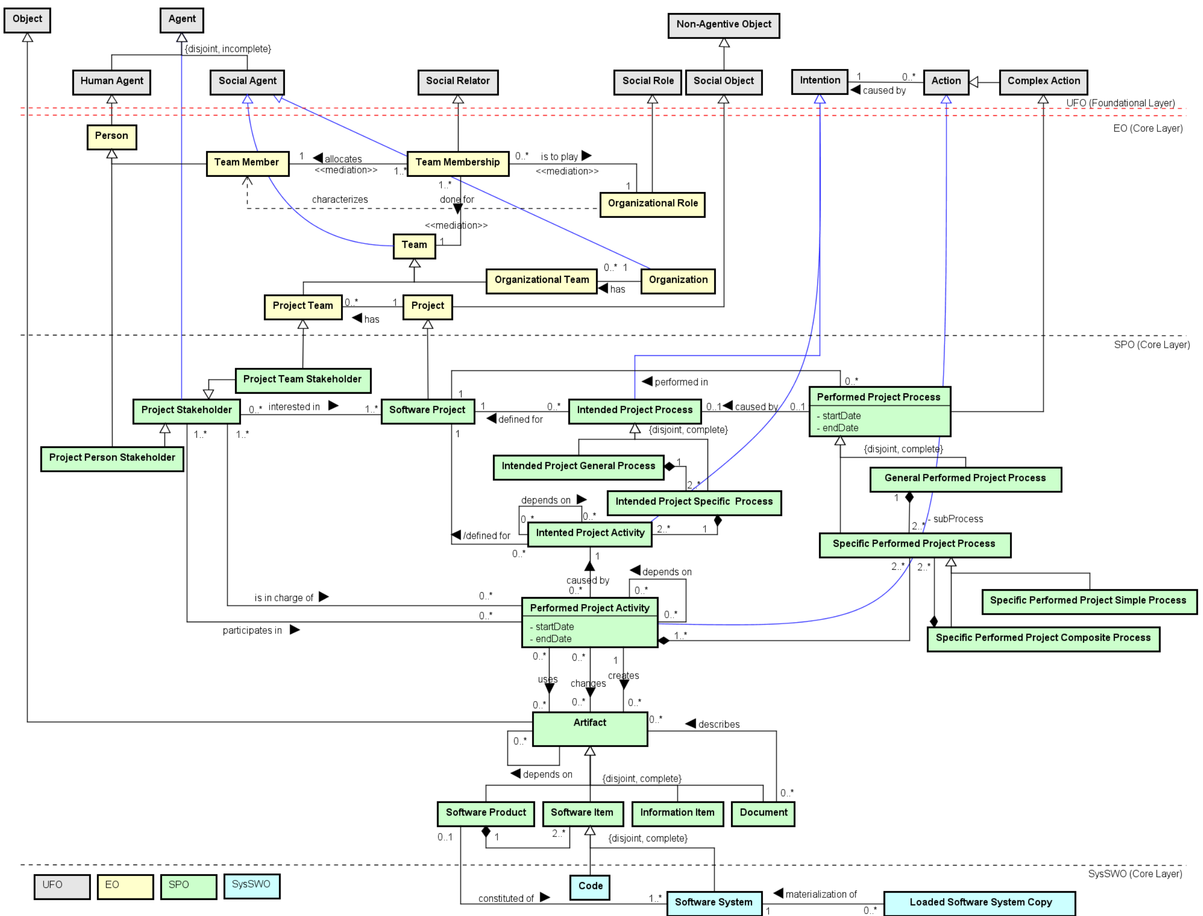


Figure 16 – A EO, SPO, and SysSWO fragment.

An *Organizational Role* is a *Social Role* recognized by the *Organization*, assigned to *Agents* when they are hired, included in a team, allocated or participating in activities. Project manager, designer, and programmer are examples of *Organization Roles* existing in an *Organization*.

A *Team Member* is a *Person* that plays an *Organizational Role* in a particular *Team*. A *Team* can be related to a *Project* (*Project Team*), e.g., the development team of a particular project at *Software Organization*, or to an *Organization* (*Organizational Team*), e.g., the marketing team of Software Organization. The allocation of a *Team Member* to play an *Organizational Role* in a *Team* is made through the social relation *Team Membership*. For example, a team membership can allocate John as a team member to play the programmer organizational role in the development team of a particular project in a software organization.

In the software domain, a *Software Project* is a *Project* related to software development or maintenance. An *Agent* interested in a particular *Software Project* is a *Project Stakeholder*.

It can be a *Project Person Stakeholder* (e.g., the project manager) or a *Project Team Stakeholder* (e.g., the project development team).

An important distinction in SPO is between *Intended* and *Performed Project Process*. The former refers to a process intended to be performed in the project, i.e., the process planned for that project. The latter refers to the process as actually executed in the project. Therefore, an intended process is understood as an intention to execute certain types of actions; in its turn, a performed process is understood as a complex action (an "occurrence") which may not correspond to the original intention.

An *Intended Project Process* can be a *General Intended Project Process*, which refers to the whole process defined for a project, or a *Specific Intended Project Process*, which is defined with a specific purpose for a project. The Project Management and the Requirements Engineering processes defined for a particular project in a software organization are examples of *Specific Intended Project Process*. The whole process comprising the Project Management, Requirements Engineering, Design, Implementation, and Test specific processes defined to that project is an example of *General Intended Project Process*. A *Specific Intended Project Process* is composed of a set of *Intended Project Activities* that support the achievement of the process purpose. For example, Requirements Elicitation and Requirements Documentation could be intended activities of the Requirements Engineering intended process.

Analogous to *Intended Project Process*, a *Performed Project Process* can be a *General Performed Project Process* or a *Specific Performed Project Process*, which is composed of *Performed Project Activities*. Performed processes and activities can be *caused by* intended processes and activities. For example, the intended activity Requirements Elicitation defined to a particular project could cause the execution of the Requirements Elicitation activity in that project, in the sense that the intention to perform an activity can result in performing the activity. A *Specific Performed Project Simple Process* is a *Specific Performed Project Process* that contains only activities while a *Specific Performed Project Composite Process* is a *Specific Performed Project Process* that contains two or more *Specific Performed Project Process*[6].

A *Project Stakeholder* can *participate in* or be *in charge of* a *Performed Project Activity*. The relation *in charge of* indicates that the *Project Stakeholder* was responsible for performing the *Performed Project Activity*. On the other hand, the relation *participates in* means that the *Project Stakeholder* contributed with the execution of the *Performed Project Activity*. For example, in a particular project, the system analyst can have been *in charge of* Requirements Elicitation, while the client can have participated in that activity.

Performed activities create, use or change *Artifacts* such as *Software Products*, *Software Items*, *Information Items* and *Documents*. *Software Product* refers to one or more computer programs together with any accompanying auxiliary items, such as documentation, delivered

[6] A *Specific Performed Project Simple Process* and *Specific Performed Project Composite Process* were developed in this work to support *Continuum*'s ontologies.

under a single name and ready for use (e.g., Eclipse IDE and MSWord). *Software Item* is a piece of software considered an intermediary result of the software process (e.g., a program, script, and database schema). *Code* is a *Software Item* representing a set of computer instructions and data definitions expressed in a programming language or in a form output by an assembler, compiler, or another translator.

Information Item refers to relevant information for human use in the software process context (e.g., a bug reported and a documented requirement). *Document* is any written or pictorial, uniquely identified, information related to the software process, usually presented in a predefined format (e.g., a Design Specification). Finally, *Documents* can describe other *Artifacts* (e.g., a Design Specification describes a software architecture).

Software System is a *Software Item* representing a set of computer instructions and data definitions expressed in a programming language or in a form output by an assembler, compiler, or other translator. A *Software System* is composed of one or more *Program*. A *Program* is a *Software Item* which aims at producing a certain result through execution on a computer, in a particular way, given by the Program Specification. A *Program* is constituted by *Code*, but it is not identical to code. *Code* can be changed without altering the identity of its program, which is anchored to the program's essential property: its intended specification (*Program Specification*). A *Loaded Software System Copy* is a *Disposition* that is a materialization of a *Software System*, inhering in a *Machine*.

Resource is a *Software Product* or *Hardware Equipment* (e.g., a smartphone being used by a Testing activity) when used by a process activity. A *Software Resource* and *Hardware Resource* occur when a *Software Product* and *Hardware Equipment*, respectively, are used as *Resource* of some process activity. Figure 17 shows a fragment of SPO presents the Resource concepts.

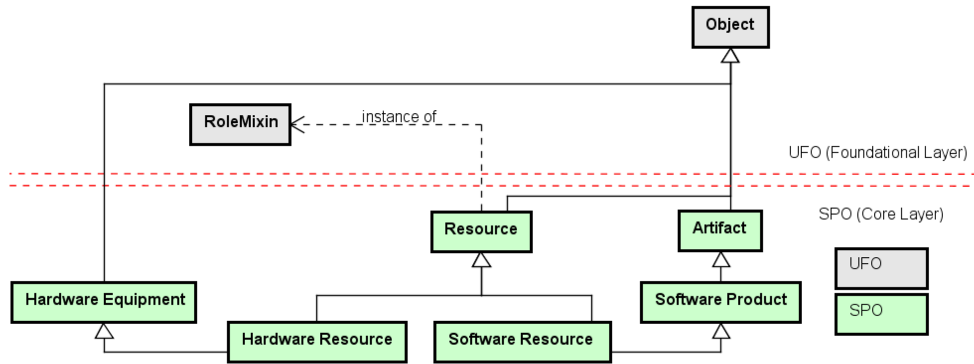


Figure 17 – A SPO fragment focuses on Resource concepts.

2.2.2.2 Configuration Management Process Ontology (CMPO)

CMPO aims at representing the activities, artifacts and stakeholders involved in the software Configuration Management Process. Since CMPO can be applied in the context of several SE subdomains, it describes some general notions applicable for diverse SEON concepts.

Figure 18 shows a fragment of SEON with CMPO, using a UML class diagram. Concepts represented in purple are belong to CMPO, those in green are to SPO, in light blue to SysSWO.

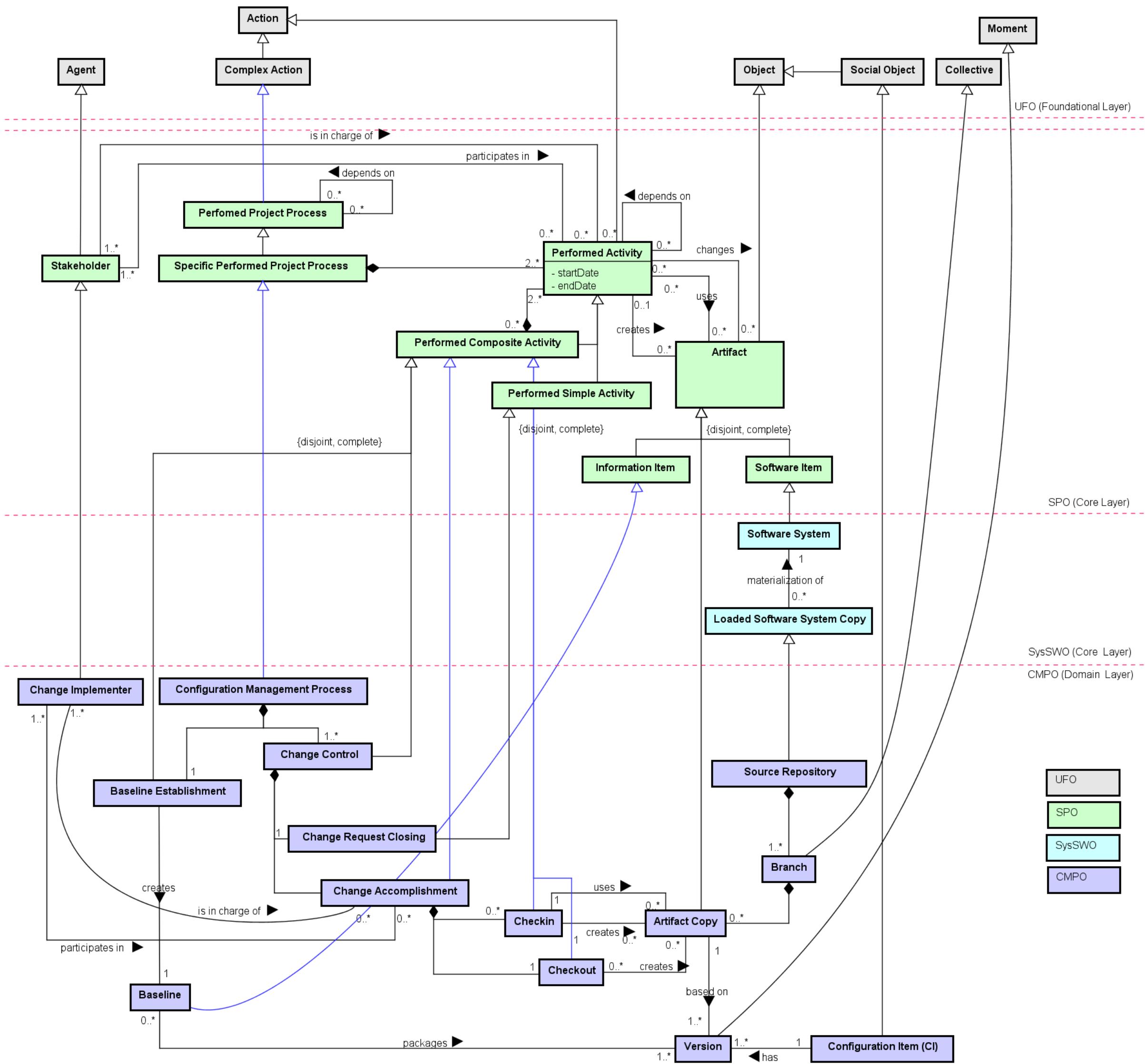


Figure 18 – CMPO fragment.

Configuration Management Process is a *Specific Performed Project Process* for conducting the activities related to software configuration management, ensuring the completeness and correctness of software *Configuration Items*. It is composed of: *Change Control* and *Baseline Establishment* activities. A *Change Control* is a *Performed Composite Activity* for formally controlling the modification of *Configuration Items*, in a process of requesting, evaluating, changing and reviewing. A *Change Control* is composed of following activities: *Change Accomplishment* and *Change Request Closing*. A *Change Request Closing* is *Performed Simple Activity* for closing a reviewed and approved Change Request.

Change Implementer is a *Stakeholder* responsible for implementing a change in the *Configuration Items* under *Version Control* in a *Change Accomplishment* (*Performed Composite Activity*) activity. A *Change Accomplishment* is a *Composite Performed Project Activity* that

performs authorized changes in a set of *Configuration Items* under version control. A *Configuration Item* is an object whose configuration is being managed, i.e., artifacts, process descriptions, tools under *Configuration Management*.

Change Accomplishment activity is composed of the following *Performed Composite Activities*: *Checkout* and *Checkin*. For example, a developer creates or updates a source code that implements a project's requirement. *Checkout* is an activity for accessing defined versions of a *Configuration Item (CI)* from a version control repository, usually for changing purposes, creating an *Artifact Copy* in an environment while a *Checkin* is an activity for including new *Versions* of *Configuration Items* into a *Source Repository*. *Artifact Copy* is a copy of an artifact (e.g., code or database script) that is under version control. A *Source Repository* is a *Loaded Computer System Copy* (e.g., an instance of GitLab installed in a Continuous Integration Server) whose purpose is to handle the changes of an *Artifact Copy* (e.g., copy of a source code).

Finally, a *Baseline Establishment* is a *Performed Composite Activity* for establishing a *Baseline* embracing a planned set of *Configuration Items' Versions*. A *Baseline* is an *Information Item* packaging a set of *Configuration Items' Versions* at a specific time in the product's life (for product delivery or to establish a relevant point in the Project).

2.2.2.3 Reference Ontology on Software Testing (ROoST)

ROoST is partially integrated to SEON, representing the activities, artifacts, and stakeholders involved in the Software Testing Process, considering only dynamic tests. Since the testing process is a technical process in software development, ROoST shares concepts with other SEON networked ontologies. Figure 19 shows a fragment of SEON with ROoST and SysSwO. The concepts represented in green belong to SPO, and those in light blue to SysSwO, light red belongs to ROoST.

Testing Process is a *Specific Performed Project Process* for planning and executing the dynamic testing activities for the software in development. A *Testing Process* has different *Level-Based Testing activity* (e.g., Unit Testing, System Testing, and Integration Testing). A *Level-Based Testing activity* is composed of the following *Performed Project activities*: *Test Coding* and *Test Execution*.

A *Test Coding* is a *Simple Performed Activity* for implementing the *Test Cases* as *Code* artifacts (*Test Code*) to be used during *Test Execution*. A *Test Case* is a *Document* containing the input data, expected results, steps, and general conditions for testing some situation regarding a *Code To Be Tested*. A *Test Code* is a *Code* produced for implementing a *Test Case* while a *Code To Be Tested* Portion of *Code* (software module) to be tested by a *Test Case*.

A *Performed Test Execution* is a *Performed Simple Activity* for effectively executing the *Test Cases*, by running the *Test Code* and producing the *Test Results*. A *Test Result* is a *Document*, containing the actual results, that describes *Fault (Runtime Defect)*, and identified issues relative

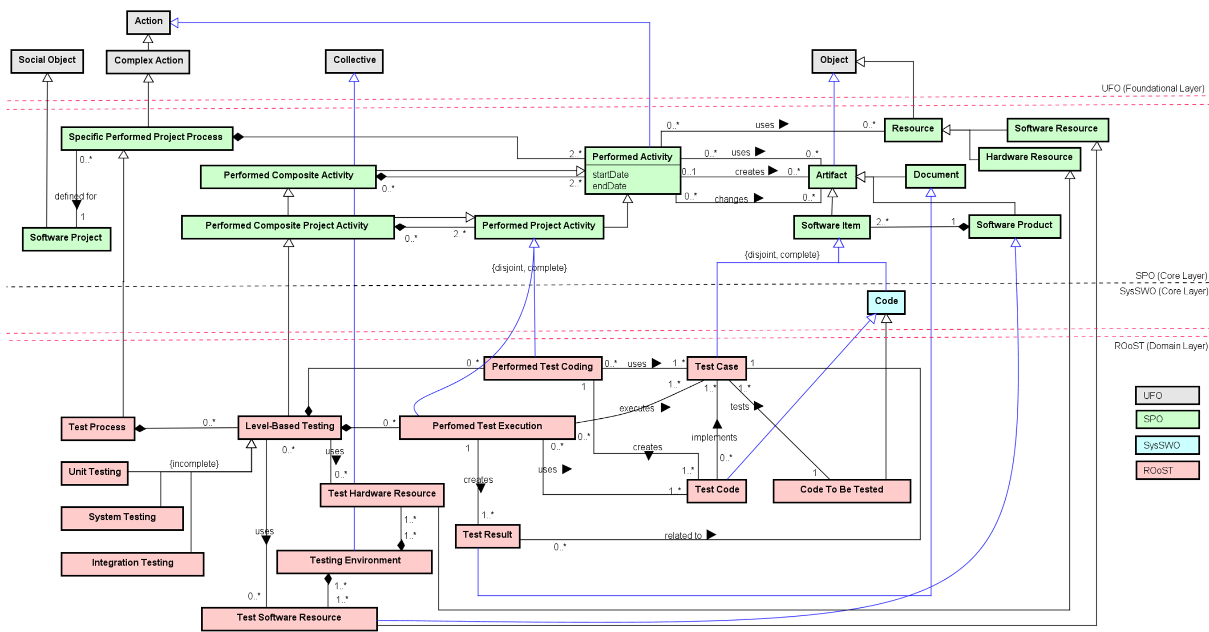


Figure 19 – ROoST fragment.

to a *Test Case* execution.

A *Test Process* is composed of the *Level-Based Testing*. A *Level-Based Testing* is a *Performed Composite Project Activity* consisting of testing activities and classified by the different levels they can be performed that can use a *Testing Environment*. *Unit Testing* is a *Level-based Testing* focusing on the unit or the individual components that have been developed, ensuring that the unit functions correctly in isolation. A *System Testing* is a *Level-based Testing* focusing on the behavior of the entire system, ensuring that it is in conformance with its requirements. An *Integration Testing* is a *Level-based Testing* focusing on larger components, ensuring that a collection of units functions as desired.

A *Testing Environment* is a set of resources (*Test Hardware Resources* and *Test Software Resources*) that are used to perform testing activities of a *Project*. A *Test Hardware Resource* is a *Hardware Resource* (e.g., a computer) while a *Test Software Resource* is a *Software Resource* (e.g., a continuous integration application) that comprises the *Testing Environment* of a *Software Project*.

2.2.2.4 Quality Assurance Process Ontology (QAPO)

QAPO aims at representing the activities, artifacts, and stakeholders involved in the Quality Assurance Process. Since QAPO can be applied in the context of several SE subdomains, it represents some general notions applicable for diverse SEON processes and artifacts. Figure 20 shows a fragment of SEON with QAPO and SPO, using a UML class diagram. Concepts represented in orange belong to QAPO and those in green belong to SPO.

Quality Assurance Process is a *Specific Performed Process* for conducting the activities

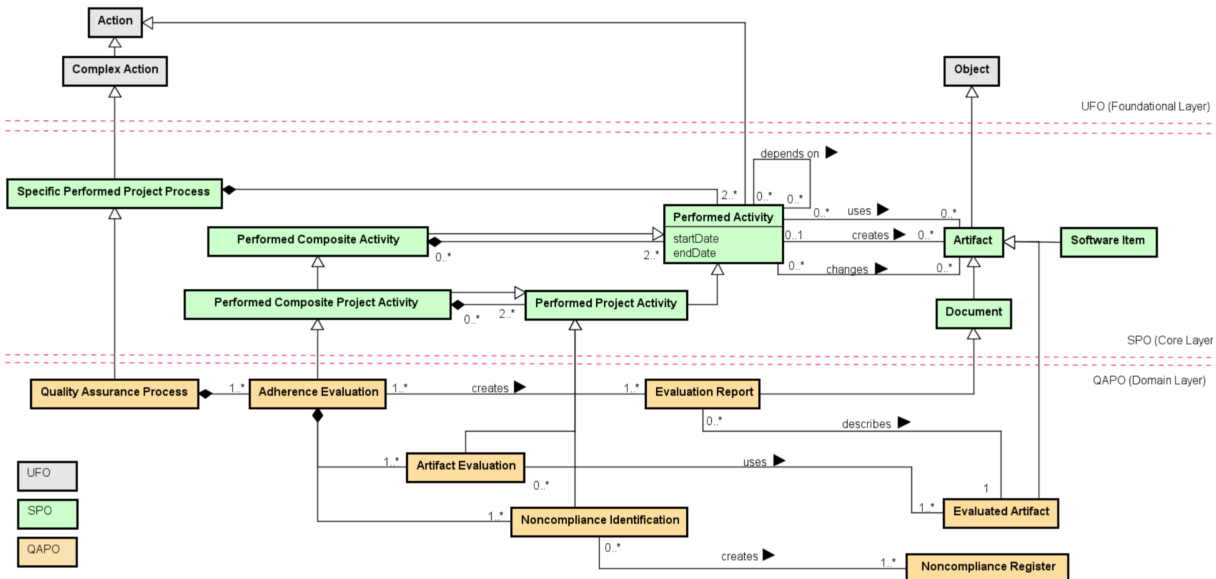


Figure 20 – A QAPO fragment.

related to software quality assurance, assessing and assuring adherence of the performed processes and produced artifacts to the applicable requirements. A *Quality Assurance Process* is composed of one or more *Adherence Evaluation* activities for objectively evaluating the adherence of processes and products to the applicable requirements, registering the identified issues, creating an *Evaluation Report*. An *Evaluation Report* is a *Document* describing the evaluation results and identified issues.

An *Adherence Evaluation* is composed of the *Performed Project Activities*: *Artifact Evaluation* and *Noncompliance Identification*. An *Artifact Evaluation* is an activity for objectively evaluating the adherence of products and deliverables (*Evaluated Artifact* to the applicable requirements while an *Noncompliance Identification* is an activity for registering the noncompliances identified in processes and artifacts in a *Noncompliance Register*. A *Noncompliance Register* (e.g., a neglected activity in a process, a document wrongly specified) is an *Information Item* describing a noncompliance (a failure or refusal to conform to an applicable requirement) in a process or artifact, and related information to solve it.

2.2.2.5 Reference Ontology of Software Defects, Errors, and Failures (OSDEF)

OSDEF is a domain ontology that provides a conceptualization about defects, errors, and failures in a software. Figure 21 shows a fragment of SEON with OSDEF, SysSWO, ROoSt, and SPO, using a UML class diagram. Concepts represented in dark yellow belong to OSDEF, those in green to SPO and those in light blue to SysSWO.

Failure is defined as an *Event* in which a program does not perform as intended, i.e., an event that hurts the goals of stakeholders. As Events, *Failures* are directly related with two distinct *Situations*, the first one is the *Situation* that exists prior to the occurrence of that *Failure* and that triggers the *Failure*. This *Situation* is represented in the ontology as a *Vulnerable State*

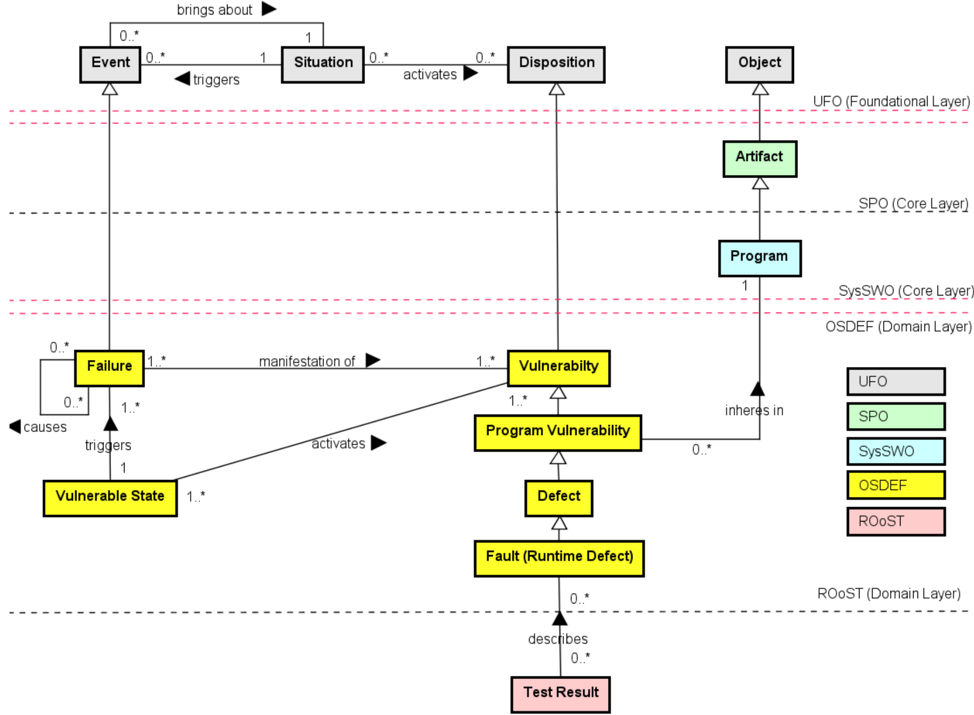


Figure 21 – A OSDEF fragment.

and denotes the situation that activates the *Disposition* that will be manifested in that *Failure* (DUARTE et al., 2018a). The second one is the *Situation* that is brought about by the occurrence of the *Failure*, which is defined in OSDEF as a *Failure State* (DUARTE et al., 2018a).

The occurrence of the failure transforms a portion of reality to another: in its pre-situation, the software is executing, it has the disposition to manifest the failure but the failure has not occurred yet, since the disposition was not yet activated; in its post-situation, the (failure) event was triggered and reality was "transformed" to a situation in which the software is not executing its functions (at least not as intended by stakeholders).

A *Defect* is a type of *Vulnerability* that can exist in *Programs* (DUARTE et al., 2018a). Some *Defects* can (accidentally) refrain from being manifested across software executions. When a *Defect* is manifested in a *Failure*, we term that Defect a *Fault (Runtime Defect)* (DUARTE et al., 2018a).

2.2.3 Reference Software Requirements Ontology (RSRO)

RSRO aims at being a reference for software requirements notions. RSRO is centered in the conception of requirement as a goal to be achieved and addresses the distinction between functional and non-functional requirements, how requirements are documented in proper artifacts, among others. Since RSRO provides the technical concepts for requirements, it is reused in other SEON networked ontologies. A *Document* composed of *Requirements Artifacts* that describe *Requirements* is said a *Requirements Document* (e.g., a Requirements Specification).

Requirements are goals to be achieved, representing a condition or capacity needed for the user (e.g., create service order). Figure 22 presents the RSRO fragment relevant to this work.

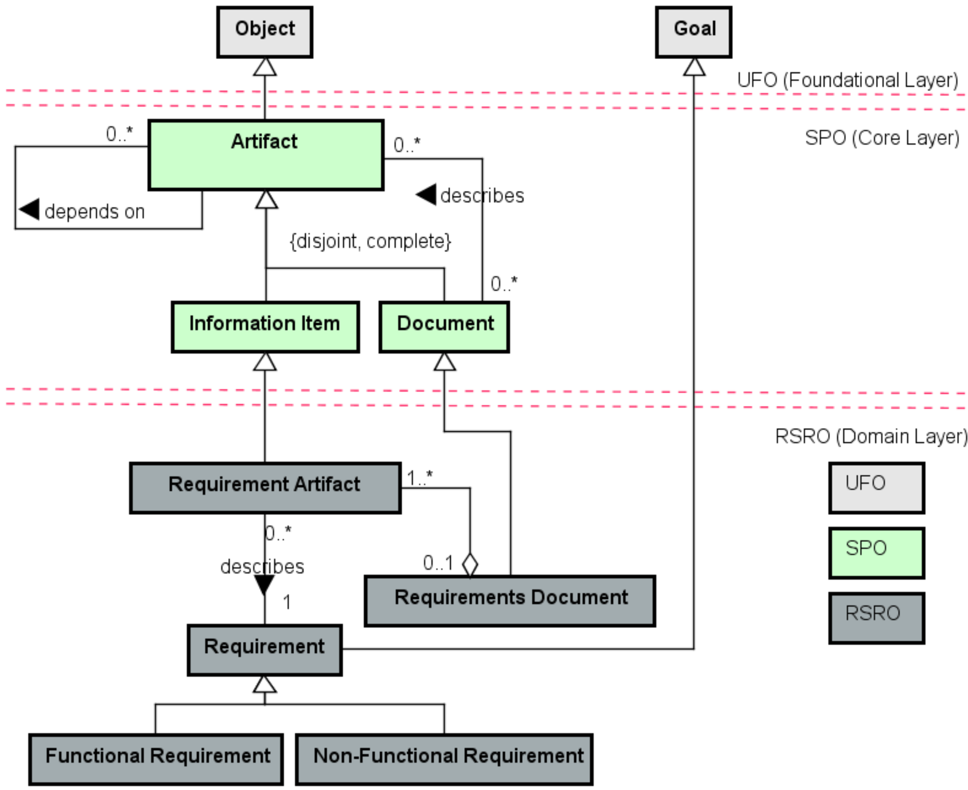


Figure 22 – A RSRO fragment.

Finally, a *Functional Requirement* is a *Requirement* defining a function to be available in the product being built (e.g., The need for the system to control client orders) and a *Non-Functional Requirement* is a *Requirement* defining criteria or capabilities for the product (e.g., Being accessible from some specific browsers, Being in conformance with a standard, and Performing a function in an established time).

2.3 Semantic Integration

Integration can be defined as the act to incorporate components into a complete set, conferring it some expected properties. The components are combined in a way to form a new system constituting a whole and creating synergy (IZZA, 2009). Interoperability, in turn, is the ability of application components to exchange or share data and services (WEGNER, 1996). Interoperability provides two or more business entities with the ability of exchanging or sharing information and of using functionality of one another in a distributed and heterogeneous environment. It preserves component systems as they are (VERNADAT, 2007). In this work, the term integration is adopted in broader sense, covering both integration and interoperability.

For integrating applications and properly supporting software-related processes, it is

necessary to create a coherent information system architecture in which the various software-related processes, data storages and applications are integrated so that they appear seamless from the point of view of the individual user (VERNADAT, 2007).

Integration is a complex task (THEMISTOCLEOUS; IRANI, 2004). Organizations have used an increasing number of applications to support software processes (FONSECA; BARCELLOS; FALBO, 2016). In general, these applications are based on different models, computing languages, platforms, and operating systems, which leads to integration challenges.

Integration can address different layers, namely: data, service, and process (IZZA, 2009). Data integration deals with moving or federating data between multiple data storages. Integration at this layer is based on bypassing application logic and manipulating data directly in the data store (e.g., a database, through its native interface). Message or service integration addresses messages exchange between the integrated applications. Any tier of an application, such as GUI, application logic or database, can originate or consume the message. Process integration views enterprises as a set of interrelated processes and it is responsible for handling message flows, implementing rules and defining the overall process execution. It constitutes the most complex integration approach.

Finally, integration can also consider different levels (IZZA, 2009). Integration at syntactical level addresses the way the data model and operation signatures are written down. Integration at semantic level, in turn, deals with the intended meaning of the concepts in a data schema or operation signature. Semantic integration requires us to contrast and harmonize the conceptualizations underlying applications to be integrated. In this setting, ontologies have an important role to play. For example, Calhau & Falbo (2010) proposed an Ontology-Based Approach for Semantic Integration (OBA-SI) that deals with the integration of applications at the data, services and process layers using ontologies to assign semantics to the structural and behavioral conceptual models of applications.

2.4 Federated Information Systems

The architecture of the *Immigrant* approach proposed in this work is inspired by some notions related to Federated Information Systems (FIS). A *Federated Information System (FIS)* is a set of distinct and autonomous information system components, the participants of a federation. Participants in first place operate independently, but have possibly given up some autonomy to participate in a federation (BUSSE et al., 1999). Figure 23 presents an example of the general three-tier architecture of a FIS.

As it can be observed in Figure 23, applications and users access a set of heterogeneous data sources through a *federation layer*. A federation layer is a software component that offers a uniform way to access data stored in different data sources. The uniformity is reached with a specific *interoperation strategy*, e.g. this layer can offer a *federated schema*, a *uniform*

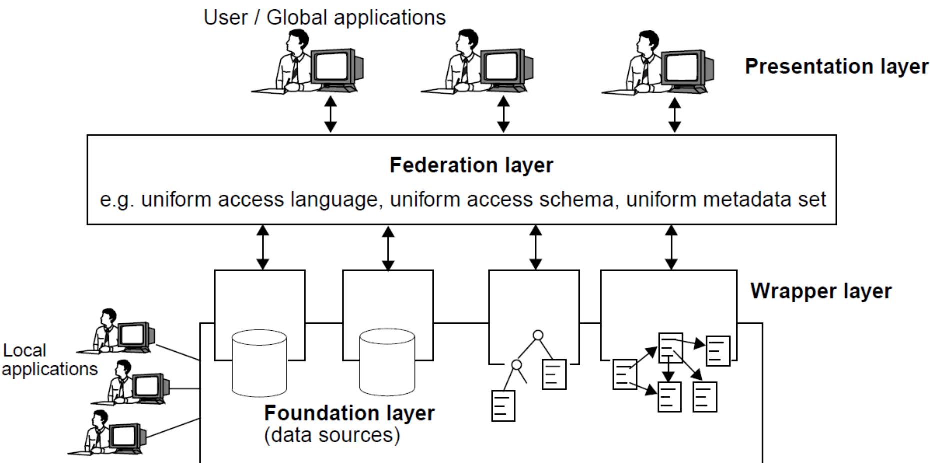


Figure 23 – A general three-tier architecture of a FIS (BUSSE et al., 1999).

query language, or a uniform set of source and content descriptions as *metadata sets*. The *data sources* are usually integrated into the infrastructure with *wrappers* that resolve some technical differences.

There are several types of FIS (BUSSE et al., 1999): Loosely Couple Information Systems, Federated Database Systems, and Mediator-based Information Systems. We are focused on the first two types of FIS. A *Loosely coupled information system* does not offer a federated schema, but only a multi-database query language to access the components. This has the advantage that components do not give up autonomy to participate in a federation. But no location and schema transparency is offered (the user has to address the respective component and the particular element in the schema of the component within her/his queries).

The federation layer is independent of the logical design of components. Since no global schema exists, changes of component schemas do not affect the system. But the missing logical integration leads to various dependencies between applications and component systems with all the negative effects on evolution known from two-tier systems (BUSSE et al., 1999).

A *Federated Database system* (FDBS) provides classical database system functionality. This includes read-and-write access for data management. The term 'database' indicates the relationship to classical database systems: components of federated database systems are structured sources, which are accessed through query languages (BUSSE et al., 1999).

Finally, Federated database systems are tightly coupled information systems. They are built bottom-up applying some schema integration techniques. The federated schema has to fulfill the requirements of completeness, correctness, minimality, and understandability, which is only possible with collection or fusion integration.

2.5 Final Considerations

This chapter presented the necessary background for this research. It was divided into four topics. The first one introduced concepts related to Continuous Software Engineering, presented some works that provide an overview of CSE by means of its stages, processes, activities, practices, and discussed the CSE processes that are the target of this work: Agile Software Development, Continuous Integration, Continuous Delivery, and Continuous Deployment. This background is particularly important to understand the conceptualization provided by *Continuum*.

The second topic was dedicated to presenting concepts related to Ontology and Ontology Network that were used to ground and create *Continuum*. Thus, we introduced fragments of UFO and SEON that were reused to create the *Continuum*'s networked ontologies.

In the third topic, we explored the foundations of semantic integration, pointed out different integration layers (data, service, and process), and the need to address semantics. Finally, in the last topic, we introduced concepts related to Federated Information Systems, which were used to define the architecture of *Immigrant*, our ontology-based approach to enable data-driven software development.

In the next chapter, we present *Continuum*, the ontology network developed in this work and used as a basis for *Immigrant*.

3 Continuum - A Continuous Software Engineering Ontology (sub)Network

As it was, then again it will be; though the course may change sometimes, rivers always lead to the sea.

Led Zeppelin, Ten Years Gone

This chapter presents the *Continuous Software Engineering ontology (sub)network* proposed in this work. It is related to the *Design Cycle*, since it proposes the ontologies that will be used as a basis for *Immigrant* (the proposed approach). The chapter is organized as follows: Section 3.1 provides and overview of *Continuum* and its architecture and discusses how *Continuum* was designed for assuring the necessary grounding for the networked ontologies. Section 3.2, Section 3.3, and Section 3.4 present the *Scrum Reference Ontology* (SRO), the *Continuous Integration Reference Ontology* (CIRO), and the *Continuous Deployment Reference Ontology* (CDRO), respectively. Section 3.5 discusses some related work and how *Continuum* contributes to the Continuous Software Engineering domain. Last, Section 3.6 presents the final considerations of the chapter.

3.1 *Continuum* Overview

Continuous Software Engineering (CSE) is a complex domain that involves Business, Software Engineering, Operations, and Innovation domains to deliver products or services that fulfill the customers' demand. Aiming to provide knowledge about CSE, we have worked on an ontology network, called *Continuum*[1], which aims at representing the conceptualization related to the processes involved in CSE. In the SE big picture, CSE appears as a (large) subdomain involving other subdomains. Thus, we developed *Continuum* as a subnetwork of SEON (RUY et al., 2016). In this sense, we reused some elements of SEON (such as its architecture, integration mechanisms, and networked ontologies) to develop *Continuum*.

Continuum can be used as a reference conceptual model to support different knowledge-related and interoperability solutions, such as communication, learning, standards harmonization, semantic documentation, and applications integration, among others. In this work, *Continuum* is used to aid data integration. It provides the conceptualization necessary to support semantic integration defined in *Immigrant*, i.e., *Continuum* has the conceptual models and axioms that are important to create some components of *Immigrant* (e.g., federated database

[1] Continuum: *something that changes in character gradually or in very slight stages without any clear dividing points*, from https://dictionary.cambridge.org/dictionary/english/continuum.

and federated services) and support semantic data integration aiming at data-driven software development.

Figure 24 shows the architecture of the current version of *Continuum*. Being a subnetwork of SEON, *Continuum* has the same three-layer architecture used in SEON. Thus, UFO (GUIZZARDI, 2005; GUIZZARDI et al., 2022) grounds core ontologies that, in turn, are used to define domain-specific ontologies. Domain-specific concepts can also be directly grounded in UFO. In Figure 24, each circle (network's node) represents an ontology. We represent only SEON ontologies directly used to develop the *Continuum* networked ontologies as presented in Chapter 2. Arrows denote the dependency relationships between networked ontologies, indicating that concepts from the target ontology are reused by the source ontology.

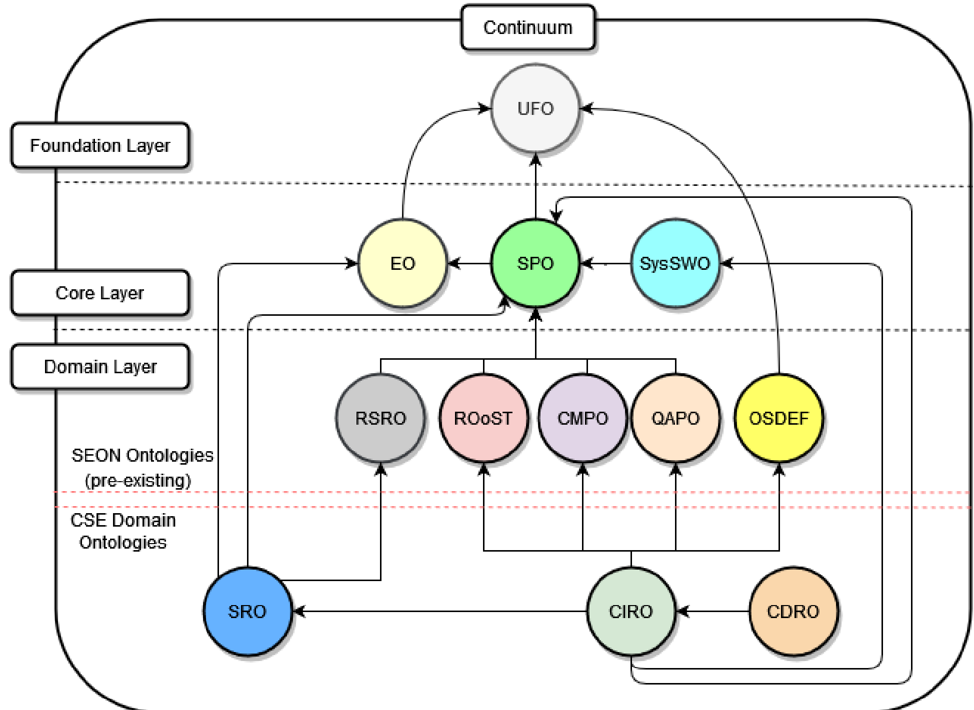


Figure 24 – *Continuum*'s architecture.

Continuum's domain ontologies proposed in this work are the *CSE Domain Ontologies* shown in Figure 24. They were developed following SABiO (FALBO, 2014), a systematic approach that guides the development of reference ontologies. As proposed in SABiO, functional requirements were established by means of competency questions, which are questions that the ontology must be able to answer and are used as a basis to build the ontology conceptual model.

The following sections present the three networked ontologies developed so far: *Scrum Reference Ontology* (SRO), which addresses aspects related to agile software development with Scrum, and *Continuous Integration Reference Ontology* (CIRO), which regards practices and other concepts related to continuous integration, and The *Continuous Deployment Reference Ontology* (CDRO), which concerns aspects related to continuous deployment. The following

conventions are used: SRO, CIRO, and CDRO concepts are written in **bold**, SEON concepts are written in *underlined italics*, UFO concepts are shown in *italics*, and examples are shown with underline. The colors used in models indicate the (sub)ontology to which a concept belongs. The black single-dashed horizontal lines separate concepts from different ontologies at the same layer. Red double-dashed lines separate the layers of SEON architecture. In the conceptual models, for better visualization, we omit UFO concepts that ground SEON concepts already presented in Chapter 2.

3.2 Scrum Reference Ontology (SRO)

The *Scrum Reference Ontology* (SRO) consolidates reference literature on the topic, using as main sources the works of Schwaber & Beedle (2002), Cohn (2010), Schwaber & Sutherland (2011), Rubin (2012), Satpathy et al. (2016). SRO is organized into five subontologies:

- The *Scrum Process subontology*, which addresses the events that occur in a project that adopts Scrum, such as the Scrum ceremonies;
- The *Scrum Stakeholders subontology*, which concerns the teams, agents, and roles involved in a Scrum project;
- The *Scrum Stakeholders Participation subontology*, which deals with the participation of stakeholders in the events of a Scrum project;
- The *Product and Sprint Backlog subontology*, which addresses aspects related to the requirements established in a Scrum project and activities planned to materialize them;
- The *Scrum Deliverables subontology*, which focuses on the results produced during a Scrum project.

Figure 25 shows an overview of SRO. In the figure, each package inside the SRO package represents a subontology of SRO. A Dependency relationship indicates that one (sub)ontology reuses concepts from another. This convention is also used in the other ontologies presented in this chapter.

3.2.1 Scrum Process subontology

The *Scrum Process subontology* aims to answer the following competency questions:

- CQ01. Which processes and activities make up a Scrum process?
- CQ02. In a Scrum project, on which other activities/processes did a certain activity/process depend?

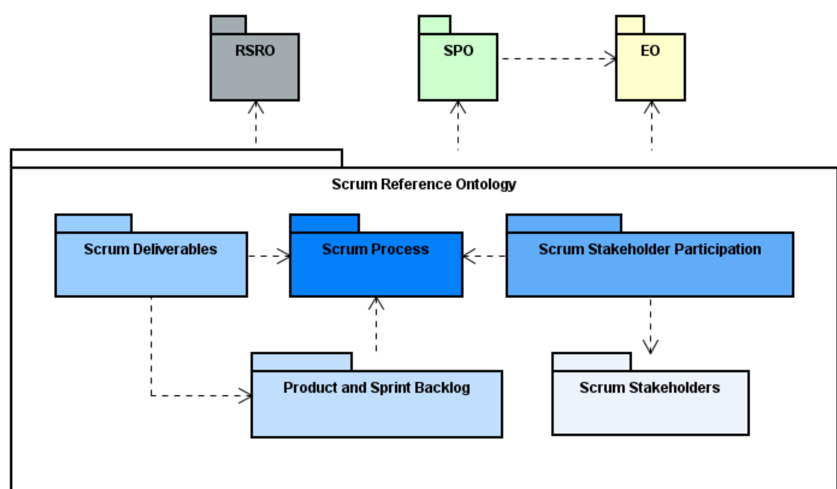


Figure 25 – SRO's architecture.

- CQ03. How many sprints were performed in a Scrum project?
- CQ04. What ceremonies were performed in a sprint?
- CQ05. What development tasks were performed in a sprint?
- CQ06. When did a Scrum project start?
- CQ07. When did a Scrum project end?
- CQ08. When did a Scrum process start?
- CQ09. When did a Scrum process end?
- CQ10. When did a Scrum project activity start?
- CQ11. When did a Scrum project activity end?

CQ01 and CQ02 regard processes and activities involved in a project adopting Scrum. They aim to provide knowledge about the Scrum process structure and the order in which its subprocesses and activities occured. CQ03 to CQ05 concern information about a particular project or sprint. For example, from CQ04 a Scrum master can identify whether a ceremony was not performed in a sprint and thus can verify the reasons and act accordingly (RUBIN, 2012; SATPATHY et al., 2016). Moreover, by answering these CQs for many projects, it is possible to get consolidated information, such as the average number of sprints performed in projects of a certain organization. CQ06 to CQ11 refer to temporal aspects (RUBIN, 2012; SATPATHY et al., 2016). They provide information about processes and activities duration and allow for project performance analysis. Figure 26 shows the Scrum Process subontology conceptual model. In the figures depicting SRO conceptual models, a red double-dashed line separates SEON and SRO concepts. This convention is also used in the other conceptual models presented in this chapter.

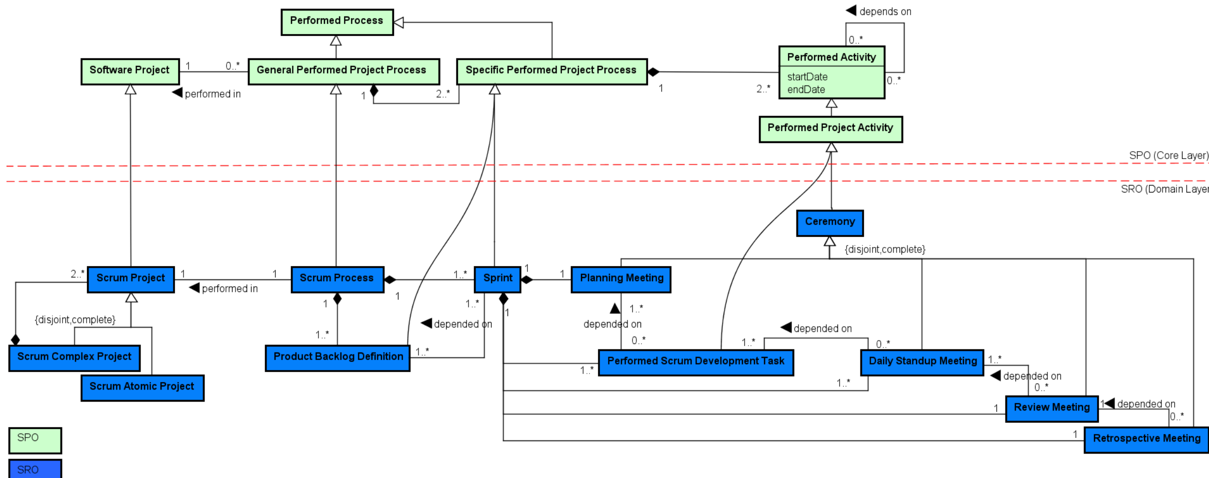


Figure 26 – Scrum Process Subontology.

Since *Continuum* was developed to support CSE data integration, each *Continuum*'s ontology concerns events performed in the CSE context (e.g., activities carried out in a project, people who carried out such activities, and results that were produced in the performed activities). Because of this, in the *Continuum* ontologies, the verbs used to describe relationships are in the past tense. After all, *Immigrant* integrates data about events that already occurred in software projects. (The same convention was not applied in the reused SEON ontologies, retaining their original terminology. We avoid changes in terminology to maintain compatibility among *Continuum* and other works that used SEON.)

A **Scrum Project** is a *Software Project* that adopts Scrum in its process (**Scrum Process**). A **Scrum Process** is a *General Performed Project Process* (i.e., it is an overall process performed in a project) composed of two types of *Specific Performed Project Process*: **Product Backlog Definition**, which aims at defining and prioritizing the functionalities to be produced in the **Scrum Project**, and two or more **Sprints**, which occur after the **Product Backlog Definition** and aims at developing the product.

A **Sprint** is composed of following types of *Performed Project Activities*: **Ceremonies** and **Performed Scrum Development Tasks**. The ceremonies that compose a Sprint are **Planning Meeting**, **Daily Standup Meeting**, **Review Meeting**, and **Retrospective Meeting**. Dependency relations (*depends on* relation between *Performed Project Activities*) establish the order in which these ceremonies occured. For example, **Performed Scrum Development Task** depends on **Planning Meeting**, because a **Performed Scrum Development Task** refers to the execution of a task planned in a **Planning Meeting**. **Daily Standup Meeting**, in turn, depends on **Performed Scrum Development Task**, because a **Daily Standup Meeting** occurs after the execution of the development tasks discussed in that meeting.

3.2.2 Scrum Stakeholders Subontology

The *Scrum Stakeholders subontology* aims to answer the competency questions CQ12 to CQ16:

- CQ12. What roles are involved in a Scrum project?
- CQ13. What teams are involved in a Scrum project?
- CQ14. Which roles are involved in a team in a Scrum project?
- CQ15. Who are the members of a team in a Scrum project?
- CQ16. Which role is played by a team member in a Scrum project?

CQ12 to QC14 focus on roles and teams involved in Scrum projects. CQ15 and CQ16, in turn, allow identifying who is allocated to a certain team and the role he/she plays in that team (SATPATHY et al., 2016). This information is important to identify the team members of Scrum projects and avoid over-allocation to the same person. Moreover, when performing new allocations, it is possible to look at allocation historical data to verify people's profiles and allocates them to play roles accordingly. The diagram of the Scrum Stakeholders subontology is shown in Figure 27.

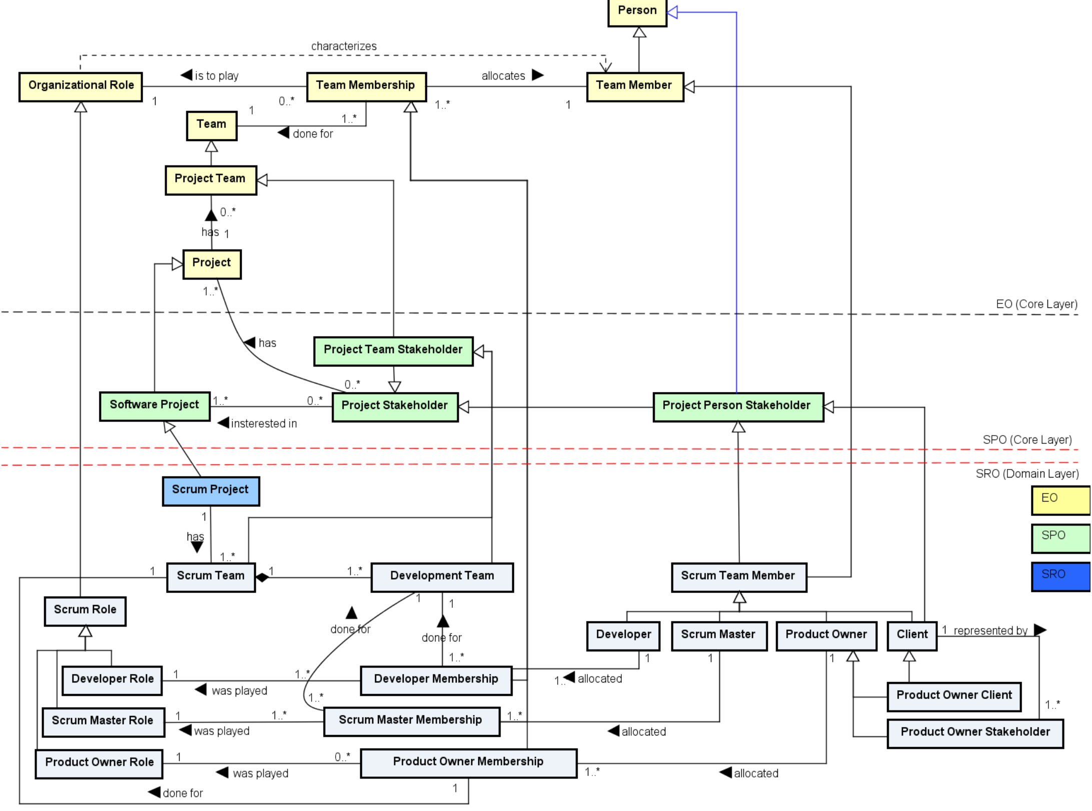


Figure 27 – Scrum Stakeholders Subontology.

A **Scrum Team Member** is a *Project Person Stakeholder* interested in a **Scrum Project**. A **Scrum Team Member** is allocated to a **Scrum Team** (a *Project Team Stakeholder* interested in a **Scrum Project**) to play the *Organizational Role* of **Product Owner Role**, **Scrum Master Role** or **Developer Role**. As explained in the Enterprise Ontology context (see Section 2.2.2), *Team Memberships* allocate *Team Members* to *Teams*. Thus, **Product Owner Membership** allocates a *Project Person Stakeholder* to play the **Product Owner Role** in a **Scrum Team**. This *Project Person Stakeholder* is called **Product Owner**. For example, if John is allocated to play a **Product Owner Role** in the **Scrum Team** *st* of the **Scrum Project** *sp*, it means that John is the **Product Owner** in *st*. Analogously, **Scrum Master Membership** and **Developer Membership** allocate, respectively, a **Scrum Master** and a **Developer** to a **Development Team**. The **Development Team** is part of a **Scrum Team** and is responsible for developing the product and intermediary results.

A **Product Owner** can be a **Product Owner Client** or a **Product Owner Project Stakeholder**. The former occurs when the **Client** himself is a **Scrum Team Member** and plays the **Product Owner Role**. The latter occurs when another person represents the **Client's** interests by playing the **Product Owner Role** in the **Scrum Project**.

3.2.3 Scrum Stakeholders Participation Subontology

The *Scrum Stakeholders Participation subontology* accounts for the participation of stakeholders in a Scrum project. It focuses on the stakeholders involved in processes and activities of a Scrum project, addressing questions CQ17 to CQ22 below:

- CQ17. Which stakeholders are in charge of the ceremonies of a Scrum project?
- CQ18. Which stakeholders participate in the ceremonies of a Scrum project?
- CQ19. Which stakeholders are in charge of development tasks of a Scrum project?
- CQ20. Which stakeholders participate in development tasks of a Scrum project?
- CQ21. Which stakeholders are in charge of processes of a Scrum project?
- CQ22. Which stakeholders participate in processes of a Scrum project?

This information helps analyze the participation of stakeholders in a project and verify team members' performance (e.g., by identifying the development tasks a team member performed and the task's duration) (SCHWABER; BEEDLE, 2002; SCHWABER; SUTHERLAND, 2011; RUBIN, 2012). The subontology also provides knowledge about the roles involved in the processes and activities of a Scrum project. Figure 28 shows the subontology diagram.

To address the involvement of stakeholders in processes and activities of a Scrum project, the subontology focuses on the *is in charge of* and *participates in* relations defined

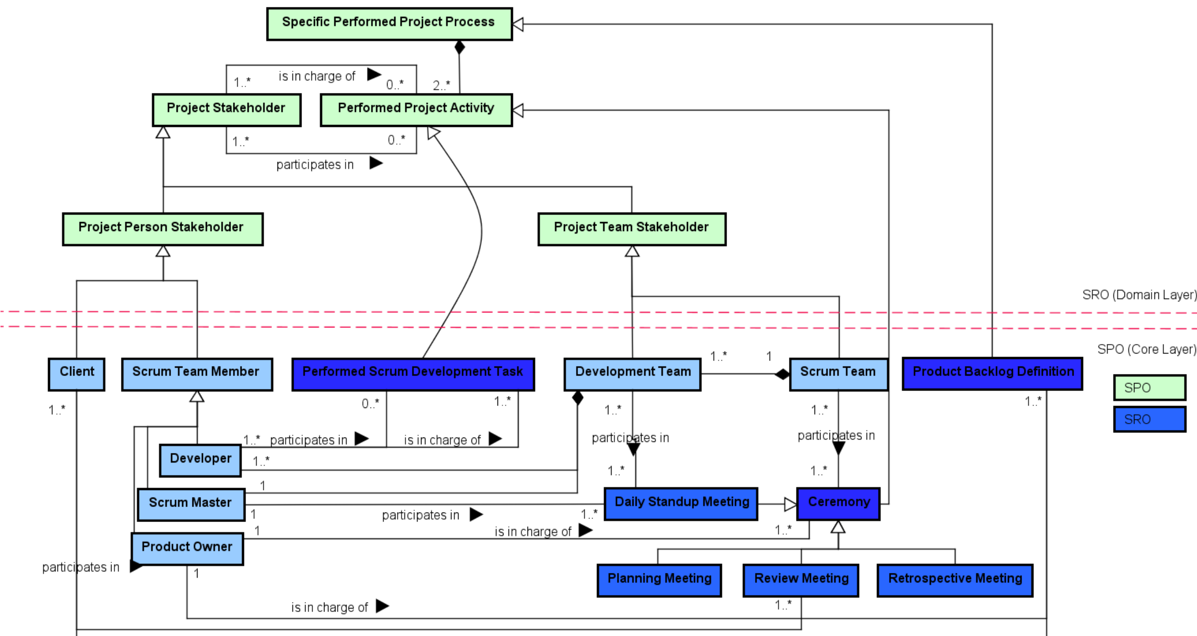


Figure 28 – Scrum Stakeholders Participation Subontology.

between stakeholders (*Project Stakeholders*) and activities (*Performed Project Activity*) in SPO. As discussed in Section 2.2.2, the former states that one or more stakeholders are responsible for performing one or more activities. The latter establishes that stakeholders contribute to the execution of one or more activities. The equivalent relationships between stakeholders and process are derived from the relationships between stakeholders and performed activities and the whole-part relationship between activities and process.

A **Product Owner** is responsible for the **Product Backlog Definition**, **Planning Meeting**, **Review Meeting**, and **Retrospective Meeting**, while a **Scrum Master** and a **Development Team** participate in these process and ceremonies. The **Client** also participates in the **Product Backlog Definition**.

A **Scrum Master** is responsible for the **Daily Stand up Meeting**, in which **Developers** participate. Finally, **Developers** are responsible for or participate in **Performed Scrum Development Tasks**.

The model depicted in Figure 28 represents the stakeholders (**Developer**, **Scrum Master**, and **Product Owner**) involved in Scrum activities and subprocess. The respective *Organizational Roles* played by these stakeholders (**Developer Role**, **Scrum Master Role**, and **Product Owner Role**) can be seen in Figure 28 and represent the roles involved in Scrum activities and subprocess (e.g., the **Scrum Team Member** in charge of a **Performed Scrum Development Task** is a **Developer**, i.e., he or she plays the **Developer Role**).

3.2.4 Product and Sprint Backlog Subontology

The *Product and Sprint Backlog subontology* focuses on aspects related to the requirements established in a Scrum project. It aims to answer the following competency questions:

- CQ23. What user stories were defined in the product backlog of a Scrum project?
- CQ24. What is the priority of a user story in the product backlog of a Scrum project?
- CQ25. How is a user story broken down into others?
- CQ26. What acceptance criteria were established for a user story?
- CQ27. Which user stories were selected for a sprint backlog?
- CQ28. What development tasks were planned to materialize a user story?
- CQ29. What development tasks were performed to materialize a user story?
- CQ30. What development tasks were planned for a sprint?
- CQ31. What development tasks were performed in a sprint?

CQ23 to CQ25 regard the user stories defined in a Scrum project and recorded in its product backlog. This information is necessary to know the project scope and the priority to address its requirements (typically in the form of user stories). Moreover, it is possible to identify user stories decomposed into others to ease development tasks and project management (SCHWABER; BEEDLE, 2002). CQ26 provides information about criteria that must be considered to evaluate the results produced when materializing a user story. This information is important to quality assurance. CQ28 and CQ30 provide information about tasks planning, while QC29 and QC31 concern tasks execution. By answering these questions, it is possible to track planned and performed tasks and evaluate the adherence between them. CQ27 allows identifying the user stories to be materialized during a sprint. By relating this information with answers to CQ31 and CQ29 it is possible to verify whether the user stories selected to be addressed during a sprint were materialized in that sprint. Figure 29 shows the diagram of the Product and Sprint Backlog subontology.

A **Product Backlog** is created during the **Product Backlog Definition**. It is a *Document* that contains the requirements of the product to be developed in the **Scrum Project**. These requirements are described by means of **User Stories**. Therefore, a **User Story** is a *Requirement Artifact* that describes *Requirements* in a **Scrum Project**. For example, (US1) I, as a Traveler, want to pay my travel ticket. A **User Story** can be an **Atomic User Story**, when it is not decomposed in others (e.g., (US1.1) I, as a Traveler, want to pay my travel ticket with my credit card; (US1.2) I, as a Traveler, want to pay my travel ticket with bank slip), or an **Epic**, when it is composed of other **User Stories** (e.g., US1, which is composed of US1.1 and US1.2).

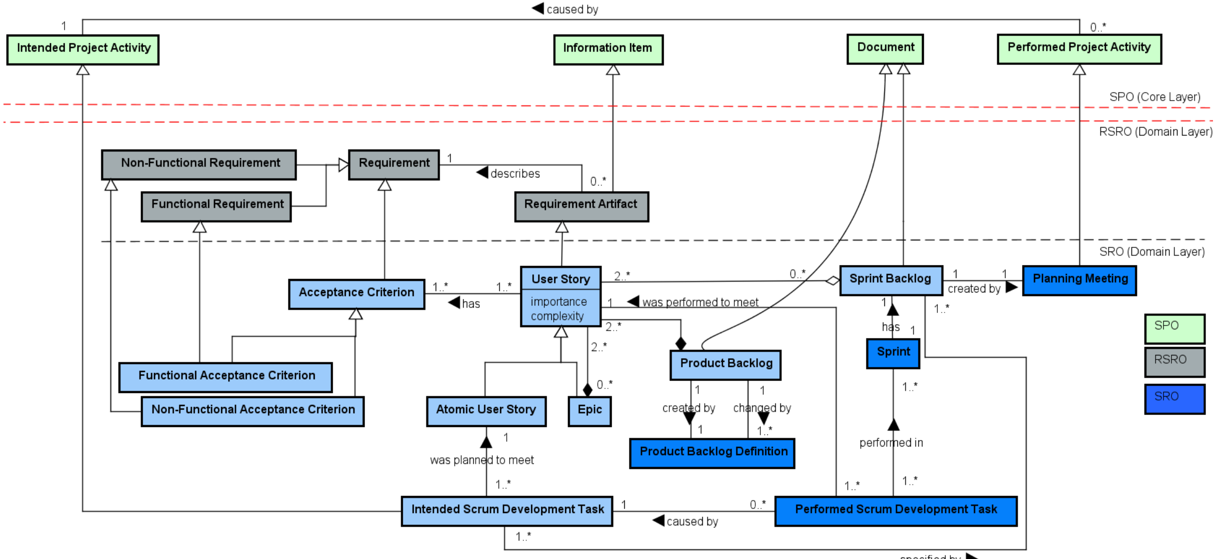


Figure 29 – Product and Sprint Backlog Subontology.

A **User Story** has two main properties (COHN, 2010): **Importance** and **Complexity**. **Importance** defines how valuable for the organization the **User Story** is. Usually, the **Product Owner** sets a number for it. The higher the number, the more valuable is a **User Story** for the organization. **Complexity** defines how difficult, or complex, it should be for the **Development Team** to implement the **User Story**. The higher the effort, the more difficult it is for the **User Story** to be materialized.

Each **User Story** has **Acceptance Criteria**, which are *Requirements* used to verify whether the **User Story** was developed correctly and meets the client's needs. An **Acceptance Criterion** can be a **Functional Acceptance Criterion** (i.e., a *Functional Requirement* used to verify whether the functionality addressed in the **User Story** was developed correctly) or a **Non-Functional Acceptance Criterion** (i.e., a *Non-Functional Requirement* establishing a quality criterion related to product characteristics or capabilities, such as usability and portability). (AC1) The credit card must be valid and (AC2) The payment authentication is done in less than 10ms are, respectively, examples of **Functional** and **Non-Functional Acceptance Criterion** related to US1.1.

During the **Planning Meeting** of a **Sprint**, the **User Stories** to be addressed in that **Sprint** are selected from the **Product Backlog**. For each selected **User Story**, **Intended Scrum Development Tasks** are planned. They describe the tasks needed to materialize each **User Story**. The selected **User Stories** and the respective **Intended Scrum Development Tasks** are recorded in the **Sprint Backlog**, a *Document* that describes the **Sprint planning**.

When the **Planning Meeting** is executed, **Intended Scrum Development Tasks** *causes* **Performed Scrum Development Tasks**, which are the tasks actually performed to materialize the **User Stories**. That is, the tasks planned to produce the **User Stories** lead to the execution of tasks with that purpose.

Intended Scrum Development Tasks defined in the **Sprint Backlog** but not executed in the respective **Sprint** (i.e., without a respective **Performed Scrum Development Task**) can be associated to the **Sprint Backlog** of next **Sprints**. Hence, an **Intended Scrum Development Task** may be related to several **Sprint Backlogs** and, consequently, to several **Sprints**.

3.2.5 Scrum Deliverables Subontology

The *Scrum Deliverables subontology* aims to answer the following competency questions:

- CQ32. Which types of deliverables are produced in a Scrum project?
- CQ33. What deliverables were produced in a Sprint?
- CQ34. Which deliverables were produced in a Scrum project?
- CQ35. Which user stories did a deliverable materialize?
- CQ36. Which deliverables were accepted in a Sprint?
- CQ37. Which development tasks produced accepted deliverables?

CQ32 to CQ35 provide information about deliverables produced during Scrum projects. This information associated to information from CQ27 to CQ31 allows verifying project performance and progress in terms of selected user stories, tasks performed to implement the selected user stories and deliverables that materialized the user stories (SCHWABER; BEEDLE, 2002; SCHWABER; KEN, 2013). Moreover, by answering CQ31, CQ33, CQ36 and CQ37 it is possible to evaluate work quality. For example, by relating information from CQ33 and CQ36 it is possible to verify whether all the deliverables produced in a sprint were accepted or whether there is a need for reworking to fix deliverables that were not accepted (SCHWABER; BEEDLE, 2002; SCHWABER; KEN, 2013). Furthermore, from CQ31 and CQ37 it is possible to identify how much work has been spent on producing deliverables that end up “not accepted”. All this information is useful to verify work quality and team productivity. The diagram of the Scrum Deliverables subontology is shown in Figure 30.

Performed Scrum Development Tasks are performed in one or more **Sprints** aiming to produce **Deliverables**, which are *Software Items* that materialize **User Stories** addressed in that **Sprint** (e.g., (i) a feature to search and select flight; (ii) a feature to pay the travel ticket using credit card).

A **Deliverable** is evaluated considering the **Acceptance Criteria** related to the **User Stories** it materializes. When the **Deliverable** is in conformance with the **Acceptance Criteria**, it is said an **Accepted Deliverable** and it means that it is “done”. Otherwise, it is a **Not Accepted Deliverable**. For instance, the feature to pay the travel ticket using credit card,

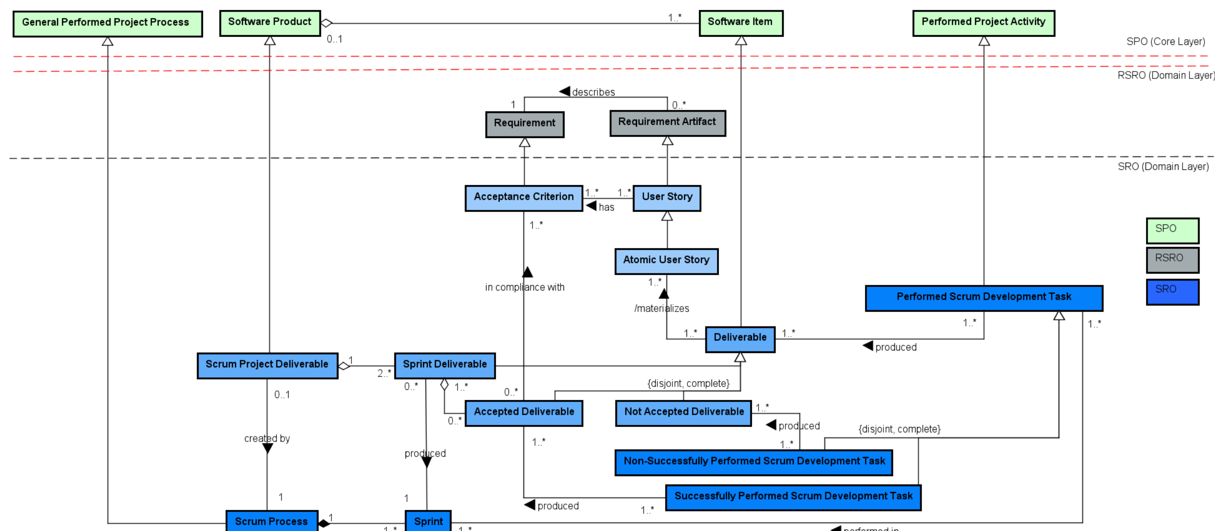


Figure 30 – Scrum Deliverables Subontology.

which materializes US1.1 (I, as a Traveler, want to pay my travel with my credit card) is an **Accepted Deliverable** if it is in conformance to both the **Acceptance Criteria** defined to US1.1: (AC1) The credit card must be valid and (AC2) The payment authentication is done in less than 10ms. Otherwise, it is a **Not Accepted Deliverable**.

A **Performed Scrum Development Task** that produced only **Accepted Deliverables** is said a **Successfully Performed Scrum Development Task**. On the other hand, a **Non-Successfully Performed Scrum Development Task** is a **Performed Scrum Development Task** that produced one or more **Not Accepted Deliverables**. **User Stories** related to **Not Accepted Deliverables** can return to the **Product Backlog** to be addressed again in next **Sprints**. For example, suppose that the feature to pay the travel ticket using a credit card, which materializes US1.1 (I, as a Traveler, want to pay my travel with my credit card), was produced by the **Performed Scrum Development Task** Build page to credit card payment in the **Sprint** S1. Moreover, consider that US1.1 was not in conformance to the **Acceptance Criterion** (AC2) The payment authentication is done in less than 10ms. That means that the feature "to pay the travel ticket using credit card" is a **Not Accepted Deliverable** and, thus, Build page to credit card payment is a **Non-Successfully Performed Scrum Development Task**. Since the **User Story** US1.1 was related to a **Not Accepted Deliverable**, after S1 is finished, US1.1 can return to the **Product Backlog** to be addressed in the **Sprint** S2 (or other) in order to be materialized by an **Accepted Deliverable**.

Accepted Deliverables produced in a **Sprint** are integrated forming a more complex and complete *Software Item* called **Sprint Deliverable**, which is the **Sprint** result delivered to the client. The set of **Sprint Deliverables** produced in the **Sprints** of a **Scrum Project** forms the **Scrum Project Deliverable**, which is the final deliverable of the **Scrum Process**. **Scrum Project Deliverable** is a *Software Product* and, as such, it is composed of (i) one or more *Software Items* (e.g., programs) working together for satisfying certain needs, and (ii)

other items to support the *Software Product* use or maintenance, such as documentation.

Some constraints not captured by the SRO conceptual models were defined by means of axioms. As an example, in the context of the Product and Sprint Backlog and Scrum Deliverable subontologies (Figures 29 and 30), **Scrum Development Tasks Performed** in a given **Sprint** must be related to a **User Story** from the **Sprint Backlog** of that **Sprint**. That is, if a **Performed Scrum Development Task** dt is performed in a **Sprint** s that has the **Sprint Backlog** sb and dt is to produce a **Deliverable** d, then there is a **User Story** us in the **Sprint Backlog** sb that is materialized by the **Deliverable** d. In a (many-sorted) first-order logic notation:

$$\forall\, dt : \text{PerformedScrumDevelopmentTask}, s : \text{Sprint}, sb : \text{SprintBacklog}, d : \text{Deliverable}$$
$$((\text{isPerformedIn}(dt, s) \wedge \text{has}(s, sb) \wedge \text{isToProduce}(dt, d)) \rightarrow$$
$$\exists us(\text{isPartOf}(us, sb) \wedge \text{materializes}(d, us)))$$

Another example, focusing on the *Product & Sprint Backlog subontology* (Figure 29), constrains that a task performed to meet a user story is caused by a task planned to do that. That is, if a **Performed Scrum Development Task** dt was performed to meet a **User Story** us from the **Sprint Backlog** sb, then dt was caused by an **Intended Scrum Development Task** idt specified in sb and planned to meet us:

$$\forall\, dt : \text{PerformedScrumDevelopmentTask}, us : \text{UserStory}, sb : \text{SprintBacklog},$$
$$idt : \text{IntendedScrum DevelopmentTask}$$
$$((\text{wasperformedtoMeet}(dt, us) \wedge \text{isPartOf}(us, sb) \rightarrow$$
$$\exists idt(\text{specifies}(sb, idt) \wedge \text{isPlannedToMeet}(idt, us) \wedge \text{causedBy}(dt, idt)))$$

3.2.6 Evaluation

For evaluating SRO, we performed Verification and Validation activities, as suggested in (FALBO, 2014). We used two approaches to ontology evaluation: assessment by humans and data-driven approach (BRANK; GROBELNIK; MLADENIC, 2005). First, we performed a verification activity by means of expert judgment, in which we checked whether the concepts, relations, and axioms defined in SRO are able to answer the competency questions. For each competency question, we identified the elements of SRO which together are able to address the question. Table 1 presents results produced during verification. In the second, we aimed to validate the ontology by assessing whether it is suitable for representing real-world situations. For that, we instantiated SRO using data extracted from a real project. Table 2 presents instantiations recorded during validation. The instances were extracted from a project developed in the software organization where we performed the study described in Section 4.2.

Table 1 – SRO Verification.

#	Competence Question	SRO Concepts, Properties, and Relationships
CQ01	Which processes and activities make up a Scrum process?	**Sprint** and **Product Backlog Definition** **Scrum Process** *composed of* **Sprint** **Scrum Process** *composed of* **Planning Meeting**
CQ02	In a Scrum project, on which other activities/processes did a certain activity/process depend?	**Sprint** *depends on* **Product Backlog Definition** **Performed Scrum Development Task** *depends on* **Planning Meeting** **Daily Standup Meeting** *depends on* **Performed Scrum Development Task** **Review Meeting** *depends on* **Daily Standup Meeting** **Retrospective Meeting** *depends on* **Review Meeting**
CQ03	How many sprints were performed in a Scrum project?	**Scrum Process** *performed in* **Scrum Project** **Scrum Process** *composed of* **Sprint**
CQ4	What ceremonies were performed in a sprint?	**Sprints** *composed of* **Planning Meeting** **Sprint** *composed of* **Daily Standup Meeting** **Sprint** *composed of* **Review Meeting** **Sprint** *composed of* **Retrospective Meeting**
CQ05	What development tasks were performed in a sprint?	**Sprint** *composed of* **Performed Scrum Development Task**
CQ06	When did a Scrum project start?	**Performed Process**.*startDate* **Performed Project Process** subtype of **Performed Process** **Scrum Process** *subtype of* **Performed Project Process** **Scrum Process** *performed in* **Scrum Project**
CQ07	When did a Scrum project end?	**Performed Process**.*endDate* **Performed Project Process** subtype of **Performed Process** **Scrum Process** *subtype of* **Performed Project Process** **Scrum Process** *performed in* **Scrum Project**
CQ08	When did a Scrum process start?	**Performed Process**.*startDate* **Performed Project Process** subtype of **Performed Process** **Scrum Process** *subtype of* **Performed Project Process**
CQ09	When did a Scrum process end?	**Performed Process**.*endDate* **Performed Project Process** subtype of **Performed Process** **Scrum Process** *subtype of* **Performed Project Process**
CQ10	When did a Scrum project activity start?	**Performed Activity**.*startDate* **Performed Project Activity** subtype of **Performed Activity** **Ceremony** *subtype of* **Performed Project Activity** **Scrum Development Task** *subtype of* **Performed Project Activity**

Table 1 – *Continued from previous page*

#	Competence Question	SRO Concepts, Properties, and Relationships
CQ11	When did a Scrum project activity end?	**Performed Activity**.*endDate* **Performed Project Activity** subtype of **Performed Activity** **Ceremony** *subtype of* **Performed Project Activity** **Scrum Development Task** *subtype of* **Performed Project Activity**
CQ12	What roles are involved in a Scrum project?	**Scrum Role**, **Developer**, **Product Owner**, and **Client** **Scrum Role** *subtype of* **Organizational Role** **Developer** *subtype of* **Scrum Role** **Product Owner** *subtype of* **Scrum Role** **Scrum Master** *subtype of* **Scrum Role** **Client** *subtype of* **Scrum Role**
CQ13	What teams are involved in a Scrum project?	**Scrum Team** and **Development Team** **Scrum Team** *subtype of* **Project Team Stakeholder** **Development Team** *subtype of* **Project Team Stakeholder** **Scrum Team** *composed of* **Project Team Stakeholder**
CQ14	Which roles are involved in a team in a Scrum project?	**Development Team** *composed of* **Developer** **Development Team** *composed of* **Scrum Master** **Scrum Team** *composed of* **Development Team** **Scrum Team** *composed of* **Product Owner** **Scrum Team** *composed of* **Client**
CQ15	Who are the members of a team in a Scrum project?	**Developer**,**Scrum Master**, **Product Owner**,and **Client** *subtype of* **Scrum Team Member Product Owner Client** *subtype of* **Client** **Product Owner Stakeholder** *subtype of* **Product Owner**
CQ16	Which role is played by a team member in a Scrum project?	**Developer Role**, **Scrum Master Role**, and **Product Owner Role** *subtype of* **Scrum Role** **Developer Role** *is to play* **Developer** **Scrum Master Role** *is to play* **Scrum Master** **Product Owner Role** *is to play* **Product Owner**
CQ17	Which stakeholders are in charge of the ceremonies of a Scrum project?	**Scrum Master** *is in charge of* **Daily Standup Meeting** **Product Owner** *is in charge of* **Product Backlog Definition**, **Planning Meeting**, **Review Meeting**, and **Retrospective Meeting**
CQ18	Which stakeholders participate in the ceremonies of a Scrum project?	**Development Team** *participate in* **Daily Standup Meeting** **Scrum Team** *participate in* **Ceremony** **Client** *participate in* **Product Backlog Definition**
CQ19	Which stakeholders are in charge of development tasks of a Scrum project?	**Developer** *is in charge of* **Performed Scrum Development Task**

Table 1 – *Continued from previous page*

#	Competence Question	SRO Concepts, Properties, and Relationships
CQ20	Which stakeholders participate in of development tasks of a Scrum project?	**Developer** *participates in* **Performed Scrum Development Task**
CQ21	Which stakeholders are in charge of processes of a Scrum project?	**Product Owner** *is in charge of* **Product Backlog Definition**
CQ22	Which stakeholders participate in of processes of a Scrum project?	**Client** *participates in* **Performed Scrum Development Task**
CQ23	What user stories were defined in the product backlog of a Scrum project?	**Product Backlog** *composed of* **User Story** **Epic** *subtype of* **User Story** **Atomic User Story** *subtype of* **User Story**
CQ24	What is the priority of a user story in the product backlog of a Scrum project?	**User Story**.*Importance*
CQ25	How is a user story broken down into others?	**Epic** *composed of* **User Story**
CQ26	What acceptance criteria were established for a user story?	**User Story** *has* **Acceptance criterion** **Functional Acceptance criterion** *subtype of* **Acceptance criterion** **Non-Functional Acceptance criterion** *subtype of* **Acceptance criterion**
CQ27	Which user stories were selected to a sprint backlog?	**Sprint Backlog** *composed of* **User Story**
CQ28	What development tasks were planned to materialize a user story?	**Sprint Backlog** *composed of* **User Story** **Sprint Backlog** *specifies* **Intended Scrum Development Task** **Intended Scrum Development Task** *is planned to meet* **Atomic User Story**
CQ29	What development tasks were performed to materialize a user story?	**Sprint Backlog** *composed of* **User Story** **Sprint Backlog** *specifies* **Intended Scrum Development Task** **Intended Scrum Development Task** *is planned to meet* **Atomic User Story** **Performed Scrum Development Task** *caused by* **Intended Scrum Development Task**
CQ30	What development tasks were planned for a sprint?	**Sprint** *has* **Sprint Backlog**. **Sprint Backlog** *specifies* **Intended Scrum Development Task**.
CQ31	What development tasks were performed in a sprint?	**Performed Scrum Development Task** *performed in* **Sprint**.

Table 1 – *Continued from previous page*

#	Competence Question	SRO Concepts, Properties, and Relationships
CQ32	Which types of deliverables are produced in a Scrum project?	**Deliverable** *subtype of* **Software Item** **Sprint Deliverable** *subtype of* **Deliverable** **Accepted Deliverable** *subtype of* **Deliverable** **Not Accepted Deliverable** *subtype of* **Deliverable**
CQ33	What deliverables were produced in a sprint?	**Performed Scrum Development Task** *performed in* **Sprint** **Performed Scrum Development Task** *produced* **Deliverable** **Successfully Performed Scrum Development Task** *produced* **Accepted Deliverable** **Non-Successfully Performed Scrum Development Task** *produced* **Not Accepted Deliverable** **Sprint Deliverable** *composed of* **Accepted Deliverable** **Sprint** *produces* **Sprint Deliverable**
CQ34	Which deliverables were produced in a Scrum project?	**Sprint** *produces* **Sprint Deliverable** **Scrum Process** *composed of* **Sprint** **Scrum Process** *creates* **Scrum Project Deliverable** **Scrum Project Deliverable** *composed of* **Sprint Deliverable**
CQ35	Which user stories did a deliverable materialize?	**Deliverable** *materializes* **Atomic User Story**
CQ36	Which deliverables were accepted in a Sprint?	**Successfully Performed Scrum Development Task** *subtype of* **Performed Scrum Development Task** **Successfully Performed Scrum Development Task** *produced* **Accepted Deliverable**
CQ37	Which development tasks produced accepted deliverables?	**Successfully Performed Scrum Development Task** *produced* **Accepted Deliverable**

Table 2 – SRO Validation.

SRO Concepts	Instance
Scrum Project	ESPM Project.
Scrum Process	Scrum process defined to the ESPM Project, comprising Product Backlog Definition, Sprints, Ceremonies, and Performed Scrum Development Tasks.
Product Backlog Definition	Process performed at the beginning of the project (from Jan 2, 2018 to Jan 11, 2018) to define the Product Backlog.
Sprint	Sprint S40, performed from Oct 21, 2019 to Nov 02, 2019.
Ceremony/Planning Meeting	Planning meeting performed at the first day of S40, on Oct 21, 2019.
Ceremony/Daily Standup Meeting	First daily meeting performed in S40, on Oct 22, 2019.
Ceremony/Review Meeting	Review meeting performed in S40, on Nov 02, 2019..

Table 2 – *Continued from previous page*

SRO Concepts	**instance**
Ceremony/Retrospective Meeting	Retrospective meeting performed in S40, on Nov 02, 2019..
Scrum Team Member/Developer	B. S. and T. M.[2]
Scrum Team Member/Scrum Master	M. E.
Scrum Team Member/Product Owner/Product Owner Stakeholder	R. S.
Scrum Team Member/Client	R. S.
Scrum Team	The team composed of the Scrum Team Members cited above.
Developer Membership	Allocation of B. S. to play the Developer role in the Scrum team.
Scrum Master Membership	Allocation of the M. E. to play the Scrum Master role in the Scrum team.
Product Owner Membership	Allocation of the R. S. to play the Product Owner role in the Scrum team.
User Story/Epic	US65: “I, as a public servant, I want to visualize my payslips”
Atomic User Story	US65.1: “I, as a public servant, I want to visualize my payslips in an application to smartphone”. US35.2: “I, as a public servant, I want to visualize my payslip in a web browser”.
Product Backlog	Product backlog containing the user stories defined to the ESPM Project. It contains US65, US65.1, US65.2 among others.
Sprint Backlog	Sprint backlog created during the planning meeting of sprint S40. It contains US65, US65.1, US65.2, among others
Acceptance Criterion/ Functional Acceptance Criterion	AC1 (related to US65.1): The payslip to be shown is determined by the month and year informed by the public servant.
Acceptance Criterion/ Non-Functional Acceptance Criterion	AC2 (related to US65.1): A payslip should appear in 500 milliseconds.
Intended Scrum Development Task	**Task** “Create payslip report functionality in mobile app” planned to implement the user story US65.1 during S40
Performed Scrum Development Task/Successfully Performed Scrum Development Task	**Task** “Create payslip report functionality in the mobile app”, performed during S40 to implement the user story US65.1 and that produced the accepted deliverable below
Deliverable/Accepted Deliverable	Functionality “Payslip report” in the mobile app, resulting from the implementation of US65.1.
Performed Scrum Development Task/Non-Successfully Performed Scrum Development Task	**Task** “Create payslip report in the web application”, performed during S40 to implement the user story US65.2 and that produced the not-accepted deliverable below.
Deliverable/Not Accepted Deliverable	Functionality “Payslip report” in the web application, resulting from the implementation of US65.2, and that was not accepted due to failure.

[2] The names of Individual project participants were omitted in favor of initials of privacy. Their names were replaced by characters from TV Show called *How I Met your mother.*

Table 2 – *Continued from previous page*

SRO Concepts	**instance**
Sprint Deliverable	The set of accepted functionalities developed during the S40 (which includes "Paylist report" in the mobile app, among others), incorporated to the software version delivered in S39
Scrum Project Deliverable	SmartCity, the software product resulting from ESPM project.

3.3 Continuous Integration Reference Ontology (CIRO)

The *Continuous Integration Reference Ontology* (CIRO) consolidates reference literature about Continuous Integration (CI), using as main sources (FOWLER, 2006; DUVALL; MATYAS; GLOVER, 2007; HUMBLE; FARLEY, 2010; STÅHL; BOSCH, 2014; SHAHIN; BABAR; ZHU, 2017; STARON et al., 2018). CIRO is organized into four subontologies, following the architecture depicted in Figure 31:

- The *Continuous Integration Process subontology*: which presents an overview of the CI process, identifying the main events that occur in this context, including continuous build, continuous test, and continuous inspection, each of which is covered in detail in a corresponding subontology.
- The *Continuous Build Process subontology*: which addresses the main activities, roles, software artifacts, and applications used to implement a continuous and automatic building process in CI context. In addition, this subontology presents the roles that a software artifact performs when it is built.
- The *Continuous Test Process subontology*: which introduces main activities, applications, and software artifacts used to implement a continuous and automatic testing process in the CI context. In addition, this subontology describes the roles that a software artifact under testing can perform in the CI context.
- The *Continuous Inspection Process subontology*: aims at representing main activities, artifacts, roles, and applications involved in a continuous and automatic inspection process in the CI context.

The next section presents an extension of CMPO that provides conceptualizations aligned with the Git philosophy necessary to develop CIRO.

3.3.1 Extension of the Configuration Management Process Ontology (CMPO)

The original version of CMPO (CALHAU; FALBO, 2012) is suitable for configuration management processes aligned to the Subversion (a.k.a. SVN) philosophy. Thus, we need to

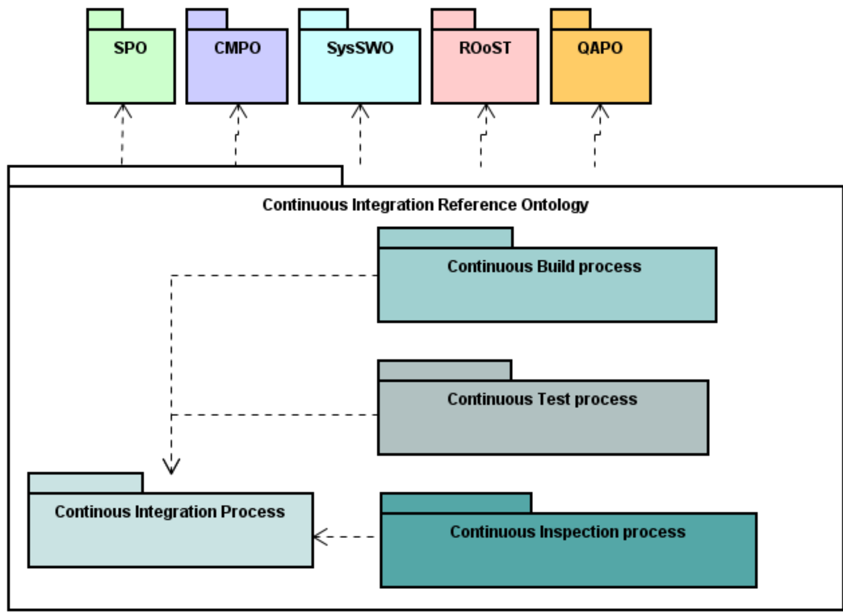


Figure 31 – CIRO's architecture.

make some changes to turn CMPO suitable for the CSE context. The changes made aimed to address configuration management processes aligned with the Git philosophy (CONSERVANCY, 2023). The changes aimed to make CMPO able to properly represent check in and check out situations present in the way software is developed nowadays. These changes were necessary mainly to develop CIRO. Figure 32 presents a CMPO fragment focusing on *Checkout*.

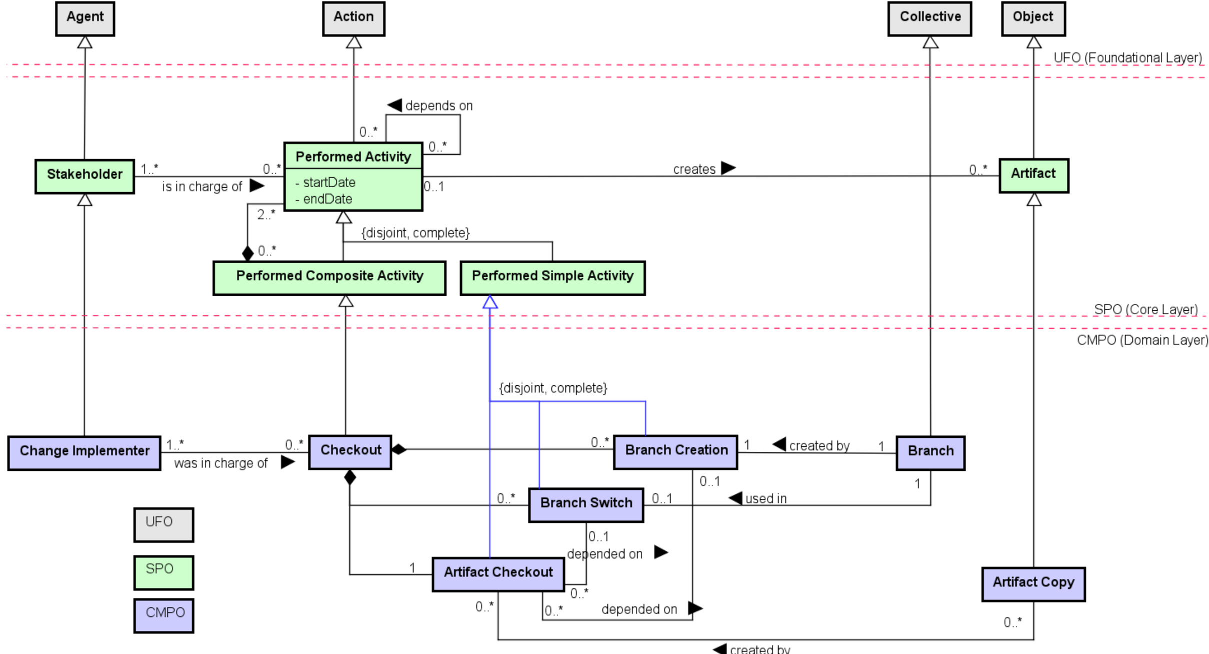


Figure 32 – CMPO fragment focusing on Checkout.

Checkout activity is composed of the following *Performed Simple Activities*: *Branch Creation*, *Branch Switch*, and *Artifact Checkout*. *Branch Creation* is an activity for creating a

Branch in a Source Repository. A Branch is a Collective of the Artifacts in a Source Repository. Branch Switch is an activity for switching between branches in a source repository while Artifact Checkout is an activity to create or update a Branch with Artifact Copy. Figure 33 presents a CMPO fragment focusing on Checkin.

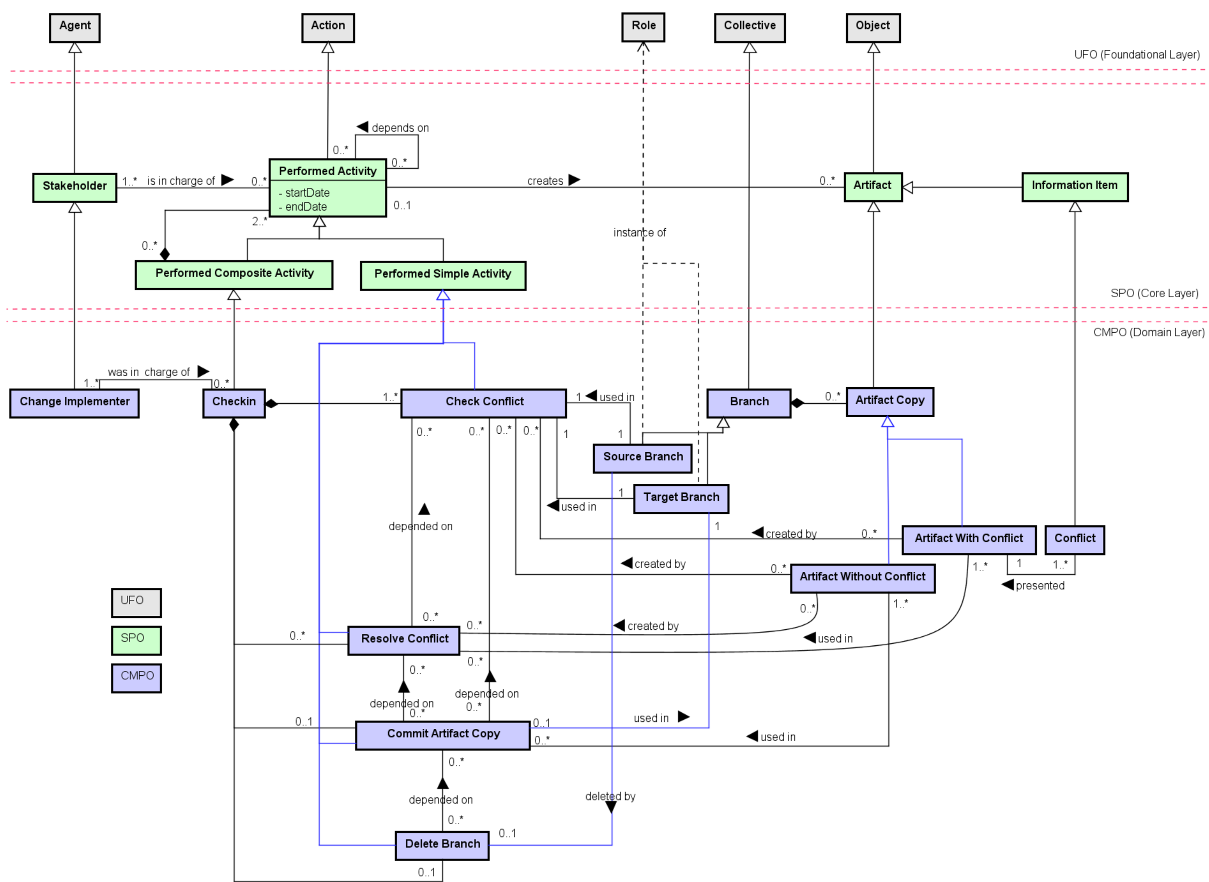


Figure 33 – CMPO fragment focusing on Checkin.

Checkin activity is composed of the following Performed Simple Activities: Check Conflict, Resolve Conflict, Commit Artifact Copy, and Delete Branch. Check Conflict is an activity that verifies if there are Conflicts between an Artifact Copy in a Source Branch with a version of it in a Target Branch. A Source Branch is a Role that a Branch assumes when there is a new version or a new Artifacts Copy is created and desire to save such Artifacts Copy in other Branch (Target Branch), after a Checkin activity. A Conflict represents a difference of content in a same region of an Artifact Copy. An Artifact Copy with Conflict is created when a Conflict is identified while an Artifact Copy without Conflict when a Conflict was not identified.

Finally, Resolve Conflict is an activity that allows a Change Implementer to fix a Conflict in an Artifact Copy with Conflict. An Artifact Copy without Conflict is created when all Conflicts are resolved. A Commit Artifact Copy is an activity that sends an Artifact Copy without Conflict to a Target Branch. Finally, Delete Branch is an activity that removes a Source Branch in a Source Repository.

3.3.2 Continuous Integration Process Subontology

The *Continuous Integration Process* subontology aims to answer the following competency questions:

- CQ01. Which processes and activities did make up a CI process?
- CQ02. In the CI process, on which other activities/processes did a certain activity/process depend?
- CQ03. When did a CI process start?
- CQ04. When did a CI process end?
- CQ05. Which artifacts participated in the CI process?
- CQ06. Which stakeholders participated in the CI process?
- CQ07. What type of event triggered the CI process?

CQ01 and CQ02 regard subprocesses and activities involved in a Continuous Integration (CI) process. They aim to provide knowledge about the CI process structure and reveal the order required of its subprocesses and activities. CQ03 to CQ04 refer to temporal aspects and aim to provide information about processes and activities duration, contributing to analyzing the process performance (SHAHIN; BABAR; ZHU, 2017). CQ05 addresses the participants (agents or objects) in a CI process, while CQ06 is concerned with roles played by agents that participated in the continuous integration process. Finally, CQ07 addresses the different types of CI process according to how they were triggered. Figure 34 shows the *Continuous Integration Process subontology* conceptual model.

A **Continuous Integration Process** is an automated *Specific Performed Project Composite Process* with the purpose of verifying whether a new software artifact can be integrated without bringing any problems to source code that is already approved, without human intervention, i.e., a **Continuous Integration Process** is an automated process that performs tests in the project's software artifacts to identify a building problem, a non-compliance with project's requirements, quality problem, and communicate the stakeholders about success or failed in any subprocess in an instance of **Continuous Integration Process** (DUVALL; MATYAS; GLOVER, 2007; SHAHIN; BABAR; ZHU, 2017).

As can be observed in Figure 34, a **Continuous Integration Process** is composed of at least (i) a **Continuous Build Process**, an automated *Specific Performed Project Process*, which aims at building a new version of the software to be tested (e.g., a software project's version 1.0.1 was created to be tested, after a commit of code that implements FR01), (ii) a **Continuous Test Process** is an automated *Test Process* that verifies whether the new version of the software is in

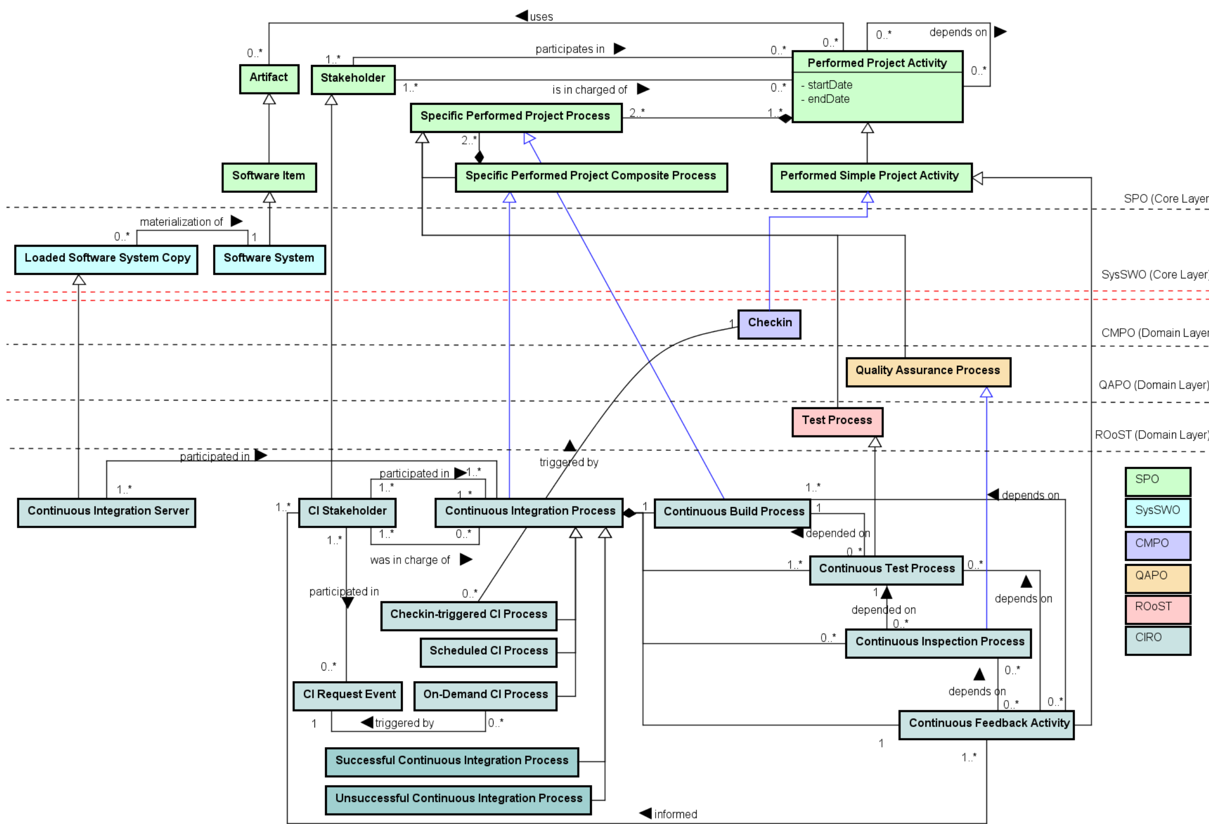


Figure 34 – Continuous Integration Process Subontology.

conformance with the software requirements applying automated tests (e.g., an automated unit test is performed to verify FR01 continues to be met, even after inclusion in the new software artifact.), and (iii) a **Continuous Feedback Activity**, an automated *Performed Simple Project Activity* that provides to a **CI Stakeholder** information about the status of a CI process (e.g., send information if a new software artifact broke a build or it failed to pass an automated unit test). A **CI Stakeholder** is a *Stakeholder* (e.g., a developer and tester) interested in information (e.g., success or failure of a software artifact during a testing process) about a **Continuous Integration Process**.

In addition to the aforementioned processes and activities, a **Continuous Inspection Process** could be part of a **Continuous Integration Process** to verify the quality of a software artifact (DUVALL; MATYAS; GLOVER, 2007; SHAHIN; BABAR; ZHU, 2017). A **Continuous Inspection Process** is an automated *Quality Assurance Process* that aims at guaranteeing the software artifacts conform to certain software engineering quality criteria (e.g., a function cannot have more than 300 lines of code or the project software artifacts cannot have more than 10% duplicate code (MARTIN, 2009)).

Each process and activity presented in a **Continuous Integration Process** is performed inside of a **Continuous Integration Server**. A **Continuous Integration Server** is a *Loaded Software System Copy* (e.g., a copy of GitLab[3] loaded in a computer) that provides *Software*

[3] https://gitlab.com/

artifacts (e.g., libraries and programs) that participates in the CI process to enable executing the process in an automatic way, without human intervention.

There are three different types of **Continuous Integration Processes** according to how they are triggered. A **Check-in-Triggered Continuous Integration Process** is a **CI Process** that is triggered by a *Check-in*, i.e., when a new *Software artifact* is checked-in to a *Source Repository*, the CI process is started. A **Scheduled Continuous Integration Process** occurs when a specific date or time is reached (e.g., every day at 9 p.m.). Last, an **On-Demand Continuous Integration Process** is the one triggered by a *CI Request Event*, which is an *Event* that occurs when a **CI Stakeholder** executes a command in a **Continuous Integration Server** to start the **Continuous Integration Process**. For example, when a developer manually starts a CI process by clicking a button on GitLab's user interface.

Finally, a **Continuous Integration Process** is a **Successful Continuous Integration Process** when a **Candidate Code** is integrated, without problems. On the other hand, a **Unsuccessful Continuous Integration Process** is a **Continuous Integration Process** that has not integrated a **Candidate Code**, due to a problem in the **Continuous Integration Process**'s processes or activity.

3.3.3 Continuous Build Process Subontology

The *Continuous Build Process* subontology aims to answer the following competency questions:

- CQ08. Which activities make up a continuous build process?
- CQ09. Which resources were used to build the software artifacts during the continuous build process?
- CQ10. Which artifacts are created during the continuous build process?

Considering that this subontology represents the conceptualization of a process, the questions expected to be answered are similar to the ones defined in the CI Process Subontology and address aspects related to the process structure (i.e., the subprocess and activities that make up the process), when the process started and when it ended, the involved artifacts and stakeholders. We did not include any questions about the stakeholders involved in this particular process because they are the same represented in the CI Process Subontology. Figure 35 shows the packages of the *Continuous Build Process Subontology* modularization.

The package **CI Building Process** is dedicated to describing a building process while the **CI Building Environment** shows the concepts that are present in a building environment, in a CI context. Figure 36 shows the *Continuous Build Environment* conceptual model.

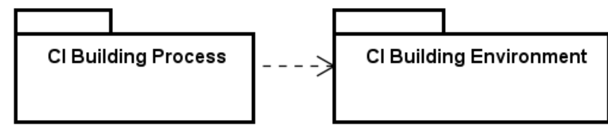


Figure 35 – Continuous Build Process Subontology Modularization.

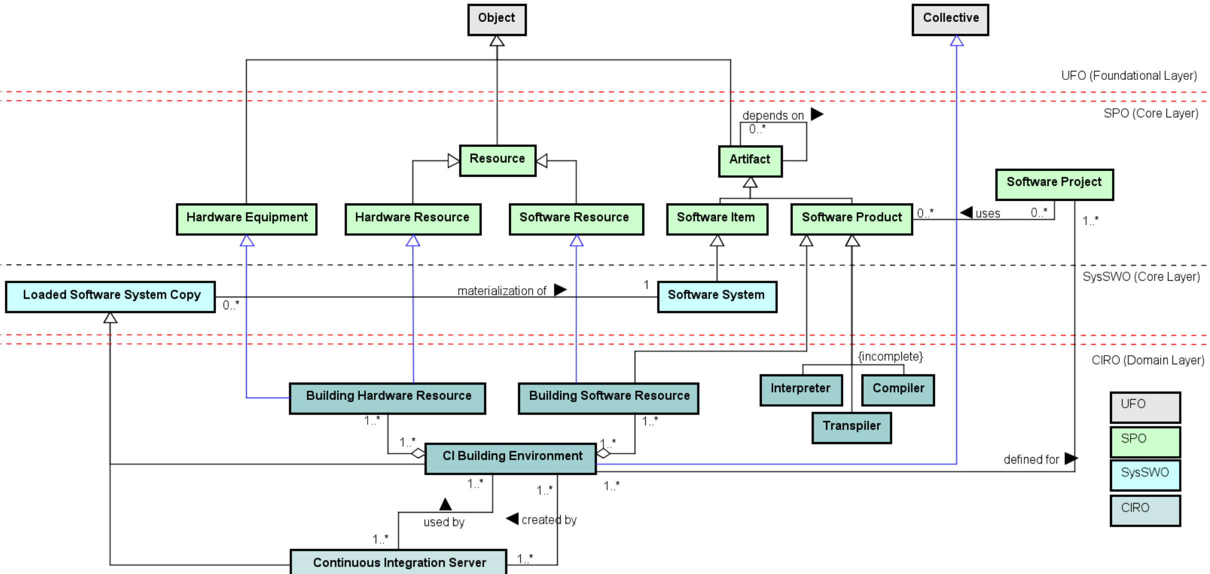


Figure 36 – Continuous Build Environment model from CI Building Environment package.

CI Building Environment is a _Loaded Software System Copy_ (e.g., a copy of build environment in a copy of the GitLab loaded in a computer) that contains **Building Software Resources** (e.g., Operational System (Linux or Windows), Compiler, Transpiler, Interpreter, or a library to build or load) and **Building Hardware Resources** (e.g., a physical machine where the **Building Software Resources** are installed and loaded) to support the building and loading activities of a **Continuous Build Process**.

Interpreters, **Compilers**, and **Transpilers** are _Software Products_ used to *"execute instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program"* (WIKIPEDIA, 2023c), *"translate a computer code written in one programming language (the source language) into another language (the target language)"* (WIKIPEDIA, 2023a), and *"translate a source code of a program written in a programming language as its input and produces an equivalent source code in the same or a different programming language"* (WIKIPEDIA, 2023f), respectively.

Building Software Resources and **Building Hardware Resources** are, respectively, _Software Products_ and _Hardware Equipments_ that comprise a **CI Building Environment**. A **Continuous Integration Server** uses a **CI Building Environment** to support a **Continuous Build Process**'s activities, as presented in Figure 37.

A **Continuous Build Process** is a _Specific Performed Project Process_ with participation of a **Continuous Integration Server**. It is composed of the following _Performed Project Ac-_

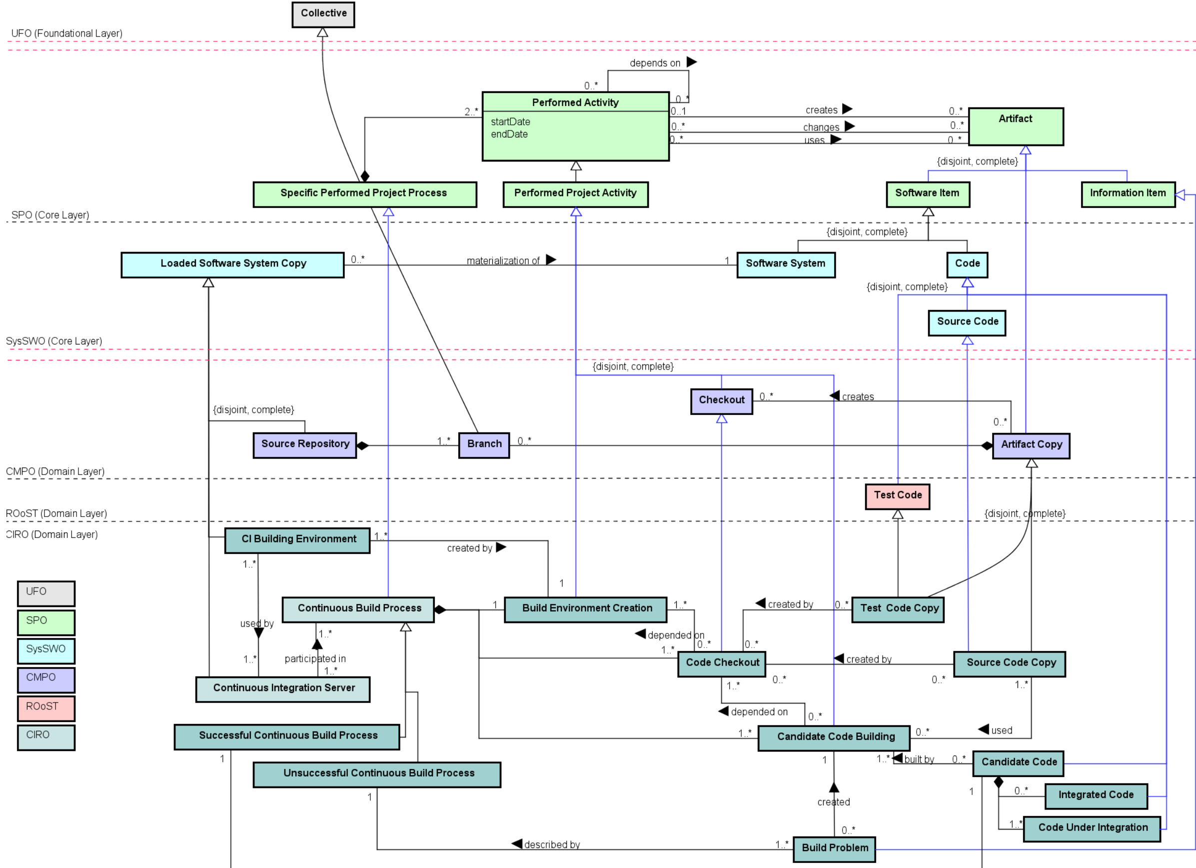


Figure 37 – Continuous Build Process model from Continuous Build Process package.

tivities: **Build Environment Creation**, **Code Checkout**, and **Candidate Code Building**. Dependency relations (*depended on* relation between *Performed Project Activities*) establish the order in which these activities have occurred. **Build Environment Creation** aims to create a **CI Build Environment** to support the activities **Code Checkout** and **Candidate Code Building**.

Code Checkout aims to create a **Source Code Copy** and **Test Code Copy** inside of a **CI Building Environment**. A **Source Code Copy** and **Test Code Copy** are, respectively, a copy of a *Source Code* (e.g, code libraries, source code under development of a project, SQL script to create the database's tables) and a *Test Code* (e.g., a code that implements a unit test or a System Test) that are in a *Source Repository* (e.g., a GitLab or GitHub). For example, an instance of GitLab[4] gl1 (**Continuous Integration Server**) created a copy of a version of some Java code, unit test code, database scripts, and other software artifacts (**Source Code** and **Test Code Copy**), from the main branch (*Branch*) present in an "Immigrant" source repository (*Source Repository*), inside of a virtual build machine vb1 (**CI Build Environment**) in gl1. **Candidate Code Building** aims to build a **Candidate Code** in a **CI Building Environment**, i.e., a **Candidate Code Building** uses the **Building Software Resources** (e.g., code libraries,

4 GitLab is composed of the components that implement Continuous Integration and Source Repository's requirements

compiler, transpiler, compiler, and interpreter) and **Building Hardware Resources** (e.g., a virtual machine with Linux or Windows), in a **CI Building Environment**, and the *Source Code Copies* to build a **Candidate Code** or to describe a **Build problem**.

Candidate Code is a collection of **Source Code Copy** (*Code*) that contains: (i) one or more items of **Code Under Integration**, which are either newly created *Code* items or altered *Code* items (e.g., a new version of a database script) that a **CI Stakeholder** desires to integrate into a *Source Repository*, and (ii) none, one or more **Integrated Code** items (e.g., code libraries, source code, database script, and HTML), which are **Code** items that were integrated into a *Source Repository* in a **Continuous Integration Process** performed in the past (DUVALL; MATYAS; GLOVER, 2007; SHAHIN; BABAR; ZHU, 2017).

A **Build Problem** is an *Information Item* about problems that occurred in a **Candidate Code Building** (e.g., referring to a code library that is missing making it impossible to compile the project).

Finally, a **Continuous Build Process** is a **Successful Continuous Build Process** when a **Candidate Code Building** builds a **Candidate Code**, without problems. On the other hand, a **Unsuccessful Continuous Build Process** is a **Continuous Build Process** that has not built a **Candidate Code**, due to a problem in **Candidate Code Building**.

3.3.4 Continuous Test Process Subontology

The *Continuous Test Process* subontology aims to answer the following competency questions that, as previously discussed, are similar to the ones defined to the CI Process Subontology:

- CQ11. Which processes and activities make up a Continuous Test process?
- CQ12. What automatic tests were performed?

Figure 38 shows the *Continuous Test Process subontology* conceptual model.

A **Continuous Test Process** is a *Test Process* with participation of a **Continuous Integration Server**. It is composed of the following automated *Performed Project Activities*: **CI Testing Environment Creation** and **Automated Testing**.

CI Testing Environment Creation aims to create a **CI Testing Environment** (*Testing Environment* and a *Loaded Software System Copy*) in a **Continuous Integration Server** (e.g., copy of a testing environment loaded inside of a copy of GitLab loaded in a computer) to support the activities of a **Continuous Test Process**. A **CI Testing Environment** contains *Test Software Resources* (e.g., Unit Test libraries and Test Management tools) and *Test Hardware Resources* (e.g., a computer to perform the unit tests).

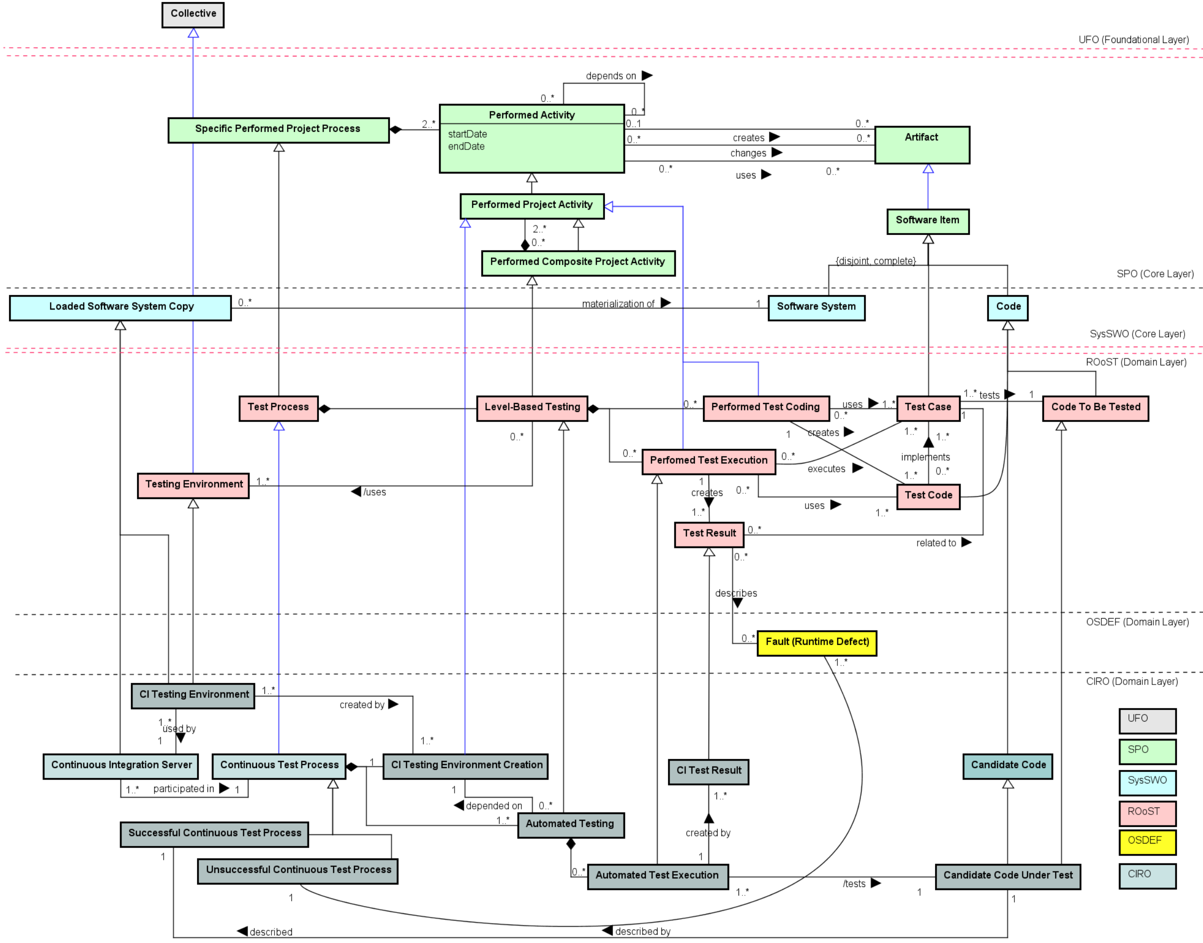


Figure 38 – Continuous Test Process Subontology.

Automated Testing is an automated *Level-Based Testing Activity* that performs tests automatically, using the *Test Software Resources* and *Test Hardware Resources* of a **CI Test Environment**. It is composed of **Automated Test Executions**, which are *Performed Test Execution* activities that automatically execute *Test Cases* by running their *Test Code* and producing **CI Test Results** (*Test Result*) that describes a result of the executed tests. For example, the *Test Case* described as TC01: Pay a trip with a credit card was executed by an **Automated Test Execution** tc01 that executed a *Test Code* (e.g., a Java code that implemented a Unit Test) unittest-tc01 in a **Candidate Code To Be Tested** cctbt, inside of a **CI Testing Environment** cite, producing a *CI Test Result* citr.

An **Automated Test Execution** tests a **Candidate Code To Be Tested**. A **Candidate Code To Be Tested** is a **Candidate Code** that plays the role *Code to Be Tested* in a **Continuous Test Process**. This way, a **Candidate Code under Test** is composed of an **Integrated Code** (e.g., the Java code tested before) and **Code under Integration** (e.g., a new version of a database script) that were tested by a *Test Code* (e.g., an implementation of a unit test to test the new version of the database script).

A **CI Test Result** is a *Test Result* that describes what was observed by applying a *Test*

Case in a **Continuous Test Process**. It may identify *Faults (Runtime Defects)* (e.g., that the function that calculates the income tax failed, with the new version of database script) that were detected when a *Test Code* was performed under a **Candidate Code To Be Tested** or serve as evidence of success when no *Faults (Runtime Defects)* are observed.

Finally, a **Continuous Test Process** is said a **Successful Continuous Test Process** when an **Automated Test Execution** produces a **CI Test Result** without *Faults (Runtime Defects)*, i.e., when the **Candidate Code To Be Tested** does not have any *Fault (Runtime Defect)* after all *Test Cases* are applied. On the other hand, a **Continuous Test Process** is a **Unsuccessful Continuous Test Process** when the **Candidate Code Under Test** failed, a *Fault (Runtime Defect)* was identified, in a *Test Case*.

3.3.5 Continuous Inspection Process Subontology

The *Continuous Inspection Process* subontology aims to answer the competency questions presented below. As previously discussed, they are similar to the ones defined to the CI Process Subontology:

- CQ13. Which processes and activities make up a Continuous Inspection process?
- CQ14. What was the project artifact in (non)conformance with the quality requirements of a project?

Figure 39 shows the packages of the *Continuous Inspection Process Subontology* modularization.

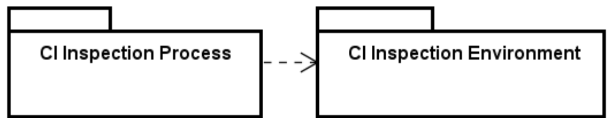


Figure 39 – Continuous Inspection Process Subontology Modularization.

The package **CI Inspection Process** is dedicated to describing an inspection process while the **CI Inspection Environment** shows the concepts that are present in an inspection environment, in a CI context. Figure 40 shows the *Continuous Inspection Environment* conceptual model.

CI Inspection Environment is a *Loaded Software System Copy* (e.g., a copy of environment with inspection software loaded inside in a computer) that contains **Inspection Software Resources** (e.g., a static code analysis tool) and **Inspection Hardware Resources** (e.g., a physical machine where some inspection software is installed and loaded) to support the inspection activities of a **Continuous Inspection Process**. **Static Code Analysis Tool** is a *Software Product* used to *"flag programming errors, bugs, stylistic errors, and suspicious constructs"* (WIKIPEDIA, 2023d).

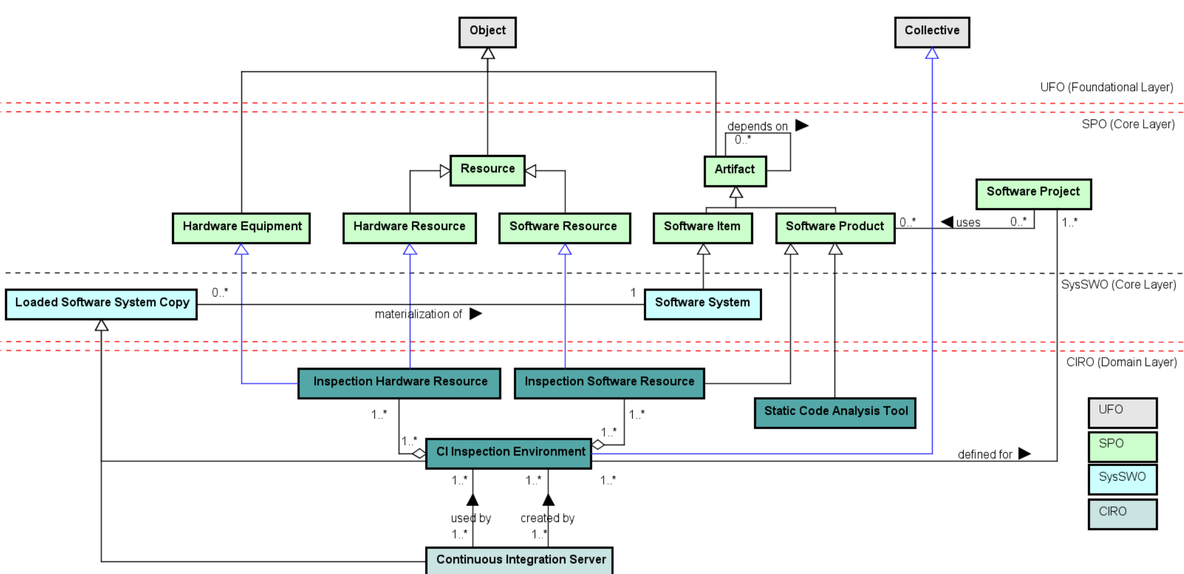


Figure 40 – Continuous Inspection Environment model from CI Inspection Environment package.

Inspection Software Resource and **Inspection Hardware Resource** are, respectively, *Software Products* (e.g., a static code analysis tool) and *Hardware Equipments* (e.g., a machine that runs a static code analysis tool). A **Inspection Software Resource** is used in a **Continuous Inspection Process**' activity via (*participates in*) a **Continuous Integration Server**, in a CI context. Figure 41 shows the *Continuous Inspection Process subontology* conceptual model.

A **Continuous Inspection Process** is a *Quality Assurance Process* with participation of a **Continuous Integration Server**. It is composed of the following *Performed Project Activities*: **Inspection Environment Creation** and **Automated Adherence Inspection**. **Inspection Environment Creation** aims to create a **CI Inspection Environment** in a **Continuous Integration Server** to support the **Automated Adherence Inspection**.

An **Automated Adherence Inspection** is an automated *Adherence Evaluation* activity that inspects the adherence of the **Candidate Code Under Inspection** executing **Automated Artifact Inspection** activities. An **Automated Artifact Inspection** is an automated *Artifact Evaluation* activity that uses **Quality Assurance Criterion Code** to inspect some *Quality Assurance Criteria* (e.g., QAC01: a function can have a maximum of 100 lines of code) in each *Software Artifact* in a **Candidate Code Under Inspection**. **Quality Assurance Criterion Code** a *Code* (e.g., a code in Java that implements QAC01) that materializes a *Quality Assurance Criterion*.

Automated Adherence Inspection produces a **CI Evaluation Report**, after executing all **Automated Artifact Inspection** activities. The **CI Evaluation Report** is an *Evaluation Report* that describes the inspection results and identified issues of a **Candidate Code Under Evaluation**. **Candidate Code Under Inspection** is a **Candidate Code** that each *Software*

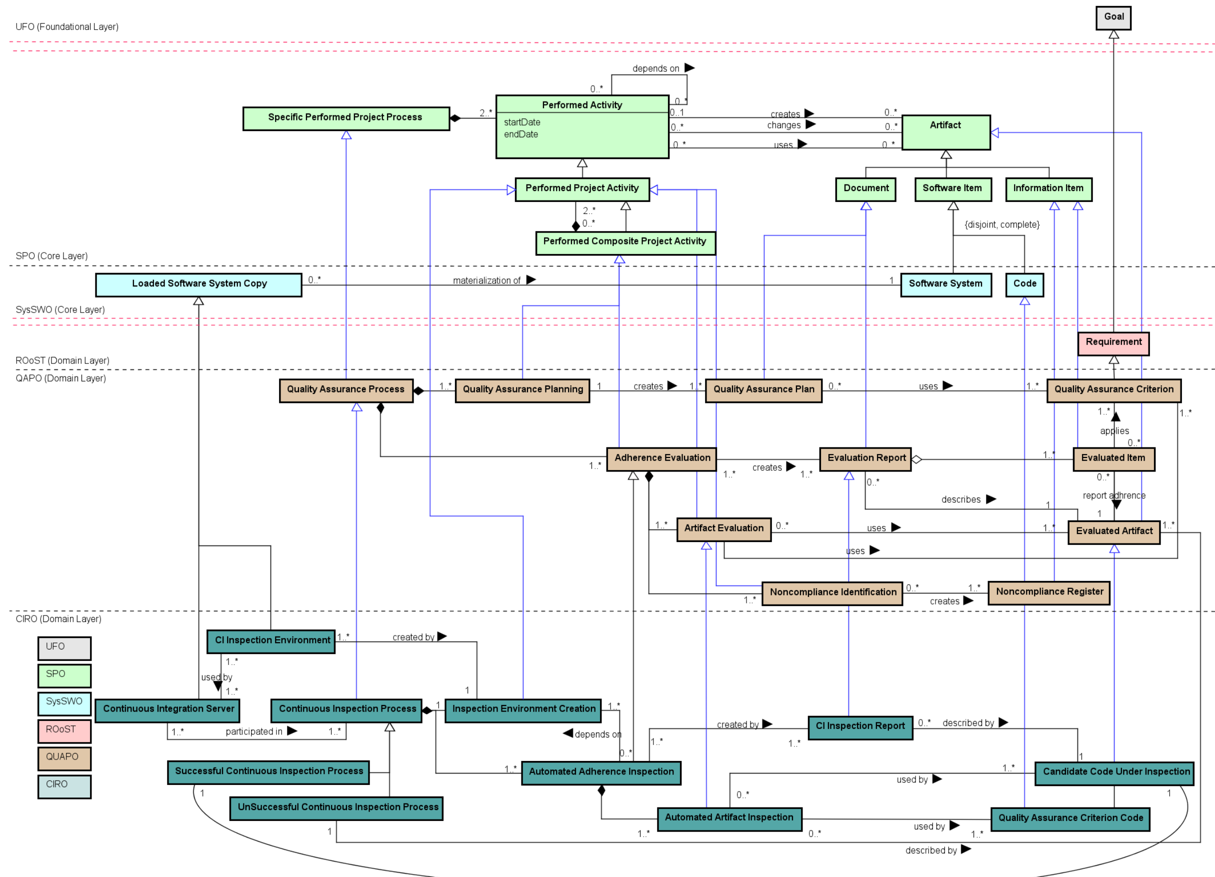


Figure 41 – Continuous Inspection Process Subontology.

<u>*Artifact*</u> plays a <u>*Evaluated Artifact*</u> role in a **Continuous Inspection Process**.

Finally, a **Continuous Inspection Process** is said a **Successful Continuous Inspection Process** when an **Automated Adherence Inspection** produces a **Candidate Code Under Inspection** that does not have any <u>*Noncompliance Identification*</u>. On the other hand, a **Continuous Inspection Process** is an **Unsuccessful Continuous Inspection Process** when the **Candidate Code Under Inspection** failed, a <u>*Noncompliance Identification*</u> was obtained, in an **Automated Adherence Inspection**.

3.3.6 Evaluation

As we did in SRO, to evaluate CIRO, we performed Verification and Validation activities by using two approaches to ontology evaluation: assessment by human (FALBO, 2014) and data-driven approach (BRANK; GROBELNIK; MLADENIC, 2005). Table 3 presents results produced during verification. Table 4 presents instantiations recorded during validation. The instances were extracted from a project developed in the software organization where we performed the study described in Section 4.2.

Table 3 – Verification of CIRO.

#	Competence Question	CIRO Concepts, Properties, and Relationships
CQ01	Which processes and activities make up the Continuous Integration process ?	**Continuous Integration Process** *composed of* **Continuous Build Process** **Continuous Integration Process** *composed of* **Continuous Test Process** **Continuous Integration Process** *composed of* **Continuous Inspection Process** **Continuous Integration Process** *composed of* **Continuous Feedback Activity**
CQ02	In the CI process, on which other activities/processes did a certain activity/process depend?	**Continuous Test Process** *depends on* **Continuous Build Process** **Continuous Inspection Process** *depends on* **Continuous Test Process**
CQ03	When did a CI process start ?	**Performed Project Process**.*startDate* **General Performed Project Process** *subtype of* **Performed Process** **Continuous Integration Process** *subtype of* **General Performed Project Process**
CQ04	When did a CI process end ?	**Performed Project Process**.*endDate* **General Performed Project Process** subtype of **Performed Process** **Continuous Integration Process** *subtype of* **General Performed Project Process**
CQ05	Which artifacts participated in the CI process ?	**Continuous Integration Server** *participated in* **Continuous Integration Process**
CQ06	Which stakeholders participated in the CI process ?	**Stakeholder** **Change Implementer** *subtype of* **Stakeholder** **CI Stakeholder** *subtype of* **Stakeholder**
CQ07	What type of event triggered the CI process ?	**Checkin-triggered CI Process** *subtype of* **Continuous Integration Process** **Scheduled CI Process** *subtype of* **Continuous Integration Process** **On-Demand CI Process** *subtype of* **Continuous Integration Process**
CQ08	Which activities make up a continuous build process?	**Continuous Build Process** *composed of* **Build Environment Creation** **Continuous Build Process** *composed of* **Code Checkout** **Continuous Build Process** *composed of* **Candidate Code Building**
CQ09	Which resources were used to build the software artifacts during the continuous build process?	**Build Environment** *created* **Build Environment Creation** activity

Table 3 – *Continued from previous page*

#	Competence Question	CIRO Concepts, Properties, and Relationships
CQ10	Which artifacts are created during the continuous build process ?	**Source Code Copy**, **Test Code Copy**, and **Candidate Code**
CQ11	Which processes and activities make up a Continuous Test process?	**Continuous Test Process** *composed of* **CI Testing Environment Creation** **Continuous Test Process** *composed of* **Automated Testing**
CQ12	What automatic tests were performed?	**Level-Based Testing** *composed of* **Perfomed Test Execution** **Perfomed Test Execution** *uses* **Test Code**
CQ13	Which processes and activities make up a Continuous Inspection process?	**Continuous Inspection Process** *composed of* **Inspection Environment Creation** **Continuous Inspection Process** *composed of* **Automated Adherence Inspection** **Automated Adherence Inspection** *composed of* **Automated Artifact Inspection**
CQ14	What was the project artifact in (non)conformance with the quality requirements of project?	**CI Evaluation Report** *described* **Candidate Code Under Inspection**

Table 4 – Validation of CIRO.

CIRO Concepts	Instance
Continuous Integration Server	An instance of a GitLab in a computer.
CI Stakeholder	B. S.
Continuous Integration Process	Continuous Integration Process defined to a software project, with identification CIP188, comprising automated Build, Test, Inspection processes. and Feedback activity.
Checkin-triggered CI Process	Continuous Integration Process that was instantiated by GitLab after a commit from a new version of database script created by Barney S., on Oct 10, 2019.
Scheduled CI Process	Continuous Integration Process that is instantiated by GitLab, with identification GL188, on Oct 22, 2019.
On-Demand CI Process	Continuous Integration Process that was instantiated manually by Barney S, on Oct 22, 2019.
CI Request Event	Event, with identification e546, that it was created by Barney S to start an On-Demand CI Process, on Oct 22, 2019.
Continuous Build Process	Continuous Build Process, with identification CBP188, performed in Continuous Integration Process, with identification CIP188, on Oct 22, 2019.

Table 4 – *Continued from previous page*

CIRO Concepts	**Instance**
Continuous Test Process	Continuous Test Process, with identification CTP188 , performed in Continuous Integration Process, with identification CIP188, on Oct 22, 2019.
Continuous Inspection Process	Continuous Inspection Process, with identification CINP188, performed in Continuous Integration Process, with identification CIP188, on Oct 22, 2019.
Continuous Feedback Activity	Continuous Feedback Activity, with identification CFA888, performed in Continuous Integration Process, with identification CIP188, on Oct 22, 2019.
CI Building Environment	An instance of virtual machine, with identification VM188, with libraries that is used to build a code in a software, inside of a instance of GitLab,with identification GL188, on Oct 22, 2019.
Building Hardware Resource	A machine that is used to instance a CI Building Environment on Oct 22, 2019.
Building Software Resource	A compiler installed in CI Building Environment, with identification VM188, on Oct 22, 2019.
Building Environment Creation	a Building Environment Creation activity, with identification BEC188, perfomed in CBP188, on Oct 22, 2019.
Code Checkout	Code Checkout activity performed in CBP188, on Oct 22, 2019
Candidate Code Building	a Candidate Code Building performed in CBP188, on Oct 22, 2019.
Source Code Copy	Source Code copied to Build environment by Code Checkout activity, on Oct 22, 2019.
Test Code Copy	Test Code copied to Build environment by Code Checkout activity, on Oct 22, 2019.
Candidate Code	Candidate Code built by Candidate Code Building, on Oct 22, 2019 .
Build Problem	a Build Problem identified in a Candidate Code Building activity, on Oct 22, 2019 .
CI Testing Environment	is a instance of a Testing Environment created, inside of a Continuous Integration Server, in Continuous Test Process CTP188, on Oct 22, 2019.
CI Testing Environment Creation	CI Testing Environment creation performed in CTP188, on Oct 22, 2019.
Automated Testing	a Unit Testing performed in CTP188, on Oct 22, 2019.
CI Test Result	a report with non-conformance detected by Automated Testing, on Oct 22, 2019.
Candidate Code Under Test	a code that was tested in a CTP188, on Oct 22, 2019.
CI Inspecting Environment	an instance of an environment with library and equipment to inspect a code in a CI Inspection Process, on Oct 22, 2019.
Inspection Environment Creation	Inspection Environment Creation activity in CINP188, on Oct 22, 2019.

Table 4 – *Continued from previous page*

CIRO Concepts	Instance
Automated Adherence Inspection	Automated Adherence Inspection activity performed in CINP188, on Oct 22, 2019.
Automated Artifact Inspection	Automated Artifact Inspection activity performed in CINP188, on Oct 22, 2019.
Quality Assurance Criterion Code	A code that implements a Quality Assurance Criterian code: A function can have a maximum of 300 lines
CI Inspection Report	A CI Inspection Report created in CINP188, on Oct 22, 2019.
Candidate Code Under Inspection	A Database script under quality evaluation in a CINP188, on Oct 22, 2019.

3.4 Continuous Deployment Reference Ontology (CDRO)

The *Continuous Deployment Reference Ontology* (CDRO) consolidates reference literature on the topics, using as main sources the works by (HUMBLE; FARLEY, 2010; SHAHIN; BABAR; ZHU, 2017; HUMBLE; KIM, 2018). CDRO is organized into two subontologies:

- The *Continuous Delivery Activity subontology*: aims at representing how a delivery activity is present in a CD context.
- The *Continuous Deployment Process subontology*: which presents an overview of the CD process, identifying the main events, stakeholders, and artifacts present in this context.

Figure 42 shows an overview of CDRO.

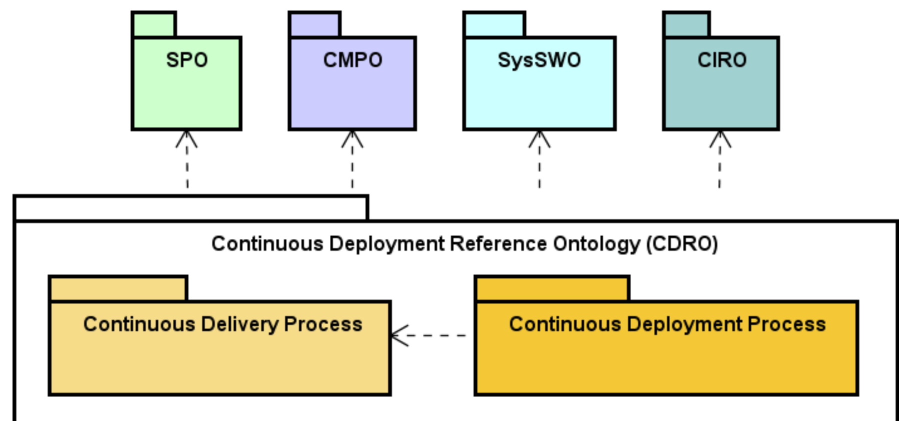


Figure 42 – CDRO's Architecture.

3.4.1 Continuous Delivery Activity Subontology

The *Continuous Delivery Activity* subontology aims to explain how a delivery activity is present in a *Continuous Deployment* process. It aims answering the following competency questions:

- CQ01. When did a delivery activity start?
- CQ02. When did a delivery activity end?
- CQ03. Which artifacts participated in a delivery activity?
- CQ04. What does a Delivered code ?
- CQ05. Which resources did make up a Delivery Environment

CQ01 and CQO2 refer to temporal aspects and aim to provide information about a delivery activity duration. CQ03 concerns the participants (agents or objects) in a delivery activity while CQ04 identify what is a Delivery Code in CD context. CQ05 describes the resources that are used by a Delivery Environment. Figure 43 shows the *Continuous Delivery Activity* subontology.

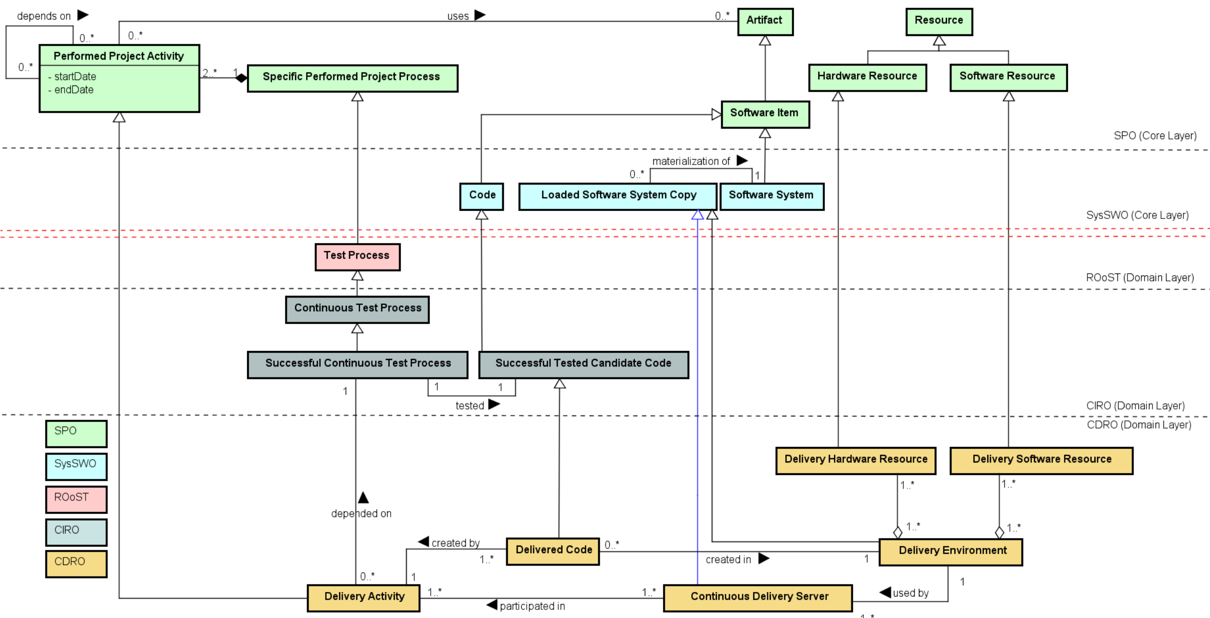


Figure 43 – Continuous Delivery Activity Subontology.

A **Delivery Activity** is an automated <u>*Performed Project Activity*</u> that involved the participation of **Continuous Delivery Server** and delivered a **Delivered Code** in a **Delivery Environment** without human intervention. A **Delivery Activity** is performed after a **Candidate Code** be tested with success in a **Successful Continuous Test Process**. In turn, a **Delivered Code** is a **Success Candidate Code Tested** that was created in a **Delivery Activity**.

A **Continuous Delivery Server** is a <u>*Loaded Software System Copy*</u> (e.g., <u>a copy of GitLab[5] loaded in a computer</u>) that provides <u>*Software artifacts*</u> (e.g., <u>libraries</u> and <u>programs</u>)

[5] https://gitlab.com/

that participated in a **Delivery Activity** to enable executing the activity in an automatic way, without human intervention.

A **Delivery Environment** is a Loaded Software System Copy (e.g., *a copy of delivery environment in a copy of the GitLab loaded in a computer*) that is constituded of **Delivering Software Resources** (e.g., *Operational System (Linux or Windows)*) and **Delivery Hardware Resources** (e.g., a physical machine where the **Delivery Software Resources** are installed and loaded) to support **Delivery Activities**. **Delivery Software Resource** and **Delivery Hardware Resource** are, respectively, Software Resources and Hardware Resource (e.g., *a delivery tool* and *a machine that runs a delivery tool*). A **Delivery Software Resource** is used in a **Delivery Activity** (participates in) via a **Continuous Delivery Server**.

3.4.2 Continuous Deployment Process Subontology

The *Continuous Deployment Process subontology* aims to answer the following competency questions:

- CQ06. Which processes and activities did make up a CD process?
- CQ07. In the CD process, on which other activities/processes did a certain activity/process depend?
- CQ08. When did a CD process start?
- CQ09. When did a CD process end?
- CQ10. What is a Deployed Code?
- CQ11. Which artifacts participated in the CD process?
- CQ12. Which resources did make up a Deployment Environment?
- CQ13. Which stakeholders participated in the CD process?

CQ06 and CQ07 regard subprocesses and activities involved in a CD process. They aim to provide knowledge about the CD process structure and reveal the order required of its subprocesses and activities. CQ08 to CQ09 refer to temporal aspects and aim to provide information about processes and activities duration, contributing to analyzing the CD process performance (SHAHIN; BABAR; ZHU, 2017). CQ10 addresses to identify which is a Deployed Code. CQ11 addresses the participants (agents or objects) in a CD process, while CQ12 is addressed to identify the resources that are used by a Continuous Deployment Server. Finally, CQ13 is concerned with roles played by agents that participated in the CD process.

A **Continuous Deployment Process** is an automated process that performs deployment of a **Deployed Code** in a production-like environment and communicate the stakeholders

about success or failure (HUMBLE; FARLEY, 2010; SHAHIN; BABAR; ZHU, 2017; HUMBLE; KIM, 2018). It is an automated *Specific Performed Project Composite Process* that involved the participation of one or several **Continuous Deployment Servers**, with the purpose of deploying a **Deployed Code** in a **Deployment Environment** without human intervention. Figure 44 presents Continuous Deployment Process subontology.

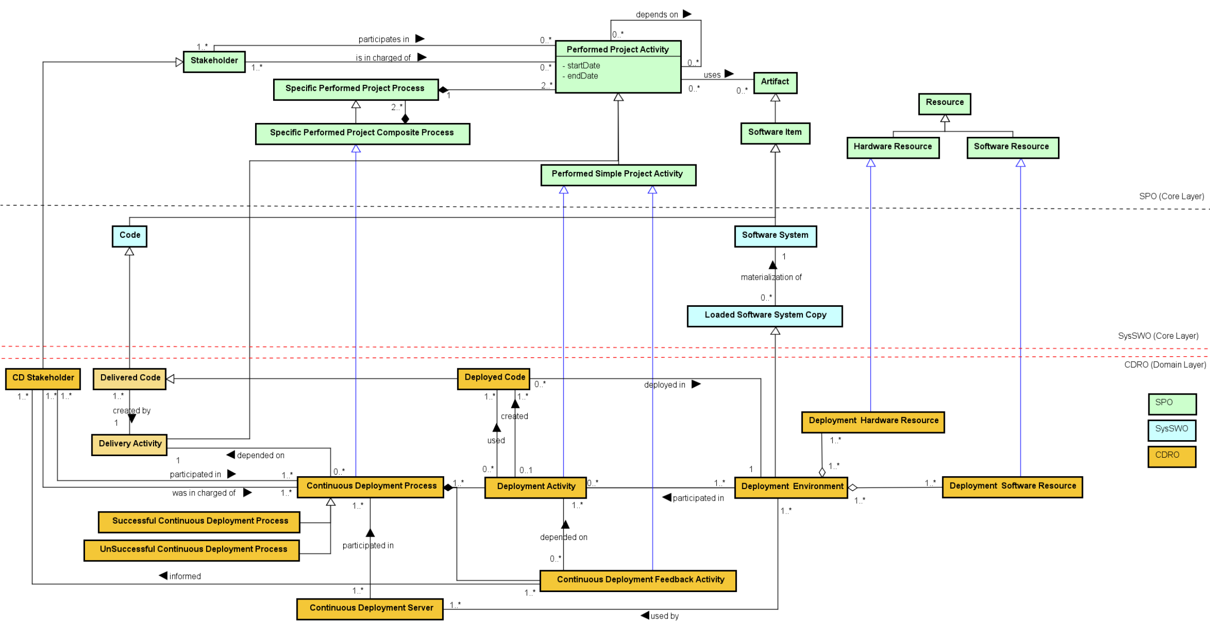


Figure 44 – Continuous Deployment Process Subontology.

A **Continuous Deployment Process** is composed of (i) at least one **Deployment Activity**, which is an automated *Performed Project Activity* that deployed a **Deployed Code** in a **Deployment Environment**, and (ii) **Continuous Deployment Feedback Activity**, an automated *Performed Simple Project Activity* that provided to a **CD Stakeholder** information about the status of a CD process (e.g., send information if a code was deployed with success or failure in an production-like environment). A **CD Stakeholder** is a *Stakeholder* (e.g., a developer or a client) that participates or was charged of a **Continuous Deployment Process**. Besides a **CD Stakeholder** interested in information (e.g., success or failure of the deployment of a version of a code in a production-like environment) about a **Continuous Deployment Process**.

Continuous Deployment Server is a *Loaded Software System Copy* (e.g., a copy of ArgoCD[6] loaded in a computer) that provided *Software artifacts* (e.g., libraries and programs) that participated in a **Continuous Deployment Process** to enable executing the process in an automatic way, without human intervention.

A **Deployment Environment** is a *Loaded Software System Copy* (e.g., *a copy of a machine with linux in a computer*) that contains **Deployment Software Resources** (e.g.,

6 <https://argo-cd.readthedocs.io/en/stable/>

Operational System (Linux or Windows)) and **Deployment Hardware Resources** (e.g., a physical machine where the **Deployment Software Resources** are installed and loaded) to support the **Deployment Activities**. **Deployment Software Resource** and **Deployment Hardware Resource** are, respectively, *Software Products* (e.g., a deployment tool) and *Hardware Equipment* (e.g., a machine that runs a deployment tool). A **Deployment Software Resource** is used in a **Deployment Activity** via (*participates in*) a **Continuous Deployment Server**.

Finally, a **Continuous Deployment Process** is a **Successful Continuous Deployment Process** when a **Deployed Code** is deployed in a production environment, without problems. On the other hand, a **Unsuccessful Continuous Deployment Process** is a **Continuous Deployment Process** that has not deployed a **Deployed Code**, due to a problem in the **Continuous Deployment Process**'s processes or activity (e.g., a machine without adequate resources to operate a **Deployed Code**)[7].

3.4.3 Evaluation

As we did in SRO and CIRO, to evaluate CDRO, we performed Verification and Validation activities by using two approaches to ontology evaluation: assessment by human (FALBO, 2014) and data-driven approach (BRANK; GROBELNIK; MLADENIC, 2005). Table 5 presents results produced during verification. Table 6 presents instantiations recorded during validation. The instances were extracted from a project developed in the software organization where we performed the study described in Section 4.2.

Table 5 – Verification of CDRO.

#	Competence Question	CDRO Concepts, Properties, and Relationships
CQ01	When did a Delivery Activity start ?	**Performed Project Activity**.*startDate* **Delivery Activity** *subtype of* **Performed Project Activity**
CQ02	When did a Delivery Activity end ?	**Performed Project Activity**.*endDate* **Delivery Activity** *subtype of* **Performed Project Activity**
CQ03	Which artifacts participated in the Delivery Activity ?	**Delivered Code** *created by* **Delivery Activity** **Delivered Code** *created in* **Delivery Environment** **Continuous Delivery Server** *participated in* **Delivery Activity** **Delivery Environment** *used by* **Continuous Delivery Server**
CQ04	What is a Delivery Code	**Successful Tested Candidate Code** **Successful Tested Candidate Code** *subtype of* **Delivered Code**
CQ05	Which resources did make up a Delivery Environment?	**Delivery Environment** *composed of* **Delivery Hardware Resource** **Delivery Environment** *composed of* **Delivery Software Resource**

[7] The absence of detailed discussions on deployment strategies and techniques within this thesis is due to constraints in research scope and timeline. Future studies are encouraged to explore this area for a more

Table 5 – *Continued from previous page*

#	Competence Question	CDRO Concepts, Properties, and Relationships
CQ06	Which processes and activities make up the Continuous Deployment process ?	**Continuous Deployment Process** *composed of* **Deployment Activity** **Continuous Deployment Process** *composed of* **Continuous Deployment Feedback Activity**
CQ07	In the Continuous Deployment process, on which other activities/processes did a certain activity/process depend?	**Continuous Deployment Process** *depends on* **Delivery Activity** **Continuous Deployment Feedback Activity** *depends on* **Deployment Activity**
CQ08	When did a Continuous Deployment process start ?	**Performed Project Process**.*startDate* **General Performed Project Process** *subtype of* **Performed Process** **Continuous Deployment Process** *subtype of* **Specific Performed Project Composite Process**
CQ09	When did a Continuous Deployment process end ?	**Performed Project Process**.*endDate* **General Performed Project Process** subtype of **Performed Process** **Continuous Deployment Process** *subtype of* **Specific Performed Project Composite Process**
CQ10	What is a Deployed Code?	**Delivered Code** **Deployed Code** *subtype of* **Delivered Code**
CQ11	Which artifacts participated in the Continuous Deployment process ?	**Continuous Deployment Server** *participated in* **Continuous Deployment Process** **Delivered Code** *used* **Deployment Activity** **Delivered Code** *created* **Deployment Activity** **Deployment Environment** *used by* **Continuous Deployment Server**
CQ12	Which resources did make up a Deployment Environment?	**Deployment Environment** *composed of* **Deployment Hardware Resource** **Deployment Environment** *composed of* **Deployment Software Resource**
CQ13	Which stakeholders participated in the Continuous Deployment process ?	**Stakeholder** **CD Stakeholder** *subtype of* **Stakeholder**

comprehensive understanding.

Table 6 – Validation of CDRO.

CDRO Concepts	**Instance**
Delivery Environment	An instance of virtual machine, with identification VM188, with libraries that is used to delivery, inside of an instance of GitLab, with identification GL188, on Oct 22, 2019
Delivery Hardware Resource	A machine that is used to instance a Delivery Environment, on Oct 22,2019
Delivery Software Resource	A delivery software installed in a Delivery Environment, with identification VM188, on Oct 22,2019
Continuous Delivery Server	An instance of GitLab, named GL01, loaded in a computer.
Delivery Code	A code that was tested in a CTP 188, on Oct 22,2019, and delivery in GL01.
Delivery Activity	Delivery Activity defined to a software project, with identification CDEP188.
Deployment Environment	An instance of virtual machine, with identification VM188, with libraries that is used to deploy, with identification DPE188, on Oct 22, 2019
Deployment Hardware Resource	A machine that is used to instance o Deployment Environment, on Oct 22,2019
Deployment Software Resource	A deployment software installed in a Deployment Environment, with identification VM188, on Oct 22,2019
Continuous Deployment Server	An instance of ArgoCD loaded in a computer.
CD Stakeholder	L. A.
Continuous Deployment Process	Continuous Deployment Process defined to a software project, with identification CDP188, composed of automated deploy, and Feedback activity.
Deployment Activity	A Deployment Activity instance performed in CDP188, on Oct 22,2019.
Deployed Code	A delivered code created and used in a Deployment Activity instance performed in CDP188, on Oct 22,2019.
Continuous Deployment Feedback Activity	Continuous Deployment Feedback Activity, with identification CDFA888, performed in Continuous Deployment Process, with identification CDP188, on Oct 22, 2019.
Successful Continuous Deployment Process	Continuous Deployment process,with identification CDP188, that performed without erro, on Oct 22, 2019.
Unsuccessful Continuous Deployment Process	Continuous Deployment process,with identification CDP188, that performed with build's error, on Oct 22, 2019.

3.5 Related Work

We discuss here some related efforts with respect to ontologies and metamodels that cover similar aspects to SRO, CIRO, and CDRO.

Regarding ontologies addressing Scrum, the studies most closely related to SRO were conducted by Parsons (2010), Kiv et al. (2019), and Lin et al. (2012). Parsons (2010) presented a general ontology on agile methods to propose an analytical framework to understand how an overarching agile methodology is constructed. Lin et al. (2012), in turn, introduced a Scrum ontology based on concepts from CRIO metamodels (COSSENTINO et al., 2007), modelled using OWL. CRIO is an organisational metamodel with four concepts (Role, Interaction, Organisation, and Holon) used to model and design Multi-Agent Systems (COSSENTINO et al., 2007; GALLAND; GAUD; KOUKAM, 2008). Kiv et al. (2019) proposed an agile method ontology modeled using UML and implemented with OWL to represent knowledge about projects. Differently from SRO, the works by Parsons (2010) and Kiv et al. (2019) propose general ontologies about the agile paradigm, describing methods and goals without a focus on Scrum. Lin et al. (2012), in turn, propose a lightweight ontology, which provides a limited conceptualization. Moreover, these ontologies are not connected to other aspects of Software Engineering. SRO describes the conceptualization about Scrum in the Software Engineering context. Thus, SRO concepts are related to concepts from other Software Engineering sub-domains such as Requirements and Software Process.

In addition to ontologies covering similar ground, the Method Engineering field has produced some metamodels concerned with agile methods. This is the case of Damiani et al. (2007), who present Scrum metamodels using MOF (Meta-Object Facility), and Ayed, Vanderose & Habra (2012), who introduces an approach to model an agile process according to an organization's characteristics, based on Situational Method Engineering (SME) (HARMSEN; BRINKKEMPER; OEI, 1994) and using SPEM (Systems Process Engineering Metamodel). Different from SRO, the metamodel proposed in (DAMIANI et al., 2007) focuses only on few concepts related to the ceremonies and backlog, resulting in a limited view of Scrum. In (AYED; VANDEROSE; HABRA, 2012), the metamodel concerns agile development in general, and as such, does not address specific aspects of Scrum or other particular agile methods. Because of this, the proposed metamodel is defined at a rather abstract level, hindering its use as semantic grounding for operational data that is handled by the various tools. Moreover, the purpose of the proposed models is not to provide a comprehensive conceptualization able to address semantic issues. Instead, they are intended to support process/method definition and evolution.

Regarding the Continuous Integration domain, the study most closely related to CIRO was presented by Moriconi et al. (2022). Moriconi et al. (2022) presents an operational ontology for representing builds in a continuous integration process. This ontology was implemented as a knowledge graph (FENSEL et al., 2020)[8] to identify failed builds caused by infrastructure issues (DURIEUX et al., 2020). Different from work proposed by Moriconi et al. (2022), CIRO is a reference ontology that describes the conceptualization of Continuous Integration in the Software Engineering context. Therefore, the CIRO concepts are related to concepts from other

[8] No other reference or operational ontology was found in the continuous integration literature.

Software Engineering sub-domains such as Quality and Test Processes.

Concerning the Continuous Deployments, the studies most closely related to CDRO weres presented by Macarthy & Bass (2020), Pardo, Orozco & Guerrero (2023), and Guerrero, Calvache & Orozco (2023) about DevOps (HUMBLE; KIM, 2018). Macarthy & Bass (2020) present a study performed with 11 DevOps practitioners, across nine organizations, to define an empirical taxonomy of DevOps; while Pardo, Orozco & Guerrero (2023) present the DevOps ontology that describe in high level the DevOps's concepts (e.g., Activity, Practice, and Principle). It was created using REFSENO methodology (TAUTZ; WANGENHEIM, 1998), modeled in UML, implemented in OWL. Guerrero, Calvache & Orozco (2023) present general concepts from the literature about DevOps (Dimension, Approach, Principle, Value, and Technological Tool), process concepts (e.g., ROI, Activity, Product, and Task) from the Ontology of Process-reference Models (PrMO) (CALVACHE et al., 2014), and measurement concepts (e.g., Indicator, Measure, Measurement, and Scale) from the Software Measurement Ontology (SMO) (GARCÍA et al., 2006). Different from the works proposed by Macarthy & Bass (2020), Pardo, Orozco & Guerrero (2023), and Guerrero, Calvache & Orozco (2023), CDRO is a reference ontology that considers a more holistic view of CD and describes its conceptualization in the Software Engineering context. Therefore, like in CIRO, CDRO concepts are related to concepts from other Software Engineering sub-domains such Testing.

Due to the strong connection between the Scrum, CI, and CD processes and other Software Engineering aspects, SRO, CIRO, and CDRO were developed as networked ontologies of SEON (RUY et al., 2016). This modeling decision allowed us to reuse concepts from other SEON ontologies and achieve a broad understanding about Scrum, CI, and CD in the Software Engineering context. For example, by connecting SRO to other SEON ontologies it is possible to understand that User Story is a Requirements Artifact and, as such, describes requirements of stakeholders of the project. It is also possible to understand that a Sprint is a process composed of activities performed during a time box. Understanding the Scrum, CI, and CD processes in the context of the Software Engineering domain contributes to a better understanding of the conceptualization and to make comparisons or integration of information regarding different paradigms. For example, by acknowledging that User Story is a Requirement Artifact, when looking at information about projects developed using different process models, one can understand that, in a project that adopts the Scrum process, the User Story plays the same role than the Requirement Description plays in a project that adopts the Waterfall process, since both are types of Requirements Artifacts in SEON. This broad analysis is not possible in any of the cited works.

Another difference of SRO, CIRO, and CDRO when compared to the aforementioned works, is that SRO, CIRO, and CDRO provide a more precise conceptualization. For example, SRO establishes the meaning of "Done" and what are the impacts of this concept in different aspects of a software process based on Scrum. Finally, being networked ontologies of SEON,

SRO, CIRO, and CDRO are ultimately grounded in UFO (the Unified Foundational Ontology) (GUIZZARDI, 2005), which results in a well-founded conceptualization that can better represent real-world situations.

Considering *Continuum* as a whole, the closest work to it was presented by Guerrero, Calvache & Orozco (2023). They propose the DevOps Ontology, which is an ontology to support the understanding of DevOps (LEITE et al., 2019), presenting general concepts from the literature about DevOps, process concepts, from the Ontology of Process-reference Models (PrMO) (CALVACHE et al., 2014), and measurement concepts, from the Software Measurement Ontology (SMO) (GARCÍA et al., 2006). Although Guerrero, Calvache & Orozco (2023) claims that DevOps Ontology supports the understanding of DevOps, the ontology does not present any specific process, role, or artifact related to DevOps that would allow one to understand how DevOps works.

Different from DevOps Ontology, *Continuum* is an ontology (sub)network integrated to SEON (RUY et al., 2016). It describe the processes, roles, and artifacts involved in the CSE core processes. In addition, *Continuum* is strongly connected with Software Engineering aspects. As said before, this modeling decision allows us to reuse concepts from SEON ontologies and achieve a broad understanding of the CSE in the Software Engineering context. Furthermore, being part of SEON, *Continuum* is supported by evolution mechanisms that help extend its conceptualization to cover other CSE aspects.

3.6 Final Considerations

Continuum is a subnetworked of SEON (RUY et al., 2016) and reuses concepts from UFO (GUIZZARDI, 2005), SPO (BRINGUENTE; FALBO; GUIZZARDI, 2011; RUY, 2017), EO (FALBO, 2014), SysSWO (DUARTE et al., 2018b; COSTA et al., 2022), CMPO (CALHAU; FALBO, 2012), RSRO (FALBO; NARDI, 2008) , ROoST (SOUZA; FALBO; VIJAYKUMAR, 2017), QAPO (RUY, 2017), and OSDEF (DUARTE et al., 2018a). The number of ontologies reused shows the comprehensiveness of the scope of the CSE processes addressed, crosscutting the software lifecycle.

The current version of *Continuum* is composed of the *Scrum Reference Ontology* (SRO), the *Continuous Integration Reference Ontology* (CIRO), and *Continuous Deployment Reference Ontology* (CDRO). SRO provides conceptualization about agile development with Scrum and consists of five subontologies (Scrum Process subontology, Scrum Stakeholders subontology, Scrum Stakeholders Participation subontology, Product and Sprint Backlog subontology, and Scrum Deliverables subontology). CIRO, in turn, addresses relevant aspects of continuous integration and is composed of four subontologies (Continuous Integration Process subontology, Continuous Build Process subontology, Continuous Test Process subontology, and Continuous Inspection Process subontology). CDRO concerns aspects of continuous deployment and is

composed of two subontologies (Continuous Delivery Activity subontology and Continuous Deployment Process subontology).

Continuum provides the conceptualization necessary to support semantic integration in *Immigrant*. Different from other works, *Continuum* introduces processes, roles, and artifacts involved in the CSE core processes (agile development, continuous integration, and continuous deployment) and relates them to more general Software Engineering concepts. Details about how *Continuum* is used in *Immigrant* to support application data integration are discussed in the Chapter 5.

4 Learning Iterations Towards Immigrant

Many times I've lied, many times I've listened, many times I've wondered how much there is to know.

Led Zeppelin, Over The Hills And Far Away

This chapter presents an overview of the studies carried out in the three learning iterations performed to design *Immigrant*. They helped us better understand the target problem and gradually evaluate the design choices we have made to develop *Immigrant*. Thus, this chapter is related to the *Design Cycle*, because it regards studies in which we developed the first version of Immigrant and components that were incorporated in the current version. It is also related to the *Relevance Cycle* because the studies aided problem understanding. Section 4.1 briefly introduces the learning iteration concept. Section 4.2 regards the first learning iteration, an exploratory case study that resulted in the first version of *Immigrant*. Section 4.3 concerns the second learning iteration, a participative case study in which we proposed *California*, a System Theory-based process to aid organizations in the journey from traditional to data-driven software development. Section 4.4 presents the third learning iteration, a multiple case study in which we proposed *Zeppelin*, a diagnosis instrument that helps get an overview of CSE practices adoption in software organizations. Finally, Section 4.5 presents the final considerations of the chapter.

4.1 Learning Iterations

Several authors advocate the use of empirical studies in DSR projects. These studies have often been used to evaluate the proposed artifacts (BASKERVILLE, 2008; WIERINGA, 2014). According to Barcellos et al. (2022), although the use of empirical studies in DSR projects has focused on the artifact evaluation activity, they can also be used to support other activities of the DSR process. The authors suggest organizing empirical studies as learning iterations (LIs), i.e., studies performed in iterations that allow the researcher to learn something about the problem or the proposed artifact. In this context, LIs provide useful knowledge to understand the problem, develop the artifact, evaluate, or improve it. The authors argue that, by using LIs, the researcher experiences a more fluid DSR process, which harmonizes several studies in iterations, contributing to research soundness. In addition, the studies also contribute to the development of better-grounded artifacts, which are expected to be more suitable for solving the target problem.

Each LI comprises a study that aims at building knowledge guided by knowledge questions that motivate the study. LI are categorized according to their purpose, which can be

(i) problem investigation; (ii) artifact foundation; (iii) artifact design; (iv) artifact evaluation; and (v) artifact evolution (BARCELLOS et al., 2022). Figure 45 depicts schematically the elements involved a LI.

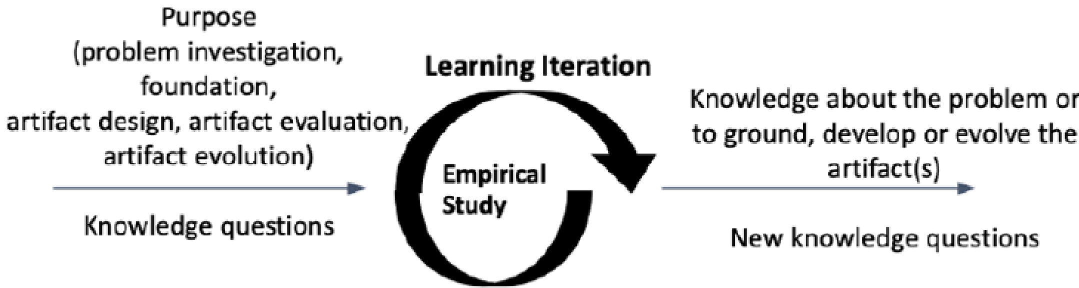


Figure 45 – Learning Iteration (BARCELLOS et al., 2022).

The study purpose and the knowledge questions help define the empirical study to be performed (e.g., a systematic literature review, a case study, a controlled experiment) and its scope. After performing the study, the researcher gets knowledge about the problem or to ground, develop or evolve the artifact(s). New knowledge questions can also arise and lead to another study in a new LI. In a nutshell, LI contains the LI plan, the empirical study package, and the LI report. The plan establishes the LI purpose and knowledge questions, and identifies which empirical study will be carried out and how it connects to the DSR project as a whole. The study package refers to the execution of the LI, which involves activities related to the referred empirical study, including to define the research protocol to be followed, run it to collect and analyze data, and record the study results. The LI is concluded with a report that highlights the acquired knowledge and connects the study results with the DSR project (BARCELLOS et al., 2022).

As explained in Chapter 1, we performed three empirical studies as LIs aiming at the *Immigrant* development. Figure 46 illustrates an overview of them. In the following sections, we provide information about each LI, including (i) an overview of the LI scope by means of its purpose, study type and knowledge question; (ii) the study execution and main results; and (iii) the acquired knowledge and its relation to the whole research. Concerning the studies execution, in this chapter we provide only an overview of them. The complete description of the studies and their results are available in publications and supplementary materials indicated in the next sections.

4.2 First Learning Iteration: Towards an Ontology-Based Approach to Integrate Data Application

As discussed in chapters 1 and 2, application integration should take semantic aspects into account. Therefore, to develop the artifact aimed by this work, we decided to integrate

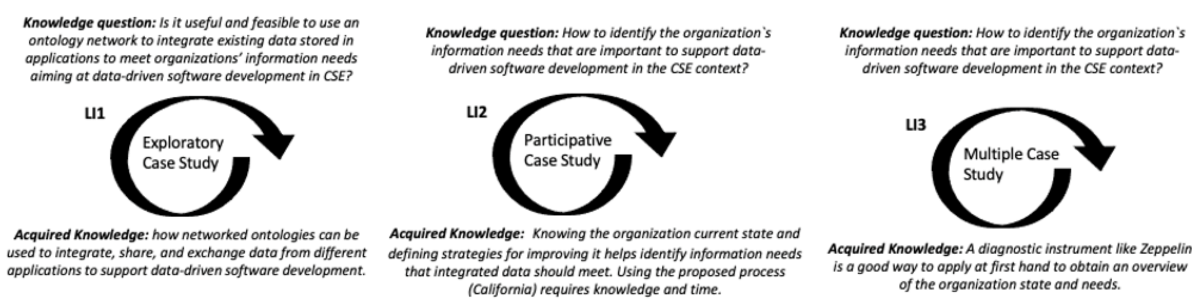


Figure 46 – Overview of the LIs performed to develop *Immigrant*.

application data by using networked ontologies to assign semantics to data and, thus, support semantic integration. This decision was made to meet requirement R2: *the proposed artifact must address semantic issues involved in data integration in CSE domain*, (see Section 1.4). We chose to use networked ontologies because CSE involves several subdomains related to each other and to SE general concepts. Thus, semantic aspects in CSE can be better supported by an integrated, consistent, and comprehensive conceptualization provided by an ontology network.

We performed the first learning iteration with the purpose of evaluating our design choice of using networked ontologies to integrate application data. It consisted of an *exploratory case study*[1] to answer the following question: *Is it useful and feasible to use an ontology network to integrate existing data stored in applications to meet organizations' information needs aiming at data-driven software development in CSE?*

The study was performed in the software development unit of Prodest[2], a Brazilian government agency responsible for IT (Information Technology) solutions in the state of Espírito Santo that faced problems to integrate application data to support data-driven software development. The unit consists of around 50 employees, organized in various teams. Software managers involved in projects adopting Scrum reported the need to obtain integrated data to better monitor software projects, time-tracking, and product quality.

Microsoft Azure DevOps[3], an application that supports project management in agile software projects, and Clockify[4], an application that supports time-tracking, were used in projects at Prodest in a complementary way. While Microsoft Azure DevOps has been used to aid in project management in general (e.g., to create a new project, to record scope by means of user stories, to define tasks and to allocate them to a team), Clockify has been used to enable detailed control of tasks duration, schedule, effort, and cost. Thus, general data about the project (e.g., concerning sprints, user stories, and related tasks) is handled by Microsoft Azure DevOps, while detailed data about how the tasks were performed over time (e.g., concerning time entries,

[1] It is empirical research that seeks to better understand a contemporary phenomenon, usually complex, in its real context (DRESCH; LACERDA; JÚNIOR, 2015).
[2] <https://prodest.es.gov.br/>
[3] <https://azure.microsoft.com/pt-br/services/devops/>
[4] <https://clockify.me>

effort, and cost related to each time entry) is handled by Clockify. Since the applications were not integrated, redundant data needed to be manually entered in both of them (e.g., the same task was created in both applications), and when integrated data was needed (e.g., data about how tasks that implemented user stories of a given sprint were performed), human intervention was needed to retrieve data from both applications and integrate them by using spreadsheets.

For example, to obtain information about the amount of hours spent on development tasks in a given sprint, managers used to perform the following procedure: (i) export a spreadsheet from each application with data about the project, (ii) select from Microsoft Azure DevOps spreadsheet tasks related to the desired sprint; (iii) retrieve from Clockify spreadsheet data corresponding to the selected tasks; (iv) for each task, sum hours of all its time entries recorded in Clockify spreadsheet; (v) record data resulting from (ii) and (iv) in a new spreadsheet and sum time spent on all the tasks. This process demands effort and has to be repeated every time integrated data is needed. If any mistake is made over the process, the resulting information can be incorrect and lead to poor decision support. Moreover, manual consistency management was required in the two applications. For example, when a development task was created in Microsoft Azure DevOps, a team member had to create the same task in Clockify, where task execution is controlled in detail. When the task execution was concluded, the member had to change its status from "In Progress" to "Done" in Microsoft Azure DevOps and Clockify manually. These manual activities naturally create opportunities for data inaccuracy. Ideally, when a task is concluded in Clockify, the number of hours recorded in all its time entries should be summed up, and automatically recorded in Microsoft Azure DevOps and the task status should be changed to "Done".

To address the lack of off-the-shelf integration that hampered data-driven software development, we used an extract of Continuum (focusing on SRO concepts) to build a solution that integrates Microsoft Azure DevOps and Clockify, enabling automatic sharing and exchange of data between the applications and providing consolidated data useful for decision making. Next, we present general information about the study and its results. Detailed information can be found in (SANTOS et al., 2021).

4.2.1 Execution and Results

The study was performed from August to December of 2019. To develop the integration solution, we defined and followed the process illustrated in Figure 47, which is constituted of two macro-activities: *Conceptual Integration* and *Integration Design and Implementation*. The ontology (i.e., the fragment of *Continuum* relevant to the application domain) was used to assign semantics at the conceptual level.

The first activity, *Conceptual Integration*, uses ontology as a basis to identify semantic mappings that will serve as a basis for data integration. For that, it is necessary to *Transform Ontology Model into Information Model*. An information model concerns what kind of information

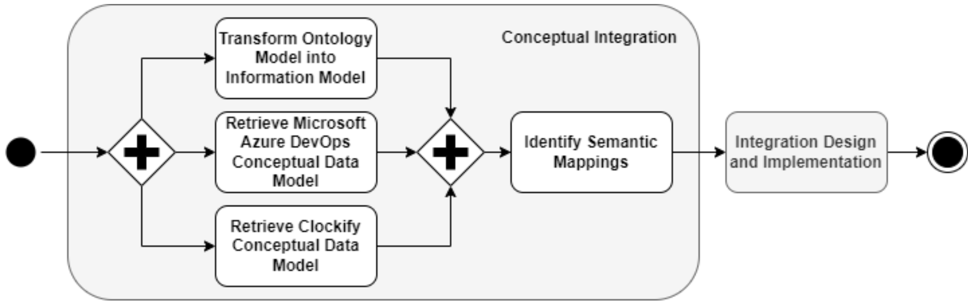


Figure 47 – Process followed to develop the integration solution (SANTOS et al., 2021).

may be stored and exchanged considering demands of specific agents (the "recorded world"), while an ontology model concerns metaphysical aspects of a domain (i.e., it concerns what is considered to exist in the "real world") (CARRARETTO, 2012). Thus, by turning an ontological model into an information model, the resulting model preserves the conceptualization in a structure more suitable for computing demands. Besides the information model, to produce the integration model, it is necessary to *Retrieve the Conceptual Data Model* of each application. By analyzing Microsoft Azure DevOps, Clockify, and their documentation, we retrieved their conceptual data models.

Once we have obtained the applications' conceptual data models and the ontology information model, we needed to *Identify Semantic Mappings*. We used the ontology information model to establish the mappings, assigning semantics to application elements by relating them to the ontology elements. The semantic mappings are important to implement the integration rules to enable service integration. The semantic mappings use ontology as a bridge between the applications and identify which elements of the different applications are equivalent according to the ontology conceptualization. By doing that, it is possible to know which data can be integrated and how they must be stored.

Once we have assigned semantics to applications' elements by means of semantic mappings, we performed *Integration Design and Implementation*. In this activity, we developed software artifacts (database, code libraries, services, and dashboard) and combined them into integration processes that coordinate data integration in the integration solution architecture. Details about the integration architecture and implementation are available in (SANTOS et al., 2021). To provide the integration solution users with data to support decision-making, we created a dashboard that shows, among others: data regarding some agile metrics (e.g., Work in Progress (WIP) and Lead Time), key process indicators (KPIs) (e.g., number of projects and the average amount of hours spent on each project), user stories where story points are missing, a total of hours spent on tasks, total of deliverables considered "done". Figure 48 shows a fragment of the dashboard.

After being developed, the integrated solution was used in a real project with 14 team members (two Scrum masters and 12 developers) lasting from 2018 to 2020. From August to December of 2019, the doctorate candidate (playing the Scrum master role) and the other Scrum

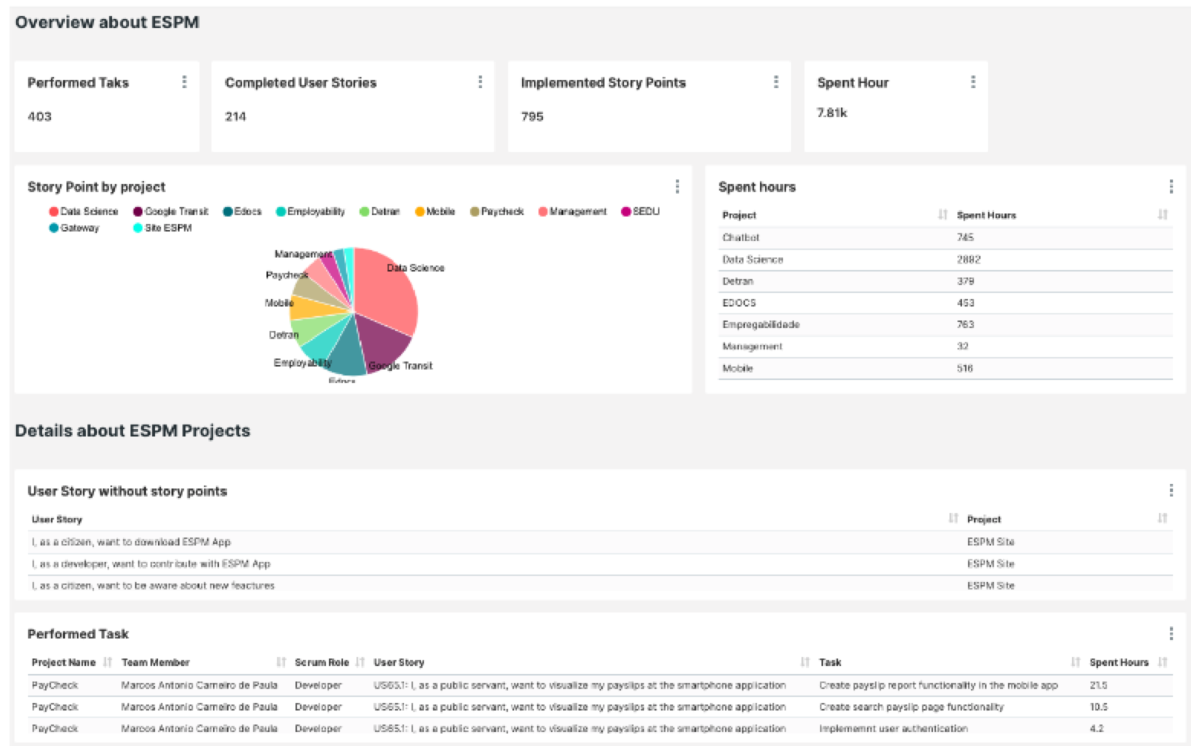


Figure 48 – Fragment of the dashboard showing integrated data (SANTOS et al., 2021).

master used the integrated solution to facilitate project management. During this period, they accessed the dashboard and shared information with the team in biweekly meetings. After that, the doctorate candidate performed semistructured interviews with the other Scrum master and nine developers to evaluate the solution.

The interviewees reported that based on information provided in the dashboard, it was possible to make decisions along the project and also to create more realistic plans to new projects. For example, decisions about team member allocation and duration estimates became more accurate. Team member allocation was performed considering information about previous allocations provided in the dashboard. Therefore, to allocate team members to tasks, it was possible to look for team members that have worked on projects or tasks with similar characteristics (considering functional and non-functional requirements). Effort and duration estimates were performed based on information about the member's productivity. By using historical data available at the dashboard to support estimates, an average deviation of 7.3% in estimates was achieved, which is smaller than the average deviation (24%) in projects that started before August/2019. During the interview, a developer reported that the first sprints of the projects used to have inaccurate time planning and pointed out that the dashboard aided in better estimate the duration of each task. Another developer said that data provided in the dashboard allowed to monitor the team's velocity, contributing to observe the team's evolution and considering its velocity to plan the sprints more realistically.

It was also noted that using the integrated solution the development team truly under-

stood the meaning of "Done" in a Scrum project since only deliverables in conformance to all acceptance criteria appeared in the "Done" section of the dashboard. This operationalization of the notion in the integrated tool helped the development team to understand that it is necessary to deliver valuable and accepted software artifacts in each sprint, contributing to product quality and team performance as well as improved project planning. In this context, a developer reported that by properly understanding the meaning of "Done" and knowing the individual and team's WIP, the team members were encouraged and motivated to increase their performance and produce better deliverables.

The integrated solution also enabled the identification of problems in the execution of the Scrum process in the project. For example, the dashboard provides information about user stories without story points (i.e., estimated effort to implement the user story) defined to them. This may lead the Scrum master to ask the team to complete the information. In the project, when asking some team members to set the story points, it was realized that being novices in Scrum, they did not know how to properly estimate using story points. Thus, training in this matter was provided to the team and the problem was solved.

The Scrum master highlighted that the integrated solution helped to address problems earlier (as in the example cited above) and aided the team to change its practices and work processes "on the fly", according to the information provided in the dashboard, promoting self-organization. Moreover, she said that the information provided in the dashboard helped her make decisions together with developers, which increased their engagement. Another benefit of using the integrated solution was the automatic synchronization of data between the applications, which decreased manual work, avoided errors due to manual manipulation of the same data in different applications, and contributed to the team focusing their attention on development activities.

4.2.2 What did we learn?

The main lesson we learned from this study is that *using networked ontologies is viable and useful to address semantics and produce an integrated solution that aids in improving software development work and supports decision making in the CSE context.* This corroborates our design choice for the artifact aimed by this work to meet R2. Moreover, by developing the integrated solution created in the study, we also learned how to use networked ontologies to integrate, share and exchange data from different applications to support data-driven software development. The process, architecture, and implementations created during the study represent the initial version of *Immigrant.*

Concerning the integration solution, we observed that implementing integration at service layer may contribute to data integration. The opposite (i.e., data integration is needed to service integration) is quite well-known (IZZA, 2009), but, besides that, we realized that it may be advantageous to implement integration at the service layer to facilitate data integration

among applications. During the software development process, it is common the use of different applications referring to the same entity (i.e., a task, a user story). However, sometimes each application addresses different aspects of that entity. For example, concerning task, Microsoft Azure DevOps addresses the person responsible for it and whether the task is done; while Clockify addresses the task start and end dates. Considering that Microsoft Azure DevOps and Clockify are not integrated, the tasks are created in each application separately. Thus, pieces of information about a task are stored in the Microsoft Azure DevOps repository while other pieces are stored in the Clockfy repository. If the correspondence between the task created in Microsoft Azure DevOps and the one created in Clockify is not correct (e.g., if they are recorded with different names), data integration will fail because it will not be possible to identify the same task in the different applications. To help in this matter, service integration should synchronize the applications (e.g., triggered by events performed during the software process). For example, when a task is created in Microsoft Azure DevOps, a corresponding task is automatically created in Clockify, with the same name and responsible, ensuring data consistency and facilitating data integration.

In this study, we also noticed that data provided in the dashboards was useful to the organization because it was aligned with the organization's needs. Thus, we confirmed in practice what the literature points out (JONES et al., 2020): *data must meet the organization information needs, otherwise it is useless.* However, we also observed that, it may not be easy for organizations to identify which information should be satisfied. In the study, we first needed to understand the organization goals, how it worked, and which practices it performed so that we were able to identify its information needs. Then, considering the applications used by the organization and the available data, we decided which information should be presented in the dashboard.

At Prodest, where we performed the participative case study, different projects can use different practices. Thus, we noticed that they could have different information needs. Therefore, we learned that although the applications used by organizations are essential to decide which data will support data-driven software development (after all we need the stored data to provide integrated data), organizations[5] using the same applications (e.g., Microsoft Azure DevOps, Github, and Clockify) can work in different ways, even when performing the same software processes (e.g., agile development, continuous integration, continuous deployment). Thus, it is necessary to understand how the organization perform such processes (e.g., which practices are adopted, at which level (e.g, project, team, process, organization). Moreover, the same information need may demand different data to be satisfied in different organizations. For example, if an organization can implement agile development using epics, user stories, tasks, and sprints while another organization can use tasks and sprints only, and both want to know the average cost to release. In this case, different data will be needed to meet that information

[5] This can also be true for different organizational units, projects or teams of the same organization.

need in each organization.

By observing that, we learned that *understanding the CSE practices adopted by the organization and how they are adopted help identify the organization's information needs.* This knowledge is related to R1 (*the artifact must support identifying the organization's information needs that are important to support data-driven software development in CSE*) and led us to perform the two next LIs.

4.3 Second Learning Iteration: California

In the first study, we confirmed in practice the necessity of properly identifying the organization's information needs to guide data integration and we noticed that organizations may face difficulties to identify such information. Moreover, we observed that understanding how the organization works and the CSE practices it adopts is useful to identify the organization's information needs. Based on that, and considering the artifact requirement R1 (*the artifact must support identifying organization's information needs that are important to support data-driven software development in CSE*) (see Section 1.4), we performed the second learning iteration with the purpose of proposing a way to support understanding how an organization works (in terms of CSE) and, thus, help identify its information needs.

The study was a *participative case study*[6] that aimed at answering the following question: *How to understand the way an organization works and, thus, help identify its information needs relevant to data-driven software development in the CSE context?*

The study was performed in a Brazilian software house (here called Organization A for anonymity reasons) which decided to evolve from traditional to agile and data-driven software development by following the StH model (OLSSON; ALAHYARI; BOSCH, 2012). The organization had made a previous unsuccessful attempt to implement agile practices and needed an approach that was more suitable for its characteristics.

Organization A has a particular characteristic that needs to be taken into account when defining strategies to implement agile practices: the software projects of Organization A are built in partnership with an European organization (here called Organization B). In this partnership, Organization B is responsible for the software requirements specification process, while Organization A is responsible for the design, coding, testing, and deployment processes. Furthermore, Organization B is responsible for the communication between Organization A and the project client. Both organizations A and B work in a traditional but many times *ad hoc* manners. This way of working has brought problems, such as budget overloading, teams divided into disciplines (testers, architects, programmers, etc.) causing many intermediary delivery points in the organization and increasing delays between them, with large periods of time required to deploy new versions of the software products; not unlike what is described in the

[6] A research method in which the researcher acts in the phenomenon being observed (BASKERVILLE, 1997).

literature (WILLIAMS; COCKBURN, 2003; OLSSON; ALAHYARI; BOSCH, 2012; KARVONEN et al., 2015).

4.3.1 Theoretical Background

In this study, we combined *Systems Theory tools* (MEADOWS, 2008), *GUT Matrix* (BECK, 1999), and *Reference Ontologies* (GUIZZARDI, 2007) into a process that guides identifying strategies to implement CSE practices. The System Theory tools allow seeing the organization as a system, consisting of elements (e.g., teams, artifacts, policies) and interconnections (e.g., the relation between the development team, the software artifacts it produces, and the policies that influence their production) coherently organized in a structure that produces a characteristic set of behaviors, often classified as its function or purpose (e.g., the development team produces a software product aiming to accomplish its function in the organization) (MEADOWS, 2008).

In the Systems Theory literature, there are several tools that support understanding the different elements and behaviors of a system, such as systemic maps and archetypes (MEADOWS, 2008; STERMAN, 2010). A systemic map (also known as a causal loop diagram) allows representing the dynamics of a system by means of the system borders, relevant variables, their causal relationships, and feedback loops. A positive causal relationship means that two variables change in the same direction (e.g., an increase in the number of bad design decisions causes increases in the number of software defects), while a negative causal relationship means that two variables change in opposite directions (e.g., an increase in test efficacy causes decreases in the number of software defects). Feedback loops are mechanisms that change the variables of the system. There are two main types: balancing and reinforcing feedback loops. The former is an equilibrant structure in the system and is source of stability and resistance to change. The latter compounds change in one direction with even more change.

One beneficial effect of using systemic maps is that they help identify archetypes. An archetype is a common structure of the system that produces a characteristic pattern of behavior. For example, the archetype *Shifting the Burden* occurs when a problem symptom is solved by applying a symptomatic solution, which diverts attention away from a more fundamental solution (KIM, 1993). Archetypes and Systemic Maps can be useful to identify problems and possible leverage points to solve them. Leverage points are points in the system where a small change can lead to a large shift in behavior (MEADOWS, 2008), as shown in Figure 49.

GUT Matrix allows to prioritize the resolution of problems, considering that resources are limited to solve them. The prioritization is based on: Gravity (G), which describes the impact of the problem on the organization; Urgency (U); referring to how much time is available to address the problem; and Tendency (T), which measures the predisposition of a problem getting worse over time (BECK, 1999).

Reference Ontology is a special kind of conceptual model representing a model of

consensus within a community. It is a solution-independent specification with the aim of making a clear and precise description of the domain in reality for the purposes of communication, learning, and problem-solving (BASKERVILLE, 1997). Thus, SEON (RUY et al., 2016) was used to provide a common conceptualization to support communication among organizations A and B and their employees in the software development context.

Next, we present general information about the study. Detailed information is available in (SANTOS; BARCELLOS; CALHAU, 2020; SANTOS; BARCELLOS; CALHAU, 2022).

4.3.2 Execution and Results

The study was performed at the beginning of 2020 and involved five participants from Organization A (two directors, one tech lead, and two developers) and two consultants, being the doctoral candidate one of them. The main goal was to improve software processes by implementing agile practices (i.e., StH first stage) in a proper and suitable way for Organization A. We started by collecting data about the organization through interviews. Then, we built systemic maps to get a comprehensive view of the organization and understand how it behaves, representing relevant variables and the relation between them (e.g., cause and effect relationship). Figure 49 illustrates a fragment of the built systemic maps. The elements in blue in the figure form a modeling pattern that reveals the presence of the archetype *Shifting the Burden*.

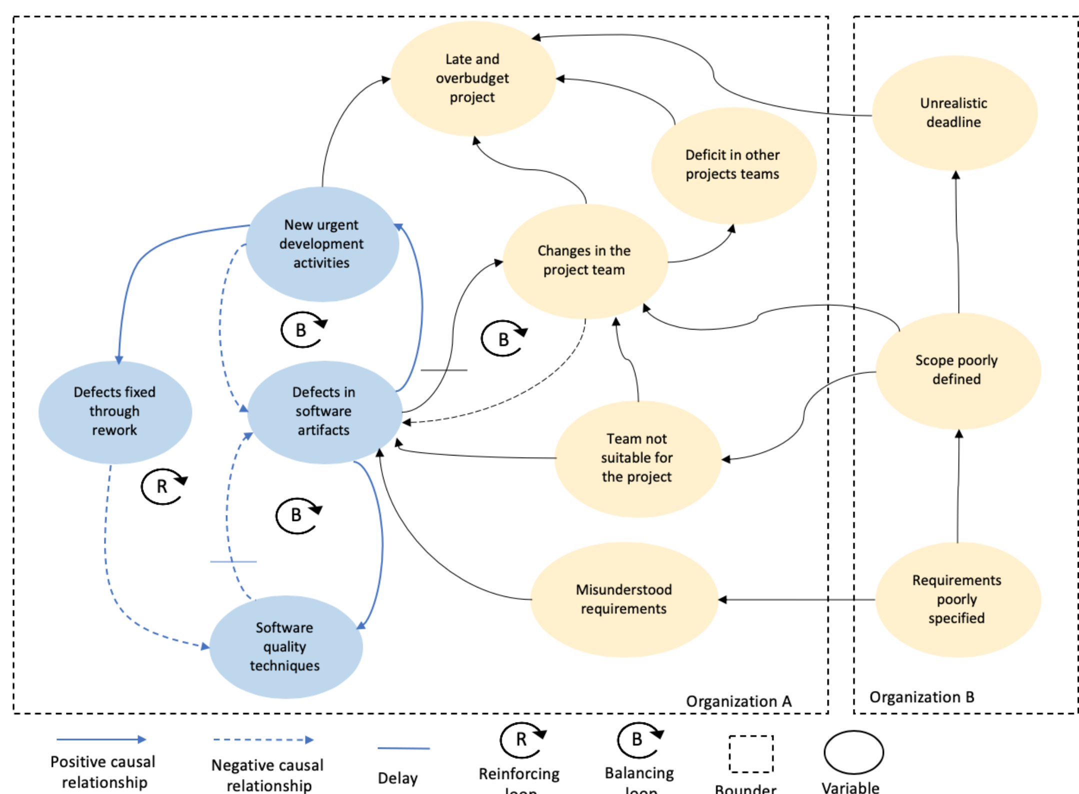


Figure 49 – Fragment of Systemic map (SANTOS; BARCELLOS; CALHAU, 2020).

As previously said, Organization B is responsible for eliciting requirements with the client, specifying and sending them for Organization A to develop the software. The development teams of Organization A often misunderstand requirements that describe the software, component, or functionality to be developed, since Organization B produces *Requirements poorly specified*, neither adopting a technique nor following a pattern to describe them. *Misunderstood requirements* contribute to increasing the number of *Defects in software artifacts*, since design, code, and test are produced based on the requirements informed by Organization B. *Defects in software artifacts* make Organization A mobilize (and often overload) the development team to fix defects by performing *New urgent development activities*, which decreases the number of *Defects in software artifacts*. These urgent activities are performed as fast as possible, aiming not to delay other activities. Thus, they do not properly follow software quality good practices. Moreover, they contribute to increasing the project cost and time (*Late and over-budget project*). *Defects in software artifacts* increase the need of using Software quality techniques that, when used, lead to fewer Defects in software artifacts. This causal relationship has a delay since the effect of using *Software quality techniques* can take a while to be perceived.

As shown in Figure 49, the archetype *Shifting the Burden* is composed of two balancing feedback loops and one reinforcing feedback loop. The balancing feedback loops (between *New urgent development activities* and *Defects in software artifacts*, and between *Defects in software artifacts* and *Software quality techniques*) mean that the involved variables influence each other in a balanced and stable way (e.g., higher/lower the number of *Defects in software artifacts*, more/less *New urgent development activities* are performed). In the reinforcing feedback loop, *New urgent development activities* are a symptomatic solution that leads to *Defects fixed* through rework, a side effect, because once urgent development activities fix the defects in software artifacts, Organization A feels like the problem was solved. This, in turn, decreases the need for using Software quality techniques, which is a more fundamental solution. As a result, software artifacts continue to be produced with defects, overloading the development team with new urgent development activities. *Shifting the Burden* is a complex behavior structure because the balancing and reinforcing loops move the system (Organization A) in a direction (*New urgent development activities*) usually other than the one desired (Software quality techniques). *New urgent development activities* contribute to increasing project cost and time (*Project is late and over-budget*) because these activities were not initially planned in the project.

When Organization B does not properly define the project scope (*Scope poorly defined*), Organization A may allocate a Team not suitable for the project, contributing to *Defects in software artifacts* and to *Changes in the project team* during the project. Usually, when the team is changed, the new members need to get knowledge about the project. Moreover, often the new members are more experienced and thus more expensive, which contributes to *Late and over-budget project*. To change the project team, members can be moved from a project to another, causing *Deficit in other project teams*. Furthermore, there is a balancing loop between *Changes in the project team* and *Defects in software artifacts*. The former may cause the latter

due to instability inserted into the team. The latter, in turn, contributes to the former because *Defects in software artifacts* may lead to the need to change the team. There is a delay in this relationship because it can take a while between defects are noticed and the need to change the team. Finally, *Scope poorly defined* causes *Unrealistic deadline*, which contributes to *Late and over-budget project*.

We reflected on the behaviors on which the strategies to implement agile practices should be focused. Then, we created a GUT matrix to identify and prioritize non-fruitful behaviors, i.e., undesirable behaviors. They were identified mainly from the systemic maps. For example, from the fragment depicted in Figure 49 based on the positive causal relationship between *Misunderstood requirements* and *Defects in software artifacts*, the following undesirable behavior was identified: *Software artifacts are developed based on misunderstood requirements*. From the *Shifting the Burden* archetype, we identified: *Software quality techniques are not often applied to build software artifacts*. To complement the information provided by the systemic maps, we used information from the interviews to look for behaviors the literature points out as desirable in organizations moving to agile (e.g., self-organized teams) (DEAN, 2016).

After identifying the undesirable behaviors, the study participants validated and prioritized them considering the GUT dimensions. Each dimension was evaluated considering values from 1 (very low) to 5 (very high). 13 undesirable behaviors were identified. Table 7 shows a fragment of the GUT Matrix.

Table 7 – Fragment of GUT Matrix.

#	**Undesirable Behaviors**	**G**	**U**	**T**	**GxUxT**
UB1	Software artifacts are developed based on misunderstood requirements	5	5	5	125
UB2	Software quality techniques are not often applied to build software artifacts	5	5	4	100
UB3	Projects are late and over budget	5	5	4	100
UB4	Organization has inconsistent knowledge of agile methods	5	5	4	100
UB5	Teams are not self-organized	5	4	4	80

For each undesirable behavior, we analyzed the systemic maps and the interviews and identified its causes. We have observed that *Software artifacts are developed based on misunderstood requirements* (UB1) **because** *Requirements are not satisfactorily described* (C1), and **because of** *Poor communication between client and development team* (C2). C1 was identified directly from the systemic map. C2 was based on information about the procedure followed by Organization A to communicate with the client. When there is any doubt about requirements, the contact was made mainly through email or comments on issues in the project management system. Only Organization B has direct contact with the client.

C1 and C2 are also causes of UB2 (*Software quality techniques are not often applied to build software artifacts*), since the lack of well-defined requirements and direct contact with the client impact verification and validation activities. Moreover, there is a *Lack of clear and*

objective criteria to evaluate results (C3) and there are *Large deliverables* (C4), which makes it difficult to evaluate results.

As it can be noticed in Figure 49, *Projects are late and over budget* (UB3) mainly because of C1 and C5 (*Unstable scope and deadline*). Moreover, C6 (*Unsuitable team allocation*) and C4 also affect project cost and time. The former is because low productivity impacts on project time and, thus, cost. The latter is because it is difficult to estimate large projects.

Regarding UB4 (*Organization has inconsistent knowledge of agile methods*), some members of the organization had previous experience with agile methods in other companies, others had a previous unsuccessful experience in Organization A, and others did not have experienced agile methods. Most of the members were not sure about agile concepts and practices. Therefore, this undesirable behavior is caused by C7 (*Organization's members had different experiences with agile*) and C8 (*Agile concepts and practices are not well-known by the organization*). Finally, *Teams are not self-organized* (UB5) due to the *Traditional development culture that produced functional and hierarchical teams* (C9). After we have identified the causes of undesirable behaviors, the study participants validated them. Table 8 shows the identified causes and respective undesirable behaviors.

Table 8 – Causes of Undesirable Behaviors.

#	Causes	UB1	UB2	UB3	UB4	UB5
C1	Requirements are not satisfactorily described	x	x	x	-	-
C2	Poor communication be-tween client and development team	x	x	-	-	-
C3	Lack of clear and objective criteria to evaluate results	-	x	-	-	-
C4	Large deliverables	-	x	x	-	-
C5	Unstable scope and deadline	-	x	x	-	-
C6	Unstable team allocation	-	-	x	-	-
C7	Organization's members had different experiences with agile	-	-	-	x	-
C8	Agile concepts and practices are not well-known by the organization	-	-	-	x	-
C9	Traditional development culture	-	-	-	-	x

The causes of undesirable behaviors and the prioritization made in the GUT Matrix showed us leverage points of the system, i.e., points that if changed could change the system behavior. Therefore, we defined strategies to help Organization A move towards the second stage of StH by changing leverage points of the system and thus creating new behaviors in the system in that direction. We started by defining strategies to change undesirable behaviors at the top of the GUT Matrix and causes related to more than one undesirable behavior. After we have defined the strategies, we presented them to the team in a meeting and they provided feedback that help us to make the strategies more suitable for the organization.

Table 9 summarizes the defined strategies, the leverage points (causes) addressed by them, and the main agile concepts involved. It is worth noticing that some agile concepts were indirectly addressed. For example, although we did not directly use Product Backlog in S1, the

set of requirements agreed with Organization B works as such. Similarly, in S3, when the team selects the requirements to be addressed in a development cycle, we are applying the Sprint Backlog notion. We decided not to use some of the original terms because Organization A had a bad previous experience trying to implement agile practices by following Scrum "by the book", which did not work and provoked resistance to certain practices. Thus, we tried to give some flexibility even to the practices' names, to avoid bad links with the previous experience. Details about the implementation of strategies in (SANTOS; BARCELLOS; CALHAU, 2020; SANTOS; BARCELLOS; CALHAU, 2022).

Table 9 – Strategies, Causes, and Agile Concepts.

#	Strategies	Agile Concepts	Causes
S1	New procedure to communicate requirements	User Story, Behavior Driven Development, Product Owner and Product Backlog	C1, C2, C3
S2	Budget and time globally and locally managed through short development cycles	Sprint, Sprint Backlog, Scrum meetings, and Small deliverables	C4, C5, C6, C9
S3	Self-organized teams	Squad and Guild	C9
S4	Agile common conceptualization	Concepts related to agile software development	C7,C8

To address C7 and C8, which cause the organization to have inconsistent knowledge of agile methods (UB4), we defined *Agile common conceptualization* (S4) as a strategy to use *Reference Ontologies* to provide a common conceptualization about the Software Engineering domain as a whole, and about the agile development process in particular. We used ontologies from SEON (RUY et al., 2016) to extract the view relevant to understand agile development. It contains a conceptual model fragment, axioms, and textual descriptions that provide an integrated view of agile and traditional development, defining concepts in a clear, objective, and unambiguous way. We suggested the use of SEON because its ontologies have been developed based on the literature and several standards, providing a consensual conceptualization. The SEON view used in the study focuses on the Scrum Reference Ontology (SRO) and can be seen in Section 3.2. To make it easier for the teams to learn and apply the conceptualization provided by an ontology, the authors created complementary artifacts that combined graphical (e.g., Kanban Board, Diagrams, and process model) and textual elements (e.g., textual definitions). Table 10 presents a dictionary of terms that was used to explain Scrum's concept. The details about other artifacts are available in (SANTOS; BARCELLOS; CALHAU, 2022). As a result of this approach, the conceptualization provided by the ontologies was made more accessible to the team, improving domain understanding and communication.

Table 10 – Some terms present in Dictionary based on SRO.

#	Concept	Description
1	Scrum Project	Software Project that adopts Scrum in its process.

Table 10 – *Continued from previous page*

#	Strategies	Agile Concepts
2	User Story	Requirement Artifact (i.e., a requirement recorded in some way) that describes Requirements in a Scrum Project. It indicates a goal that the user expects to achieve by using the system and, thus, represents value for the client. A User Story can be an Atomic User Story, when it is not decomposed into others, or an Epic, when it is composed of other Use Stories.
3	Acceptance Criteria	Requirement established to a User Story and that must be met when the User Story is materialized. Thus, it is used to verify if the User Story was developed correctly and meets the client needs.
4	Deliverable	Software Item that materializes User Stories.

After defining and validating the strategies with the team, they were executed by the organization in two projects with the supervision of the first and third authors of (SANTOS; BARCELLOS; CALHAU, 2020; SANTOS; BARCELLOS; CALHAU, 2022). The first project started and finished during this study while the second project started before the study and was still ongoing at the time we wrote in (SANTOS; BARCELLOS; CALHAU, 2020). The new practices started to be used in early February 2020.

The authors define some information needs (e.g., *Effort spent on development and bug-fixing tasks in different projects* and *Effort spent on development and bug-fixing tasks before and after applying the strategies in the project*) to monitor the implementation of strategies. Data to meet the information needs was extracted from Jira[7], which is used by Organization A to support part of the software development process. Considering that the strategies were applied in the projects in different moments (the first project adopted the strategies from its beginning to its end, while the second adopted the strategies when it was already ongoing), we decided to analyze them separately. About four months later, we conducted an interview to obtain feedback and collect project data. At that point, one of the projects had already been concluded and the other was ongoing.

Finally, we defined and executed strategies to implement agile practices taking the prioritized undesirable behaviors as leverage points and considering the organization's needs. Figure 50 shows an overview of *California*, the System-Thinking process raised from this case study.

After executing the process, we carried out interviews with the participants to get their perceptions about it. The results showed that *California* provided means to:

i. **Understand how different organizational aspects (e.g., business rules and quality software practices) are interrelated and influence each other and how these aspects and interrelations produce desirable and undesirable behaviors**: The

[7] <https://www.atlassian.com/software/jira>

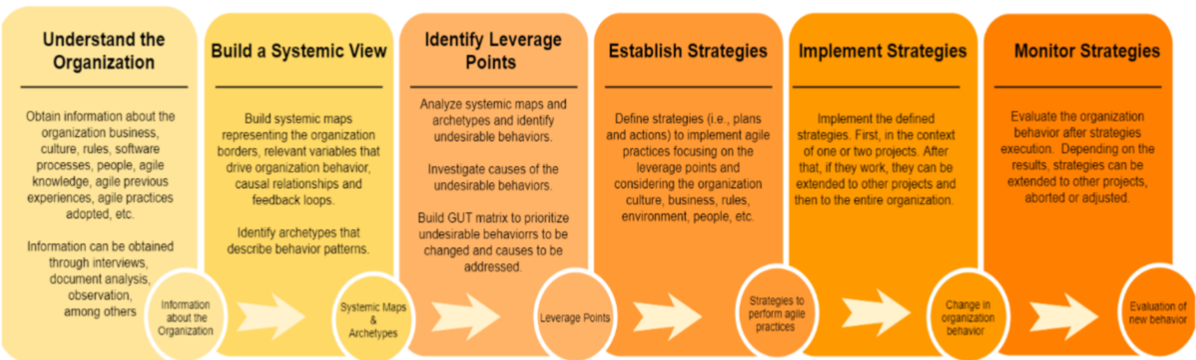


Figure 50 – California: A System Theory-Based Process (SANTOS; BARCELLOS; CALHAU, 2020).

director said that *“the systemic maps allowed me to understand how poorly specified requirements can negatively impact different parts of the project and of the organization”*;

ii. **Create strategies to change undesirable behaviors since it provided a comprehensive understanding of the organization’s behavior and supported identifying causes of undesirable behaviors**: The director said that he *“perceived the need to implement practices to guarantee the quality of the requirements and that development tasks should only start if the developer truly understood the requirement”*;

iii. **Prioritize the undesirable behaviors to be changed first**: the director stated that *GUT Matrix* was easy to use, and important to prioritize the undesirable behaviors to be changed first. According to him, using these tools *“was easier and clearer when compared to Ishikawa and Pareto diagrams, because systemic maps allow more comprehensive and freer views and GUT Matrix has a simple way of prioritization”*;

iv. **Create a common communication among project stakeholders and business partners, eliminating some misunderstandings not only about agile practices but also about Software Engineering in general, reducing misunderstandings in software requirements among the stakeholders and enabling better managing of budget and time locally, in short, development cycles**: the director reported the artifacts created based on reference ontologies were useful to create a common communication. He said that *“by using the conceptualization provided by the ontology, the team truly understood the ‘done concept’, commonly used in agile projects”*, while the tech leader commented that *“by using the artifacts created by the ontology conceptualization, the necessary information that should be included in a requirement description became clearer”*, and;

v. **Address some undesirable behaviors by applying agile practices and concepts**: the tech leader and directors commented that *“the self-organization culture has been developed in the teams and that the use of Squads has been very helpful”*. The use of Guilds was still in progress.

Table 11 – Effort spent on development and bug-fixing tasks in different projects.

Task	Project that adopted the strategies	Projects that did not adopt the strategies
Development	97.6%	81.0%
Bug-fixing	2.4%	18.9%

Table 12 – Effort spent on development and bug-fixing tasks before and after applying the strategies in the project.

Task	Before the strategies	After the strategies
Development	62,1%	88,2%
Bug-fixing	37,9%	11,8%

Aiming to meet the information needs, collected data from the two projects where the strategies were implemented (one finished and another ongoing) and from the other 22 projects that did not use the strategies and were carried out in the same time-box of our study. Table 11 and Table 12 show the results.

By comparing data regarding the tasks performed in the first project and in the other 22 projects, we have an indication that in the former only 2.4% of the effort was spent on bug-fixing while in the latter it was 18.9%, as shown in Table 11. Analyzing the two projects in which the strategies were implemented, we also have an indication of an improvement in the effort spent on bug-fixing before using the strategies (37.8%) to after using them (11.8%), as shown in Table 12.

In summary, the results showed that *California* helps understand how the organization behaves and defines proper strategies to implement or improve CSE practices according to the organization's behavior and needs.

4.3.3 Threats to Validity to the Study Results

The validity of a study denotes the trustworthiness of the results. Every study has threats that should be addressed as much as possible and considered together with the results. In this section, we discuss some threats considering the classification proposed in (RUNESON et al., 2012).

The main threat in this study is related to the researchers who conducted the study. Participative case studies are biased and subjective as their results rely on the researchers (BASKERVILLE, 1997). The first and third authors acted as consultants in Organization A and were responsible for conducting the interviews, creating systemic maps and GUT matrix and defining strategies. The researchers participation affects *Internal Validity*, which is concerned with the relationship between results and the applied treatment, *External Validity*, which regards to what extent it is possible to generalize the results from the case-specific findings to different

cases, and *Reliability Validity*, which refers to what extent data and analysis depend on specific researchers. To reduce the threat, the other study participants participated in the activities and validated results. Moreover, another researcher, external to the organization, evaluated data collection and analysis and was involved in discussing and reflecting on the study and results.

Concerning *Construct Validity*, which is related to the constructs involved in the study, the main threat is that we did not define indicators to evaluate results. Data collection was performed through interviews, which involves subjectivity. To minimize this threat, we used some measures collected in the projects to evaluate the new behaviors caused by the proposed strategies. Moreover, the use of maps facilitated knowledge sharing and the validation of their usefulness by the participants contributes to construct validity.

In case-based research, after getting results from specific case studies, generalization can be established for similar cases. However, the threats aforementioned constrain generalization. Moreover, the study involved only one organization. Thus, it is not possible to generalize results for cases without researcher intervention or for organizations not similar to Organization A.

4.3.4 What did we learn?

With this study, we learned that using *System-Thinking tools*, *GUT Matrix*, and *reference ontologies* helps understand the organization's current state and defining strategies for improving it. We also observed that it is useful to have a process with well-defined steps to be followed. Moreover, we confirmed the perception we had in the first LI that *understanding the organization helps identifying information needs*. In this study, we noticed that *when applying California, information needs can be derived to understand the organization's current state/behavior (e.g., How much time has been spent on rework? How many defects have been delivered to the client? How much rework has been caused by requirements changes? How many projects have been late? How many projects have been over-budget? What has been the turnover rate?)*. Data meeting such information needs can be used to support decisions on the strategies to be implemented. After strategies implementation, information needs can be defined aiming to verify whether the defined strategies have been effective (e.g., *Did rework decrease after using the strategy? How much? Are the team members more engaged?*).

We also observed a strong relation between the information needs and the applications used by the organization (we indeed expected that). When information needs are identified, the application(s) or repository(ies) containing the corresponding data are often also identified. That means, the processes of deriving information needs and identifying the ones feasible to be met by available data are not necessarily sequential; they can occur together and iteratively. For example, in the study, to evaluate the effectiveness of the strategy established to improve requirements communication aiming at avoiding rework due to requirements misunderstanding, we defined the information need '*How much effort has been spent on bug-fixing activities?*', which demanded data from Jira. The results of the data analysis to evaluate the strategy effectiveness

were presented in the last paragraph of Section 4.3.2.

We also learned that, *although the proposed process (California) is useful, it involves a lot of tacit knowledge and judgment, as well as knowledge of System Thinking tools, GUT matrix, and Reference ontologies. Moreover, it may demand much time to be applied.* Hence, depending on the application scenario to be considered, it may be difficult or even unfeasible to use *California*. This perception led us to perform a new learning iteration to answer the same question explored in this study, but focusing in an easier way to support understanding how an organization adopts CSE practices (e.g., practices and applications).

4.4 Third Learning Iteration: Zeppelin

Although *California* can be useful in this matter, it may demand knowledge and time to be applied. Thus, we performed the third learning iteration to increase the knowledge related to the question: *How to understand the way an organization works and, thus, help identify its information needs relevant to data-driven software development in CSE context?*

The learning iteration comprised a *multiple case study*[8] and intended to reach an easier way to understand the current state of software organizations concerning CSE practices adoption.

The study consisted in developing a diagnostic instrument (called *Zeppelin*) to support organizations to get a panoramic view of CSE adoption and using such instrument in five Brazilian software organizations (one software house, two startups, one fin-tech, and one public Organization with an IT Department). General information about the study is presented in the following. Details are available in (SANTOS; BARCELLOS; RUY, 2021).

4.4.1 Execution and Results

The first step of the study was to create *Zeppelin*. It has two components: (i) *Diagnosis Questionnaire*, which identifies the CSE practices an organization performs and the degree to which they are adopted; and (ii) *Analytics Report*, which presents consolidated data from the questionnaire answers, showing a panoramic view of the organization from the CSE perspective and pointing out possible improvement areas.

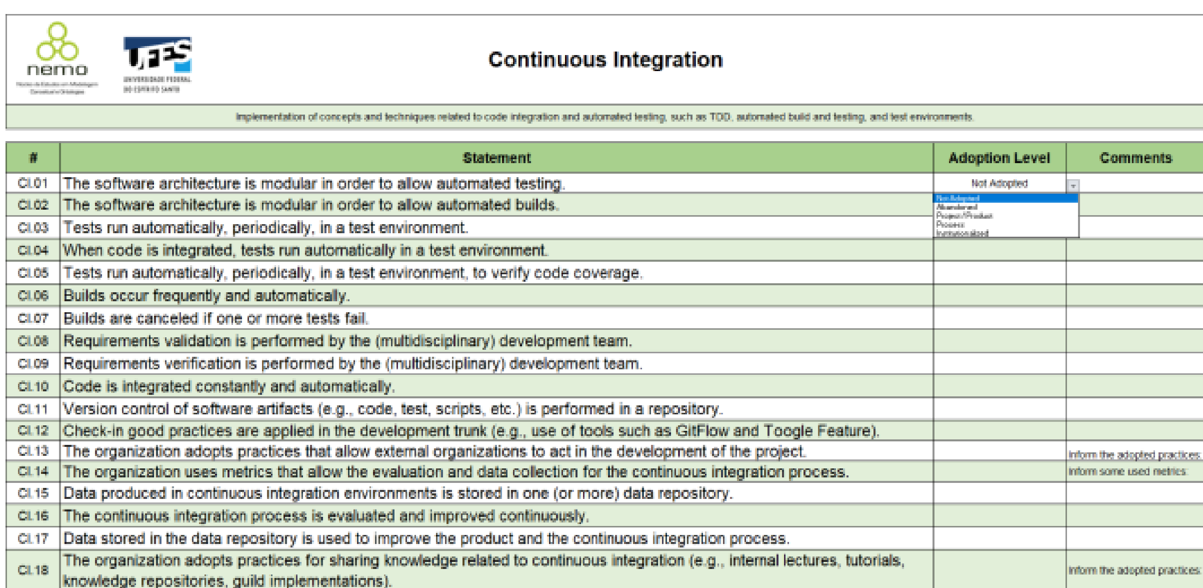
The *Diagnosis Questionnaire* consists of an electronic spreadsheet with eight forms: (i) Context, to provide a brief introduction to CSE; (ii) Instructions, which guides the user on how to fill in the other forms; (iii) Organization, to characterize the organization (e.g., organization type, size, age, development team size); (iv) User, to characterize the person answering the questionnaire on behalf of the organization (e.g., position, knowledge, and experience with CSE

[8] Multiple case study is a research method in which the researcher carries out multiple cases to investigate a phenomenon through the understanding of differences and similarities between the cases (GUSTAFSSON, 2017).

practices); and four forms concerning StH stages and containing in all 76 statements expressing CSE practices: (v) Agile Organization (26 practices), (vi) Continuous Integration (18 practices), (vii) Continuous Deployment (19 practices), and (viii) R&D as Innovation System (13 practices). The CSE practices were identified based on the literature, mainly StH (OLSSON; ALAHYARI; BOSCH, 2012), Continuous * (FITZGERALD; STOL, 2017), Eye of CSE (JOHANSSEN et al., 2019) and FCSE (BARCELLOS, 2020), and on the practical experience reported in (SANTOS; BARCELLOS; RUY, 2021).

The questionnaire is used to evaluate which practices have been adopted in the organization and to understand how comprehensive their adoption has been. When applying *Zeppelin*, for each practice, the user must indicate the level at which it is adopted in the organization. The adoption levels were defined based on (OLSSON; ALAHYARI; BOSCH, 2012) and are used to capture the comprehensiveness of each practice in the organization and help monitor its evolution. The *Not Adopted* level is used to identify practices that the organization has never used. The *Abandoned* level refers to practices that were discontinued. The *Project/Product* level is used to identify practices not formalized in the organization and used only in a particular project or product. The *Process* level indicates that the practice is formally defined (e.g., by means of procedures, guidelines, business processes, policies) but the team can decide whether to apply it in a project. Finally, a CSE practice is said to have the *Institutionalized* level when it is formally defined and used in all projects. The adoption degree of each stage (*AD*) is represented as a percentage and is established by calculating the weighted average of the adoption level (*AL*) of all practices of that stage (i.e., practices 1 to *n*, where *n* is the number of practices related to the stage). Thus, $AD_{stage} = (weight \times AL_{practice1} + ... + weight \times AL_{practicen})/n) \times 100)$. The weights of the adoption levels vary from 0 (zero) (referring to the *Not Adopted level*) to 1.0 (referring to the *Institutionalized level*). Figure 51 presents a fragment of the *Diagnosis Questionnaire*. The complete questionnaire is available at <http://nemo.inf.ufes.br/wp-content/uploads/CSE/zeppelin_analytics_report_v1.xlsx> and Appendix A.

The *Analytics Report* is an artifact that imports data from the *Diagnosis Questionnaire*, consolidates it and provides a panoramic view (by using tables, charts, and text) of CSE practices adoption in the organization. It focuses on providing three different perspectives of CSE practices adoption: (i) per 5 StH stage (i.e., Agile Organization, Continuous Integration, Continuous Deployment, and R&D as Innovation System); (ii) per 9 Eye of CSE dimension (namely, Deployment, Quality, Software Management, Team, Technical Solution, Knowledge, Operation, Business, and User/Customer), and (iii) per Eye of CSE element (33 elements, e.g., Agile Practice, Automated Tests, Continuous Deployment Releases, and Continuous Learning). Each CSE element is related to one CSE dimension. By analyzing the different perspectives, the organization identifies its strengths and weaknesses and can define improvement actions accordingly. Figure 52 illustrates some pieces of information contained in the *Analytic Report*.



Continuous Integration

Implementation of concepts and techniques related to code integration and automated testing, such as TDD, automated build and testing, and test environments.

#	Statement	Adoption Level	Comments
CI.01	The software architecture is modular in order to allow automated testing.	Not Adopted	
CI.02	The software architecture is modular in order to allow automated builds.		
CI.03	Tests run automatically, periodically, in a test environment.		
CI.04	When code is integrated, tests run automatically in a test environment.		
CI.05	Tests run automatically, periodically, in a test environment, to verify code coverage.		
CI.06	Builds occur frequently and automatically.		
CI.07	Builds are canceled if one or more tests fail.		
CI.08	Requirements validation is performed by the (multidisciplinary) development team.		
CI.09	Requirements verification is performed by the (multidisciplinary) development team.		
CI.10	Code is integrated constantly and automatically.		
CI.11	Version control of software artifacts (e.g., code, test, scripts, etc.) is performed in a repository.		
CI.12	Check-in good practices are applied in the development trunk (e.g., use of tools such as GitFlow and Toogle Feature).		
CI.13	The organization adopts practices that allow external organizations to act in the development of the project.		Inform the adopted practices
CI.14	The organization uses metrics that allow the evaluation and data collection for the continuous integration process.		Inform some used metrics
CI.15	Data produced in continuous integration environments is stored in one (or more) data repository.		
CI.16	The continuous integration process is evaluated and improved continuously.		
CI.17	Data stored in the data repository is used to improve the product and the continuous integration process.		
CI.18	The organization adopts practices for sharing knowledge related to continuous integration (e.g., internal lectures, tutorials, knowledge repositories, guild implementations).		Inform the adopted practices

Figure 51 – Fragment of the Diagnosis Questionnaire with practices related to Continuous Integration (SANTOS; BARCELLOS; RUY, 2021).

Dimension	Degree of Adoption				
	Stages of StH				Organization
	AO	CI	CD	IS	
Development	69%	58%	75%		67%
Quality	50%	65%	75%		64%
Sw Management	73%	63%	69%		70%
Team	88%			75%	81%
Technical Solution	50%	75%			67%
Knowledge	50%	50%	50%	58%	53%
Operation				88%	88%
Business	100%		60%	92%	75%
User/Customer			94%	70%	84%
Degree of Adoption	69%	64%	70%	76%	69%

Adoption Level	Number of CSE Practices				
	Stages of StH				Organization
	AO	CI	CD	IS	
Not Adopted			2	1	3
Abandoned				1	1
Project/Product	11	9	3	1	24
Process	10	8	9	3	30
Institutionalized	5	1	5	7	18
Degree of Adoption	69%	64%	70%	76%	69%

Number of Adopted Practices		
Adoption Level	**Practices**	**(%)**
Not Adopted	3	4%
Abandoned	1	1%
Project / Product	24	32%
Process	30	39%
Institutionalized	18	24%
Adopted Practices	**72**	**95%**
Total	**76**	**100%**

Degree of Adoption per Dimension

Figure 52 – Fragment of Analytic Report (SANTOS; BARCELLOS; RUY, 2021).

Zeppelin was applied in five Brazilian software organizations: two startups, one software house, one fin-tech, and one public organization with IT department. After using *Zeppelin*, the participants (one representing each organization) were asked to share their perceptions about the use of *Zeppelin* by filling in a feedback form. Concerning (Q1) *Is Zeppelin useful to identify which CSE practices have been adopted in an organization and provide a panorama about its position in the CSE evolutionary path?*. We analyzed the answers to the questions of the first section of the feedback form (SANTOS; BARCELLOS; RUY, 2021). As result, all participants

agreed (80% strongly agreed and 20% agreed) that *Zeppelin* provided a comprehensive view of CSE practices adoption and provided a faithful panoramic view of their organization.

As for (Q2) *Does Zeppelin help an organization envision an improvement path to follow*? All participants agreed (40% strongly agreed and 60% agreed) that *Zeppelin* supported them to identify areas and practices the organization succeeds and the ones that need to be improved or adopted. They also agreed that the panorama provided by *Zeppelin* and the practices contained in the *Diagnosis Questionnaire* helps define improvement actions.

When asked to provide comments and suggestions about *Zeppelin*, a participant made a comment that supports the results related to Q2: "*This evaluation instrument can strategically support the company to understand its current state and envision its future state*".

We asked the participants about dimensions they consider most important to improve in their organizations. They noticed that the answers were aligned to data provided in the *Analytics Report.* Some of the aspects cited by the participants were Knowledge, Software Management, Quality, User/Costumer, Team, Operation, and Business. In general, the participants found it important to improve CSE practices that promote:

i. Knowledge sharing and decision rationale capturing at different stages (Knowledge);

ii. Adoption of good management practices to improve agile development, continuous integration, and deployment (Software Management);

iii. Improvement and automation in tests in different stages of software development (Quality);

iv. Involvement of user and other stakeholders in all development processes and learning from user data and feedback about the products (User/Costumer);

v. Self-organized, motivated and productive team (Team);

vi. Logging and monitoring production activities; and

vii. Alignment between software development, operation, and business.

It was observed that some dimensions less addressed by the organizations were not cited by the participants, even being pointed out in the Analytics Report. This, in fact, was expected, because *Zeppelin* gives a broad view of the organization, indicating aspects more and less addressed. However, which aspects will be the target of improvement actions and how much they need to be improved is a decision to be made by the organization, based on its goals, needs, characteristics, constraints, etc.

Therefore, the results indicate that *Zeppelin* provides a panoramic view that describes the current state of the adopted CSE practices in an organization, supporting the identification

of weaknesses and strengths as well as aiding in decision-making about which aspects should be addressed in improvement actions.

4.4.2 Threats to validity to study results

The validity of a study denotes the trustworthiness of the results. Every study has threats that should be addressed as much as possible and considered together with the results. In this section, we discuss some threats considering the classification proposed in (RUNESON et al., 2012).

Regarding *Construct Validity*, which is related to the constructs involved in the study, the main threat concerns the statements used to identify CSE practices in the Diagnosis Questionnaire, which can be understood in different ways by different participants. To minimize this threat, the authors performed interviews with the participants to validate the answers. This gave us an opportunity to resolve misunderstandings. Another threat refers to the weights assigned by the authors to the adoption levels (e.g., a practice adopted at project level has weight 0.5 while an institutionalized practice has weight 1.0). This directly impacts the degree of adoption calculation. If different weights are used, the quantitative results presented in the Analytics Report may be a little different. CSE practices defined in the Diagnosis Questionnaire is also a threat. Some practices may have not been properly covered by it. To minimize this threat, CSE practices were defined based on four works addressing CSE processes and practices (OLSSON; ALAHYARI; BOSCH, 2012; JOHANSSEN et al., 2019; FITZGERALD; STOL, 2017; BARCELLOS et al., 2022).

Concerning *Internal Validity*, which is concerned with the relationship between results and the applied treatment, the main threat is related to the researchers who conducted the study. In (SANTOS; BARCELLOS; RUY, 2021), two of the authors conducted interviews to validate data . Moreover, the Analytics Reports were also elaborated by the authors. Different results could be obtained if *Zeppelin* had been used by the participants without the authors interference. To minimize this threat, the authors interfered as little as possible and did not influence the participants' feedback. Another threat refers to the participants providing answers not consistent with the organization reality (e.g., they could indicate a higher level of adoption for a practice than its actual level). We minimized this threat by performing interviews to validate the answers. In the interviews, we asked the participants to explain how each practice is performed in the organization, so that we could verify whether the indicated level of adoption was correct.

As for *External Validity*, which is concerned the extent to which it is possible to generalize results, the main threats in this study are: (i) researchers participation; (ii) small number of organizations; and (iii) feedback obtained from only one person of each organization. Concerning (i), as discussed in the context of internal validity, the researchers participation may have influenced results. As for (ii), only five organizations were involved in the study, all of them are

from the same country and most of them are small and founded in the last years. Regarding (iii), the results are based on the participants' feedback and, thus, are biased and subjective (BASKERVILLE, 1997). Thus, it is not possible to generalize results for cases different from the ones considered in the study.

Finally, concerning *Reliability Validity*, which refers to what extent data and analysis depend on specific researcher, the main threat is that data analysis was performed by the authors, as presented in (SANTOS; BARCELLOS; RUY, 2021). To minimize this threat, analysis was carried out by two of the authors and reviewed by the other one.

In summary, considering all mentioned threats, we can only present some insights regarding *Zeppelin* use and generalization is limited. Thus, obtained results cannot be considered conclusive, but preliminary evidence of *Zeppelin* feasibility and usefulness.

4.4.3 What did we learn?

With this study, we learned that *Zeppelin* provides, in an easy way, a panoramic view of CSE practices implemented in an organization. Thus, using *Zeppelin* helps us to know the organization's current state by means of the CSE practices it adopts and the adoption comprehensiveness. This information can be used to identify areas that can be improved and actions that can be performed aiming at such improvements.

It was observed that *by knowing the CSE practices the organization adopts, it is possible to identify relevant data, from practices and applications, that meets an information need*. For example, if the organization adopts automatic testing practices, it has data about them storage in a continuous integration server. Thus, the data about the execution of automated tests can be used to answer an information need related to them can be defined (e.g., *What is the automated test coverage? What is the test automatic efficiency? What is the density of detected defects in automated tests?*). On the other hand, it is not reasonable to derive information needs related to practices that the organization does not perform, because there no data is produced to meet such needs. For example, if the organization does not perform continuous experimentation practices (e.g., A/B testing), there may not be data or information needs associated to them.

It was also noticed that *information needs can be identified from the improvement areas pointed out in the diagnostic*. For example, if it is necessary to improve continuous deployment, information needs such as *What is the deployment rate? What is the deployment cost?* could be defined to quantitatively understand the current state and, later, verify the effects of the performed improvement actions.

Finally, it was learned that *knowing the level at which the practices are adopted also helps identify if an information needs can be met properly*, using data provided by the CSE practices and applications. For example, if a practice is adopted at *project level*, information needs related to the project can be defined (e.g.,*What is the sprint delivery rate?*) and met by collected data in

that project. On the other hand, if a practice is *institutionalized*, information needs regarding the organization can be defined and met by data from several projects. Therefore, there is a relation between the degree of adoption of CSE practices and the capability of a organization to create data, in an application used to support a software process, that answer an information need.

4.5 Final Considerations

This chapter presented the three learning iterations performed to aid us understand the problem and design the proposed approach (*Immigrant*). The first learning iteration was an exploratory case study in which we used networked ontologies to integrate existing data stored in diverse applications to support data-driven software development in CSE. As result, we created the first version of *Immigrant* (SANTOS et al., 2021). The first learning iteration raised a demand to investigate how to understand the way an organization works and, thus, help identify its data and information needs relevant to support data-driven software development in the CSE context.

This prompted the second learning iteration, which was executed as a participative case study in which we proposed *California* (SANTOS; BARCELLOS; CALHAU, 2020; SANTOS; BARCELLOS; CALHAU, 2022). *California* is a process that supports understanding how an organization behaves and defining strategies to implement CSE practices accordingly. Information needs can be derived from the organization's current state as well as to monitor the established strategies. Considering *California* demands much time and a lot of tacit knowledge to be applied, we performed the third learning iteration to reach an easier way to understand the organization.

The third learning iteration was a multiple case study with five organizations. In this study, a diagnostic instrument was developed, called *Zeppelin* (SANTOS; BARCELLOS; RUY, 2021; SANTOS et al., 2022), to support organizations to get a panoramic view of their adoption of CSE practices. *Zeppelin* helps understanding the current state of the organization and identifying improvement areas. *Zeppelin* supports identifying the data source (e.g., applications) and data level (e.g., Product/Project level, Process level, or Institutionalized level) that can be used to answer the information needs.

Each learning iteration helped us better understand the target problem and gradually evaluate design choices we have made to develop *Immigrant*. At the end, we reached the first version of *The Band*, and, *California* and *Zeppelin* were incorporated into *Immigrant*'s current version to support identifying information needs to be met by the ontology-based integrated solution. The current version of *Immigrant* is presented in the next chapter.

5 Immigrant

We come from the land of the ice and snow / From the midnight sun where the hot springs flow / The hammer of the gods / Will drive our ships to new lands / To fight the horde, sing and cry / Valhalla, I am coming.

Led Zeppelin, Immigrant Song

This chapter presents *Immigrant*, the main artifact produced in this work. Therefore, it is related to the *Design Cycle*. The chapter is organized as follows: Section 5.1 introduces *Immigrant*, providing a general view of the approach and its components (*California*, *Zeppelin*, and *The Band*); Section 5.2 discusses the use of ontology network (ON) and Federated Information System (FIS) in *The Band*, the *Immigrant* component responsible for data integration; Section 5.3 presents the process that was followed to develop *The Band* and introduces some components of *The Band* architecture; Section 5.4 details *The Band* architecture; Section 5.5 concerns the implementation of *The Band* in this work; Section 5.7 regards how *Immigrant* can be used; Section 5.6 discusses how *The Band* meets some FIS characteristics; Section 5.8 discusses related work, and, finally, Section 5.9 presents some final considerations.

5.1 *Immigrant* Overview

As discussed in Chapter 1, considering that Software Engineering (as a whole, and Continuous Software Engineering, in particular) is a large and complex domain, involving many subdomains, we advocate using an ON to integrate application data aiming at data-driven software development. We defend the use of SEON (RUY et al., 2016), more specifically its subnetwork *Continuum*, which is proposed in this work and is devoted to CSE (see Chapter 3).

The main artifact proposed in this work is *Immigrant*[1], an ontology-based approach that uses networked ontologies to integrate application data aiming at enabling data-driven software development in the CSE context. Figure 53 shows an overview of it.

The approach considers, from the *top-down perspective*, information needs to be obtained from the current state of the organization and the actions to implement or improve CSE practices. The organization diagnosis and the definition of the actions to be performed to move from traditional to data-driven software development are supported by *California* (SANTOS; BARCELLOS; CALHAU, 2020) and *Zeppelin* (SANTOS et al., 2021), which were introduced in Chapter 4. These artifacts are meant to meet the requirement R1 (*the approach must support*

[1] The name *Immigrant* was inspired by the *Immigrant Song* by the Led Zeppelin band because the approach allows organizations to migrate from non-data-driven software development (land of ice and snow) to data-driven software development (western shore).

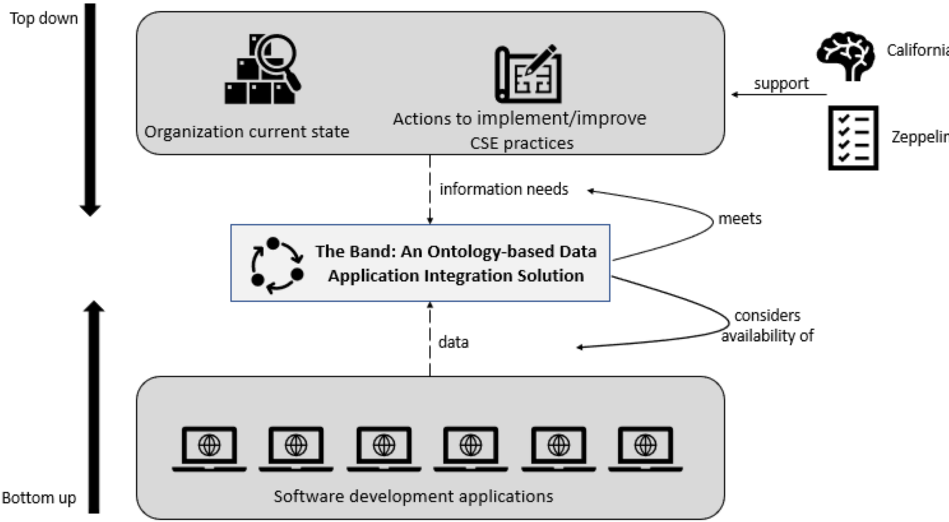


Figure 53 – Immigrant overview.

identifying the organization's information needs that are important to data-driven software development in CSE), defined in Chapter 1.

From the *bottom-up perspective*, *Immigrant* considers data available in the applications used by the organization to support its development process. Based on the information needs and available data, and using networked ontologies (e.g, Scrum Reference Ontology and Continuous Integration Reference Ontology) as an interlingua, an integration solution called *The Band*[2] defines and implements a software architecture that integrates data from different applications and shows meaningful information (e.g., in dashboards) to enable data-driven software development. *The Band* enables *Immigrant* to meet requirements R2 (*the approach must address semantic issues involved in data integration in such a complex domain*), R3 (*the approach must consider data available in the organization's applications*) and R4 (*the approach must provide integrated and meaningful data, considering the organization's information needs and available data*).

Considering that the two *Immigrant* components that support the information needs (*California*) and data sources (*Zeppelin*) identification were introduced in Chapter 4, in the next sections we focus on *The Band*, the *Immigrant* component responsible for data integration.

5.2 The use of an Ontology Network and Federated Information Systems in *The Band*

The Band is a data integration software solution based on SEON and Federated Information System (FIS) architectures. SEON architecture provides the abstraction layers to

[2] *The Band*: the idea is that each ontology-based service of the architecture is a musician that plays an instrument (*ON concepts, relations, and rules*) and the services together are responsible for creating an *music (information)* from *musical notes (data application)* to satisfy a *public (organization).*

be considered (i.e., foundational, core, and domain), the subdomains and respective concepts, relationships, and axioms to be addressed. FIS architecture, in turn, provides means to create systems that share, exchange, and combine data, and an interface for a client to access data in the systems federation. In *The Band*, each networked ontology is transformed into an *ontology-based service* (OBS) that is a system of *The Band* federation and has its own repository, called *ontology-based data repository* (OBDR). Therefore, each OBS captures, stores, and shares data related to the domain portion addressed by the referred networked ontology. Figure 54 illustrates a generic scenario of the use of ON and FIS in *The Band*.

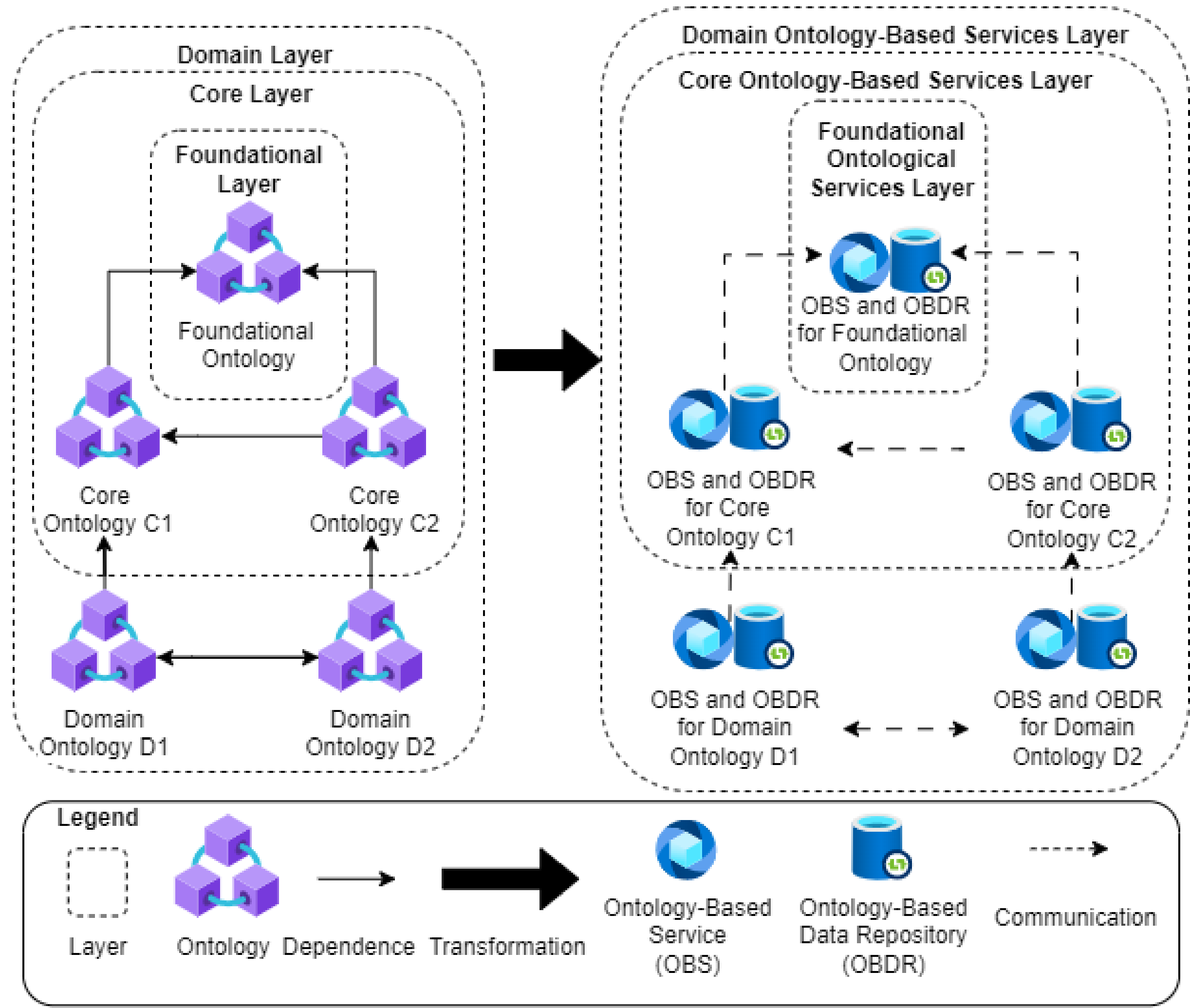


Figure 54 – Transformation of ON into FIS.

Dotted circles specify the abstraction layers, which are defined like in SEON (RUY et al., 2016), i.e., based on the ontology types (Foundational, Core, and Domain). Each networked ontology gives rise to a service, which is a system of the federation. A black line arrow indicates that the source ontology reuses concepts from the target ontology, meaning that the respective OBSs need to exchange (dotted line arrow) data related to that concept. For example, the *Code* concept (from SysSwO) is reused by SRO and CIRO. Thus, the OBSs referring to these ontologies must communicate with each other to share data related to *Code*.

An OBS is a system built based on a networked ontology. Thus, it is a system of the federation and implements concepts, axioms, and relationships from a networked ontology to manage data extracted from applications that support the software development process in the domain portion addressed by the networked ontology. It is an autonomous system that has the capability of sharing data and functionality with other OBSs or with a client.

To be autonomous, each OBS should manage all data related to the domain portion addressed by the respective networked ontology. Therefore, each OBS considers all concepts and relationships of the networked ontology, even if the concepts are from different source ontologies (i.e., reused concepts). For example, if the *Code* concept (from SysSwO) is reused by SRO, it needs to be addressed by the OBS referring to SRO. Similarly, if the *Code* concept (from SysSwO) is reused by CIRO, it needs to be addressed by the OBS referring to CIRO. As a result, the same concept is handled by more than one service. On one hand, this requires communication to ensure data consistency (e.g., if the OBSs referring to SRO and CIRO are used to integrate application data, it is necessary to ensure that data related to *Code* is consistent in both SRO and CIRO OBDRs). On the other hand, this allows one to use only the OBSs directly related to the domain portion involved in the integration scenario. For example, if the integration scenario regards agile development, one can use the OBS referring to SRO and does not need to use the OBS referring to SysSwO because the SysSwO concepts (e.g., *Code* as a *Software Artifact* produced in a **Performed Scrum Development Task**) relevant to the scrum context are also included in SRO. Thus, the SRO OBS contains the concepts necessary to represent the Scrum domains.

By creating OBSs based on networked ontologies, it is possible to observe the relationships among the networked ontologies and identify which data needs to be exchanged among different OBSs. By organizing OBSs in a FIS, relevant FIS criteria can be considered to contribute to defining the solution architecture. For example, by applying the transparency criterion[3], *The Band* must allow a client to search data without knowing where it is stored and using a query language based on a common conceptualization (i.e., concepts from the ON); while an OBS must have the capability of sharing and exchanging data with other OBSs. By applying the autonomy criterion, OBSs must be able to work independently of other OBSs and handle all necessary data to deal with the domain of interest (as discussed in the paragraph above).

5.3 Journey: *The Band* Development Process

To develop *The Band*, we defined and followed the *Journey* process, which is constituted of two phases: **Conceptual Integration** and **Integration Design and Implementation**.

[3] In FIS context, Transparency Criterion means that a client does not need to know where is the database location, schema, and query language to retrieve data from the application present in a FIS.

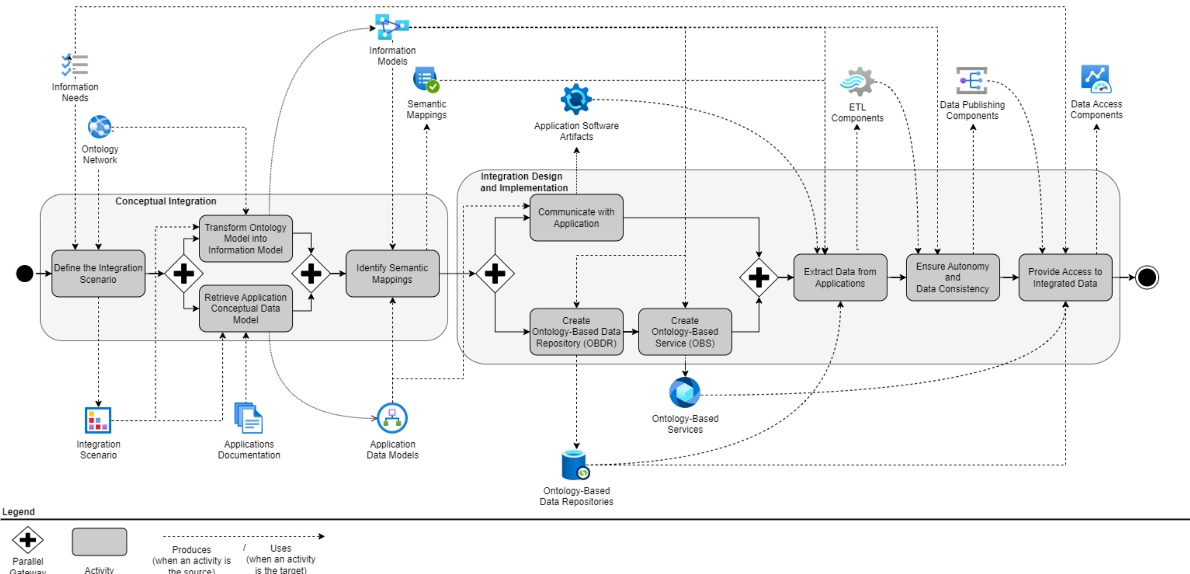


Figure 55 – *Journey* overview.

Figure 55 presents an overview of *Journey*. Next, we describe each phase and its activities.

Conceptual Integration aims to identify the semantic mappings that will serve as a basis for data integration. The first activity refers to ***Define the Integration Scenario***, which involves identifying the ON extract to be considered (i.e., the SEON fragment containing the ontologies to be used in data integration) and the applications to be integrated. The integration scenario must be defined considering the information needs to be met by integrated data.

After defining the integration scenario, it is necessary to ***Transform Ontology Model into Information Model***. In this work, we consider the transformations proposed by Carraretto (2012) and Guidoni, Almeida & Guizzardi (2020) to create information models from ontologies. The ON core and domain layers must be kept in the information models (this will be important to use the ON to create OBSs and OBDRs in the next phase). Since the ON foundational layer has the purpose of providing the common and general categorization to the other layers' concepts, and it does not contain concepts directly related to the application domain, the foundational layer is suppressed in the information model (i.e., its concepts are not replicated in this model, associations inherited from higher categories are maintained, e.g. "caused by" between an Intended and Performed Process and Activity). Figure 56 shows as an example a fragment of the SRO information model, obtained after applying transformations to the SRO conceptual model.

Besides creating an information model, it is necessary to ***Retrieve the Applications Conceptual Data Model***, which can be carried out based on the applications' documentation (e.g., API specification or software documentation) or, if documentation is incomplete or unavailable, one can extract the model by analyzing the application interface and features. For example, Figure 57 illustrates a fragment of the data model extracted from Microsoft Azure DevOps (SANTOS et al., 2021).

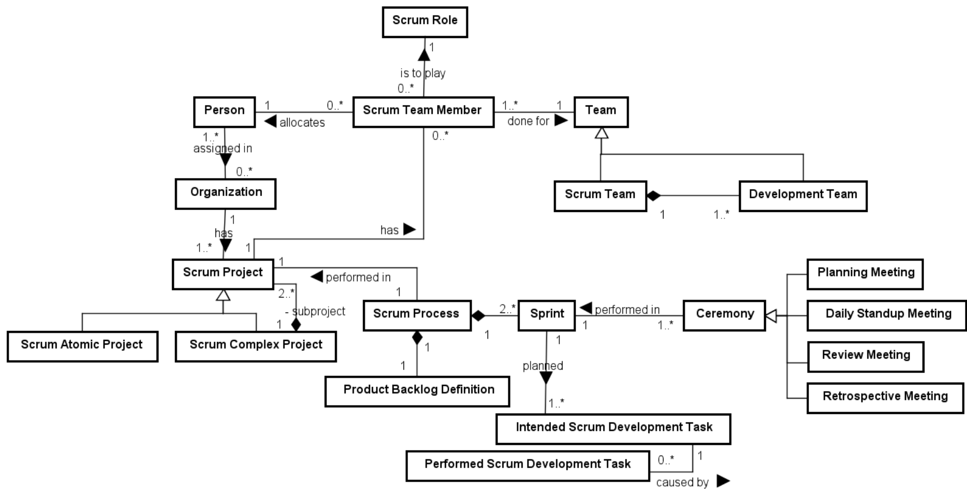


Figure 56 – Fragment of SRO Information Model (SANTOS et al., 2021).

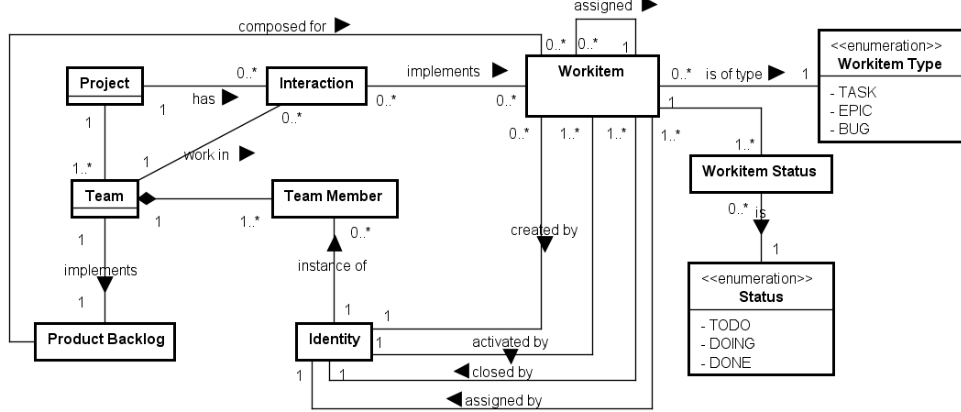


Figure 57 – Fragment of Microsoft Azure DevOps data model (SANTOS et al., 2021).

To conclude **Conceptual Integration**, it is necessary to ***Identify Semantic Mappings*** between information models and data models. In this activity, a networked ontology is used to assign semantics to the application's elements by relating the data model's elements (e.g., classes and attributes) to the information model's elements. The semantic mappings can be defined based on guidelines provided in (CALHAU; FALBO, 2010; RUY, 2017; SALAMON, 2018). Table 13 presents an example of semantic mapping between SRO information model and Microsoft Azure DevOps's Data Application Model (Semantic Mappings between Application and Networked Ontology) (SANTOS et al., 2021).

Table 13 – Examples of Semantic Mappings between SRO Information Model and Microsoft Azure DevOps Data Model (SANTOS et al., 2021).

#	**SRO Information Model**	**Microsoft Azure DevOps**
1	Person	Identity
2	Scrum Team	Team
3	Development Team	Team
4	Team Member	Identity and Team Member
5	Developer	Identity and Team Member when admin is false

Table 13 – *Continued from previous page*

#	SRO Information Model	Microsoft Azure DevOps
6	Scrum Master	Identity and Team Member when admin is true
7	Scrum Complex Project	Project, when it has many Teams
8	Scrum Atomic Project	Project, when it has one Team
9	Atomic User Story	Work Item, when WorkItem Type is "User Story"

The semantic mappings use networked ontologies as a bridge between the applications and identify which elements of the different applications are equivalent according to the ON conceptualization. They are important to implement the integration rules in the next phase. Once **Conceptual Integration** is concluded, we have *information models*, *application data models*, and *semantic mappings* that will be used in the next phase.

Integration Design and Implementation refers to defining the integration solution architecture and developing software artifacts (e.g., databases, code libraries, web services, and dashboards) to materialize data integration and support data visualization. The first activity aims to develop application software artifacts (ASA) (e.g., code library) that enable to ***Communicate with Application*** to retrieve data from or send data to it. The ASA developed to communicate with an application is based on the data model of that application. Figure 58 shows a fragment of the class diagram of an ASA (a code library) developed using Python[4] to communicate with Microsoft Azure DevOps.

The next two activities are responsible for incorporating FIS characteristics into the architecture. They concern the creation of OBSs (and respective OBDRs) as systems of a federation based on the ON, as discussed in Section 5.2. To create an OBS, first, we need to ***Create Ontology-based Data Repository (OBDR)*** from the information models, so that the OBS can use the OBDR to store and share data with other OBSs. An OBDR is a data repository based on an information model derived from a networked ontology and, thus, it represents concepts of that ontology. It can be implemented in different ways, such as a relational database or a graph database. Figure 59 depicts a fragment of the OBDR implemented as a relational database based on SRO information model (thus, we call it SRO OBDR).

Once the OBDR is created, we can ***Create Ontology-based Service*** by transforming a networked ontology (i.e., the information model derived from it) into a service, as discussed in Section 5.2. The activities that create OBDR and OBS must be repeated until all the networked ontologies represented in the information models are addressed (i.e., until each of them has an OBS and an OBDR). Figure 60 and Figure 61 show an OBS, developed in Java and using the SRO information model. A client can manage data, respectively, through REST (FIELDING; TAYLOR, 2002) or GraphQL (HARTIG; PÉREZ, 2018) interfaces, and both use the networked ontology's concepts to do that.

[4] https://pypi.org/project/azuredevopsX/

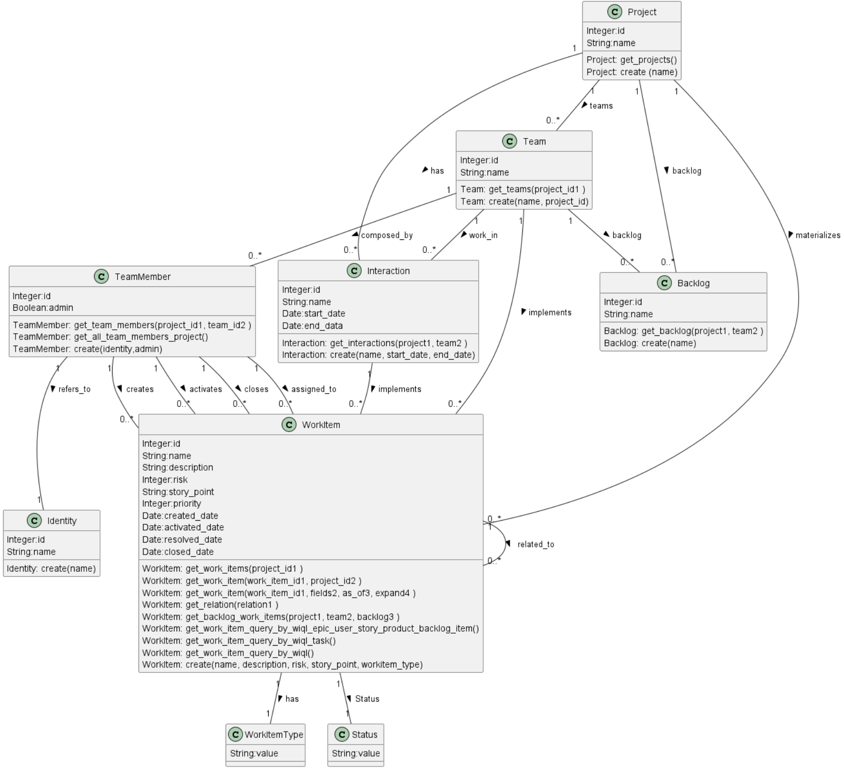


Figure 58 – Fragment of the class diagram of a lib to access Microsoft Azure DevOps data.

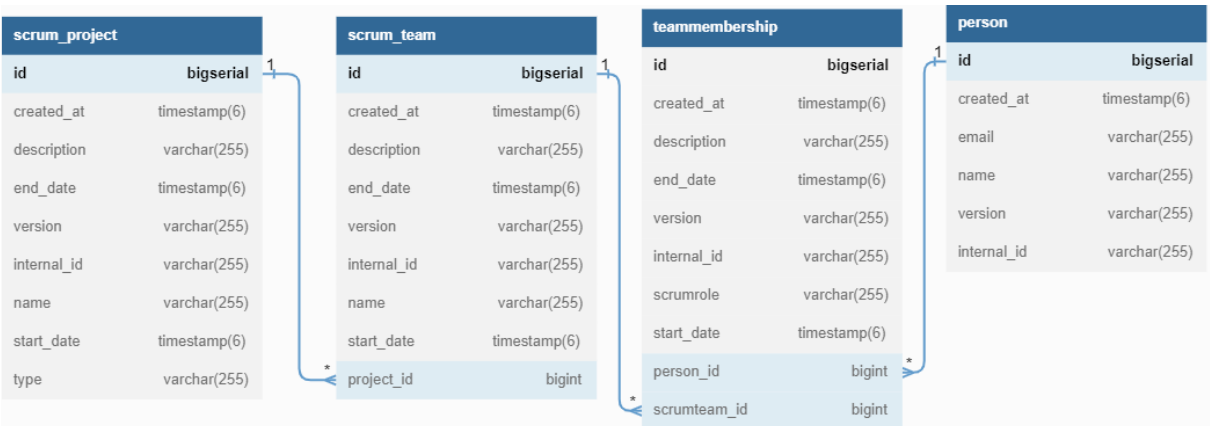


Figure 59 – Fragment of SRO OBDR.

As we explained in Section 5.2, since OBSs are autonomous, the same ontological concept (e.g., Code appears in SysSWO and CIRO) can appear in different OBDRs and it is necessary to keep consistency among them. Thus, to support data traceability (in instance level), in the OBDRs we added two attributes, *Internal_id* and *Version*, to each concept derived from the information model. *Internal_id* defines the unique identification used to identify a particular data in different OBDRs (i.e., data referring to concepts present in more than one OBDR is repeated in various OBDRs and internal_id is used to identify a particular data in all the OBDRs that contain it). *Version*, in turn, is used to identify if the OBDRs have the same version of the referred data (i.e., data referring to concepts present in more than one OBDR is repeated in various OBDRs and version is used to identify if they are synchronized).

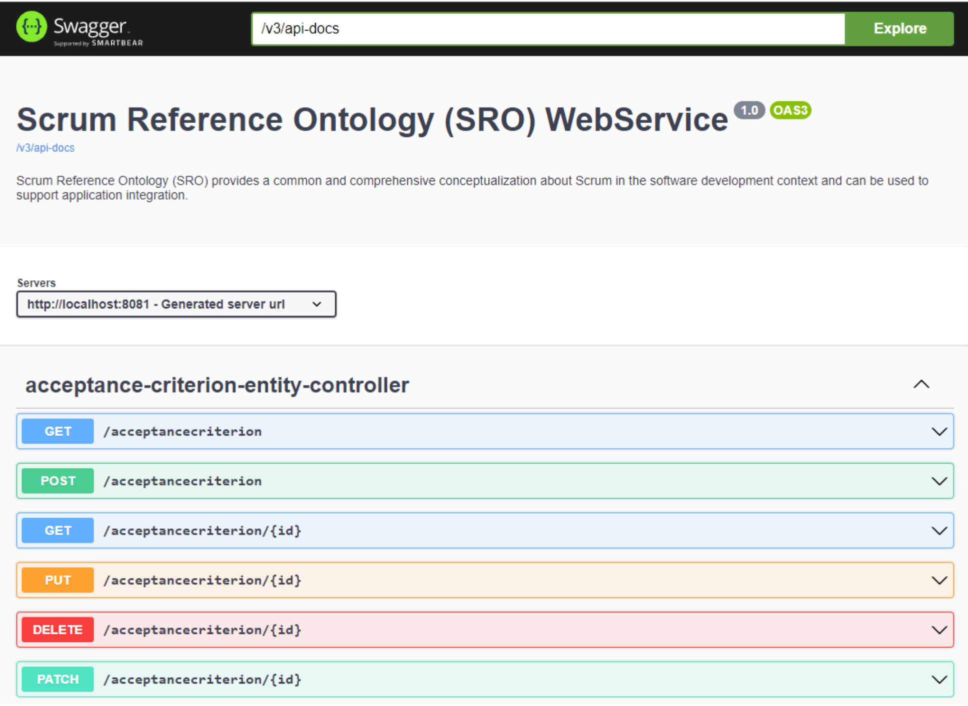


Figure 60 – SRO's OBS using a REST interface.

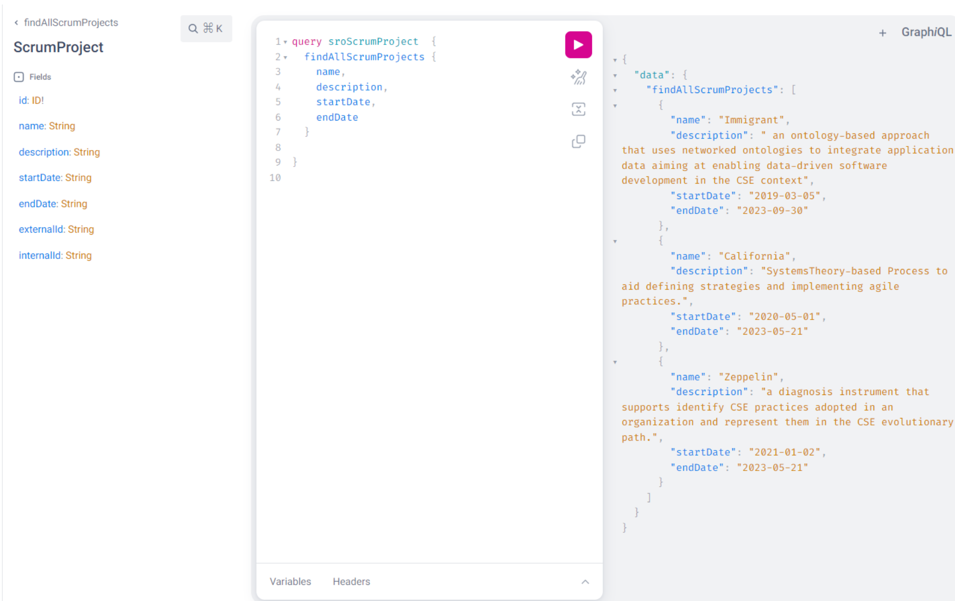


Figure 61 – SRO's OBS using a GraphQL interface.

We also added a table (considering OBDRs implemented as relational databases) called *Application Reference* to each OBDR to enable application access and data traceability to the respective application (see Figure 62). *Application Reference* provides identifiers to relate each data stored in the OBDR to a particular application and to the correspondent entity in the application database. *Application_name* refers to the application that provides data (e.g., Microsoft Azure DevOps) and *external_id* represents the identifier of the data in the application database. By doing that, data can be extracted from applications, stored in the OBDRs (which will be used in data integration), and consistency between the OBDRs and applications database

is ensured. That means, it is possible to relate each data stored in OBDR to its corresponding data in the application database.

application_reference	
id	bigserial
created_at	timestamp(6)
external_id	varchar(255)
application_name	varchar(255)

Figure 62 – Concept added to OBDR to allow data traceability.

Once ASAs, OBSs, and OBDRs are created, it is necessary to ***Extract Data From Applications*** and store it in OBDRs. Thus, we need to develop ETL components that use ASAs to extract data from the applications' database and, considering the semantic mappings, transform, and load it in OBDRs.

As we are working with a FIS based on an ON (i.e., OBSs and the respective OBDRs), it is possible that, in order to capture data related to the addressed domain portion, different OBSs need to access application data by applying different semantic mappings (e.g., the OBSs referring to SPO and SRO need to access data from Microsoft Azure DevOps to manage data about software management and each OBS considers its own semantic mappings with Microsoft Azure DevOps concepts). Thus, we can split the ETL component into two components: *Extract Component* and *Transform/Load Component*. The former extracts data from an application and publishes it in a message queue (FOWLER, 2012). The latter consumes data from a message queue, applies the transformations based on the semantic mappings, and stores data in an OBDR. This way, it is possible for several OBSs to consume data from a single queue and store transformed data in distinct OBDRs, as illustrated in Figure 63.

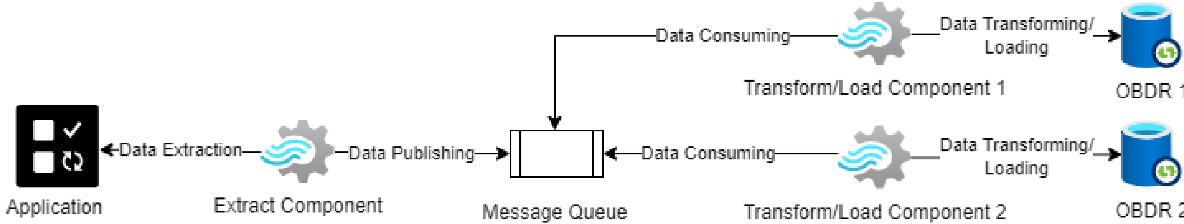


Figure 63 – Example of ETL components and OBDRs.

The next activity aims to ***Ensure Autonomy and Data Consistency***. In this activity, *data publishing components* are created to enable (i) the extract components (from ETL components) to share data with the OBSs when data is extracted from an application, and (ii) the OBSs to share and exchange data with each other when changes happen in data stored in the related OBDRs (i.e., when data is inserted, deleted, or updated in the OBDR). When data

stored in an OBDR is changed, the data publishing component uses a queue and automatically propagates the changes in all OBDRs that contain the changed data. This action is necessary to keep data consistency in all OBDRs and allow each OBS (plus its OBDR) to be autonomous in the FIS. In other words, *data publishing components* are the ones that directly support *The Band* architecture to work as FIS.

ASAs, OBSs, OBDRs, ETL components, and data publishing components are *The Band* architecture components that allow capturing data from applications, storing it in autonomous data repositories, and ensuring data consistency.

Once OBDRs are populated with application data, it is necessary to offer access to integrated data that meet the information needs to be considered in the integration scenario. For that, we must ***Provide Access to Integrated Data***, which involves developing *data access components* that enable access to data through an interface (e.g., dashboards, Rest and GraphQL APIs, or data repository command). Figure 5.1 presents an SQL Query that retrieves the names and roles from a development team member.

Listagem 5.1 – Example of SQL Command using SRO Concepts.

```
SELECT
person.name as "Name",
team_member.team_role as "Role"
FROM scrum_project
INNER JOIN scrum_process ON scrum_project.id = scrum_process.scrum_project_id
INNER JOIN scrum_team ON scrum_team.scrum_project_id = scrum_project.id
INNER JOIN development_team ON development_team.scrum_team_id = scrum_team.id
INNER JOIN team_member ON team_member.team_id = development_team.id
INNER JOIN person ON person.id = team_member.person_id
WHERE
project_name = "The Band"
ORDER BY person.name
```

The Band provides means to search data available in the OBDRs and show data that meets the information needs. Figure 64 shows a piece of the interface provided by a data access component via a dashboard.

5.4 *The Band* Architecture

In the previous section, we explained the process we followed to create *The Band* and we introduced some of the components of *The Band* architecture. In this section, we provide details about *The Band* architecture, which is shown in Figure 65.

From bottom to top, the first layer is *Application Integration Layer*, which contains ASAs to communicate with applications and ETL components (extract components) that extract data from the applications and send it to the *Internal Data Communication Layer*. It also contains *Data Publishing Components* that support the propagation of data changes to different OBDRs

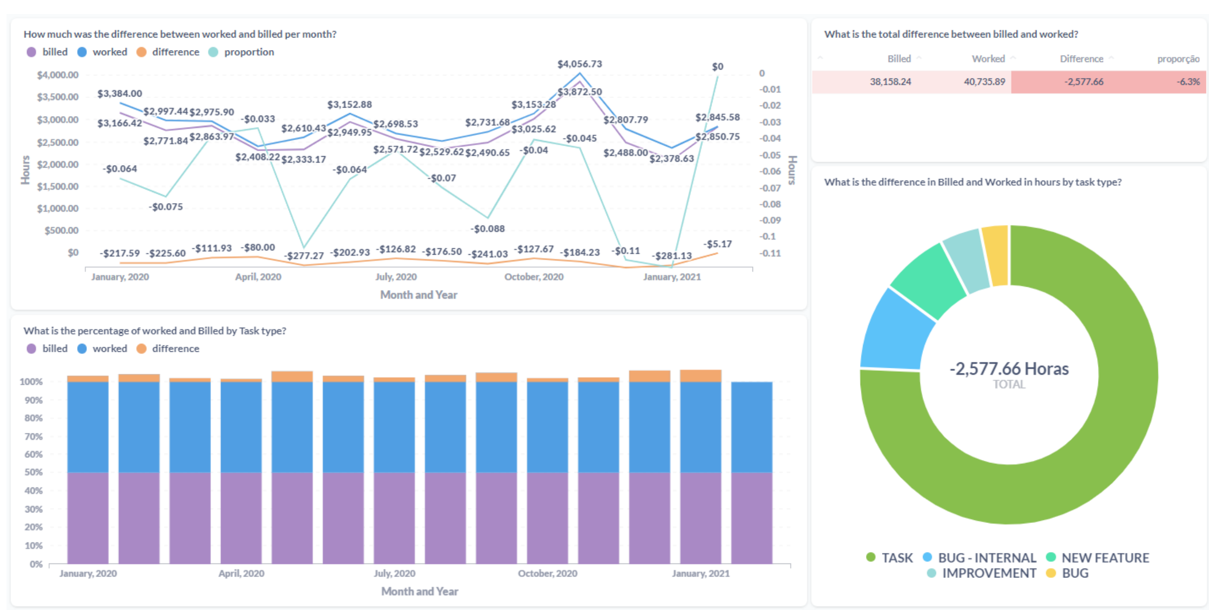


Figure 64 – Example of an interface provided by a data access component.

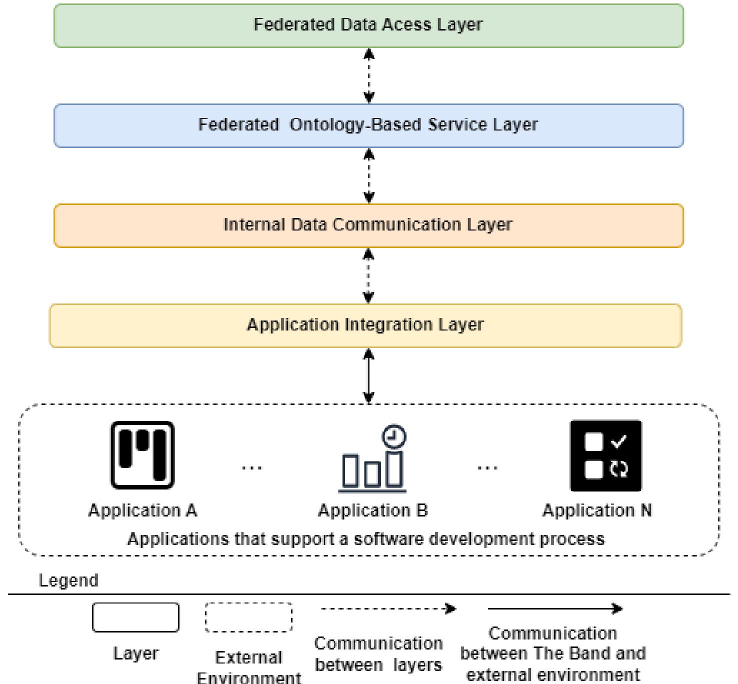


Figure 65 – *The Band* architecture overview.

of the *Federated Ontogy-based Service Layer* (via *Internal Data Communication Layer*), keeping data consistent.

The *Internal Data Communication Layer* contains *Transform/load Components* that use *ASAs* and *Data Publishing Components* from the *Application Integration Layer* to load application data into OBDRs and support data sharing among different OBSs contained in the *Federated Ontological Service Layer*. Communication among different OBSs is necessary to keep data consistent.

The *Federated Ontological Service Layer*, in turn, receives retrieve commands from the *Federated Data Access Layer*, which contains data access components and provides interfaces to data retrieving and visualization, and sends data to be presented through that layer.

When data loaded in an OBDR is changed (i.e., when loaded data is updated, deleted or new data is stored), the *Federated Ontology-Based Service Layer* communicates with the *Internal Data Communication Layer* to propagate the changes in all OBDRs containing the changed data and, thus, ensure data consistency.

The *Federated Data Access Layer* contains *Data Access components* that provide interfaces to data retrieving (e.g., APIs) and visualization (e.g., dashboard), via requests to the *Federated Ontology-Based Service Layer*. The following sections provide further information about each layer of *The Band* architecture. The notation used in the figures detailing the layers is the same of Figure 65 by adding rectangles to represent the layer components.

5.4.1 Application Integration Layer

Figure 66 presents an overview of the Application Integration Layer. As explained before, this layer contains *Extract Components* that use *ASAs* to extract data from applications and send it to the OBSs. *Data Publishing Components* publish data to be consumed by the *Internal Data Communication Layer*, which shares application data with OBSs.

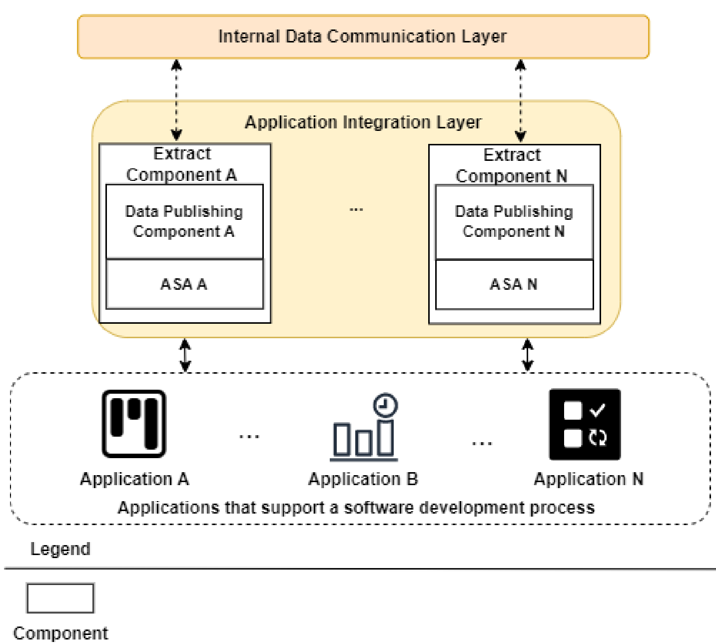


Figure 66 – Application Integration Layer architecture.

5.4.2 Internal Data Communication Layer

The *Internal Communication Layer* aims to be a channel to share and exchange data among OBSs, using a message broker (FOWLER, 2012) and *Transform/Load components.* A

message broker is an architectural pattern for messages, mediating communication among applications. It allows validating, transforming, and routing a message among the applications (FOWLER, 2012). The *Internal Communication Layer* is also used to get data from *Application Integration Layer* and send it to *OBSs*, in the *Federated Ontology-Based Service Layer* (*Transform/load Components* transform data from the message broker and load transformed data in OBDR). Therefore, the *Internal Communication Layer* is used to transport data from OBSs to *Extract components* via a message broker. Figure 67 presents the *Internal Data Communication Layer* architecture.

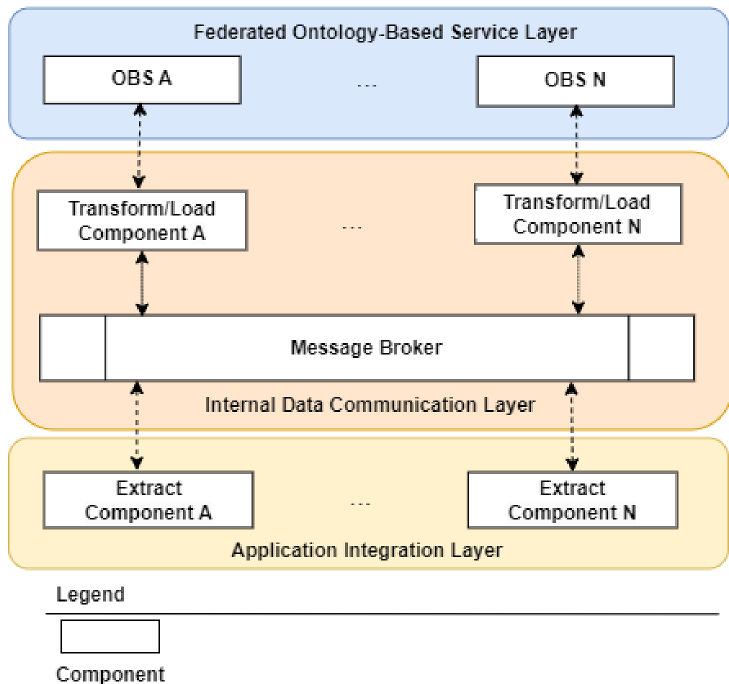


Figure 67 – Internal Data Communication Layer architecture.

5.4.3 Federated Ontology-based Service Layer

The *Federated Ontology-based Service Layer* manages the OBSs, whose architecture is organized into three layers: (i) Data Access Layer, (ii) Domain Rules Layer, (iii) Data Repository Layer, and (iv) Data Publishing Layer, as presented in Figure 68.

The *Data Access Layer* provides an interface that allows a client to manage data in the OBS, via GraphQL WebService (HARTIG; PÉREZ, 2018) or OBDR Engine. The former provides an interface based on a graph, while the latter offers an interface based on database commands (e.g., SQL). Every command given by a client is sent for the *Domain Rules Layer* to process it and return the result.

The *Domain Rules Layer* aims to manage requests from the *Data Access Layer*, send it to *Data Repository Layer*, and send the result back to *Data Access Layer*. Moreover, it contains an engine, *Domain Rules Engine*, which implements the axioms of the referred networked ontology

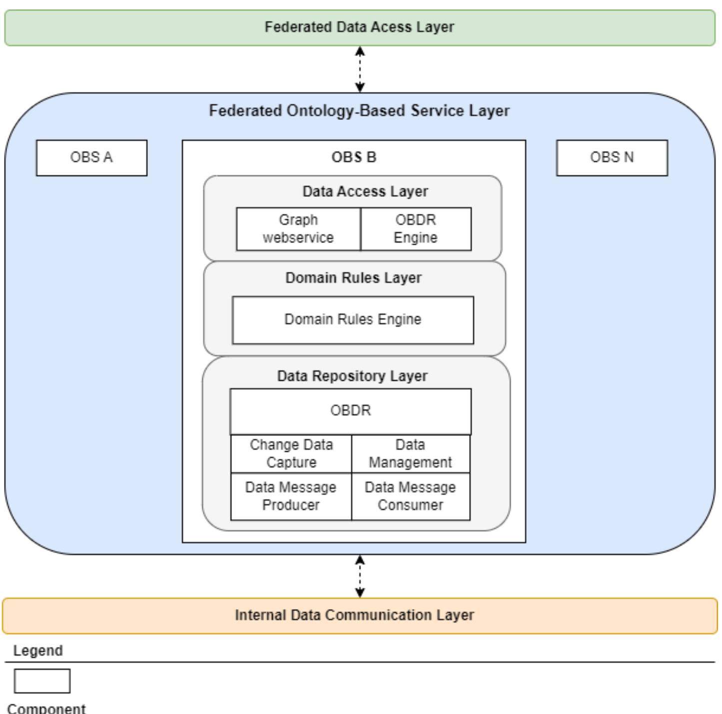


Figure 68 – Federated Ontology-based Service Layer architecture.

(as explained before, each OBS is based on a networked ontology) and verifies whether data is in conformance with the domain rules.

The *Data Repository Layer* manages the OBDR. The *Change Data Capture Component* (SHI et al., 2008) captures data when it changes (i.e. when data is created, delete, or updated) in an OBDR and then sends it to *Data Message Producer Component*. The *Data Management* component, in turn, receives data from *Data Message Consumer Component* and sends to the OBDR. *Data Message Producer Component* receives data from *Change Data Capture* and sends it to the *Internal Data Communication Layer*; while *Data Message Consumer Component*, in turn, consumes data from *Internal Data Communication Layer* and sends it to the *Data Management Component*.

5.4.4 Federated Data Access Layer

The *Federated Data Access Layer* provides access for a client to manage data in a federated way (i.e., it accesses data stored in several OBDRs in a transparent way) using *Data Access components*. This layer considers the following data access components: Dashboard, Data View, GraphQL API, and OBDR command. Figure 69 presents the *Federated Data Access Layer* and its data access components.

A client can use *GraphQL API* and *OBDR Command* components to manipulate data (i.e. create, update, and delete) or retrieve data from the OBDR using HTTP Protocol or SQL Commands, respectively. *Dashboard* allows a client to visualize integrated data that meet information needs using charts and tables. It uses the OBDR Commands (e.g., SQLs) to retrieve

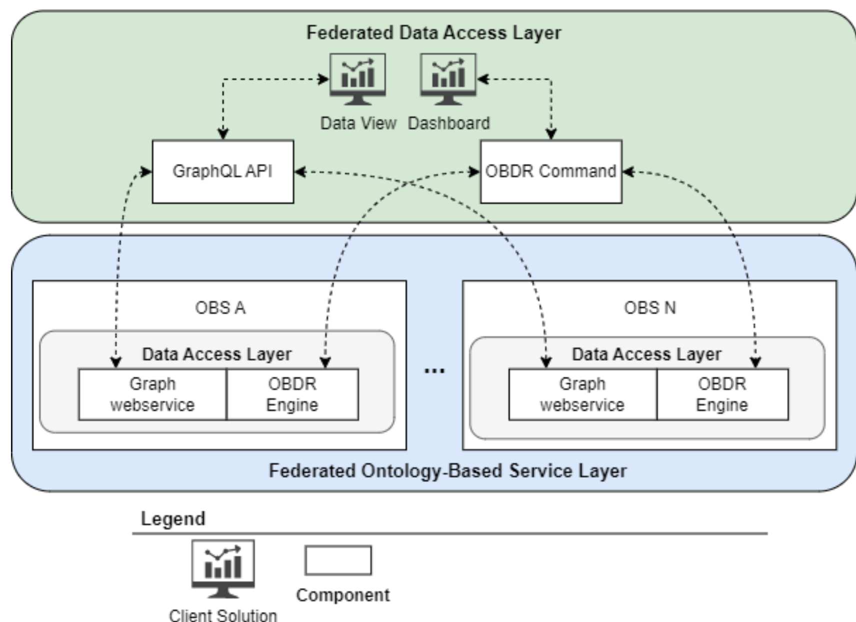


Figure 69 – Federated Data Access Layer Architecture.

data from one or more OBDRs. Figure 5.2 presents an example of SQL query using SRO and CIRO OBDRs that retrieves all software projects with did not pass an automated test.

Listagem 5.2 – Example of SQL Command using SRO and CIRO Concepts.

```
SELECT
SRO.scrum_project.name as "project_name",
CIRO.continuous_test_process.startDate as "start_date",
CIRO.continuous_test_process.endDate as "end_date",
CIRO.CI_Test_Result as "result"
FROM
SRO.scrum_project
INNER JOIN CIRO.software_project ON SRO.scrum_projetc.internal_id = CIRO.
    software_project.internal_id
INNER JOIN CIRO.continuous_integration_process ON CIRO.
    continuous_integration_process
INNER JOIN CIRO.continuous_test_process ON CIRO.continuous_test_process.id = CIRO.
    software_project.continuous_test_process_id
WHERE
CIRO.continuous_test_process.type = "UnSucessfull_Continuous_Test_Process"
ORDER BY SRO.scrum_project.name
```

Finally, a *Data View* contains forms that allow a client to manipulate and visualize data stored in an OBDR. It uses *GraphQL API* to manipulate data stored in an OBDR. Figure 70 shows a *Data View*, built using Budibase[5], that allows managing data in the SRO OBDR.

[5] https://budibase.com/

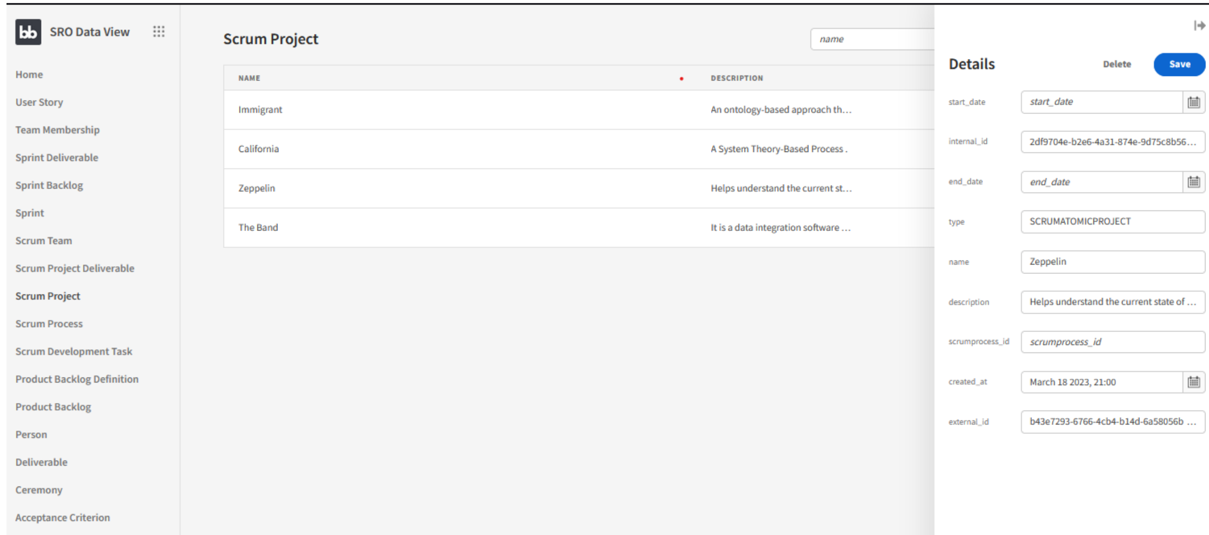


Figure 70 – SRO Data View.

5.5 Implementing *The Band*

By following the process described in Section 5.3 and the architecture presented in Section 5.4, we implemented an instance of *The Band* aiming at integrating data from Microsoft Azure DevOps (an application that supports project management), Gitlab[6] (a source repository and application that supports CI and CD), Sonar Cloud[7] (an application that addresses software quality aspects), and Clockify[8] (a time-tracking application). The used SEON fragment included the following ontologies: SPO, EO, SysSWO, RSRO, ROosT, CMPO, QAPO, OSDEF, SRO, and CIRO (these last two from the *Continuum*).

The Band is a data integration solution that looks like an "iceberg" because only 10% of it (*Data Access Components*) is above the water while 90% (ASAs, OBSs, Extract and Transform/Load Services) is under the water. In other words, a client only sees the data access components (e.g., Dashboard and Data View) while the other components supporting the data access components are not noticed by the client. Next, we present information about the technologies we used to implement *The Band*.

The dashboards were developed using Metabase[9] and Dremio[10]. Dremio creates a data lakehouse (WIKIPEDIA, 2023b) with the OBDRs and, thus, create OBDR Command that retrieves data from one or more OBDRs. Metabase allows using Dremio's SQL Commands to create dashboards. An example of a dashboard is presented in Figure 64. The data views were developed using Budibase, as shown in Figure 70. Budibase is a No-Code development platform (WIKIPEDIA, 2023e) that creates an application to visualize data saved in a data source.

6 <https://about.gitlab.com/>
7 <https://www.sonarsource.com/>
8 <https://clockify.me/>
9 <https://www.metabase.com/>
10 <https://www.dremio.com/>

The OBSs were developed using Java[11] and Spring Boot framework[12] to create Rest and GraphQL webservices, while the ontology-based data repositories were implemented using the relational database PostgreSQL[13]. An example of Rest and GraphQL is presented in Figure 60 and Figure 61.

Finally, the ASAs, Extract, Transform/Load components were developed using Python, Apache Beam[14], and Spring Cloud Stream [15], respectively. Apache Beam and Spring Cloud Stream allow extracting data from different data sources, applying transformations (e.g., joining, filter, combine, union, and split) on data, and loading transformed data on OBDR (e.g., relational database) using Python and Java, respectively.

5.6 *The Band* as FIS

As previously discussed, *The Band* is a data integration software architecture based on an ON and inspired by FIS architecture. In *The Band*, each OBS (plus its OBDR) works as a distinct, independent, and autonomous system and different OBSs can work together in a federated way.

An OBS is said autonomous because it has design, communication, and execution autonomy (BUSSE et al., 1999). *Design autonomy* means that the system is designed independently of others; while *Communication autonomy* means that the system can decide with which other systems it communicates. Finally, *Execution autonomy* means that the system executes independently, based on an incoming request.

Busse et al. (1999) define a set of criteria to characterize and classify FIS. Table 14 summarizes *The Band* classification according to each criterion.

Table 14 – Federated Information System's Criteria on The Band.

Criterion	**The Band Classification**	**Rationale**
Kind of components (that can be integrated)	Structured Source	Structured components have a pre-defined schema. All data items are intentionally defined through the schema element they belong to (BUSSE et al., 1999). The main distinction of *The Band* is the integration of OBSs. OBSs implement structured components (OBDRs) whose schema is pre-defined by SEON.

[11] <https://www.java.com/>
[12] <https://spring.io/>
[13] <https://www.postgresql.org/>
[14] <https://beam.apache.org/>
[15] <https://spring.io/event-driven>

Type of Federation	Tight	A tight federation offers a unified schema (integrated or federated schema) as an access interface to the federation. The "semantic essence" of the federated schema is a subset of the union of the semantic essence of the components schema (BUSSE et al., 1999). In *The Band*, the semantic essence is provided by the OBSs and respective OBDRs, which together provide the integrated schema (based on the information models derived from SEON).
Data Model	Common Data Model	Common data model defines a specific model that restricts the components that can be integrated (BUSSE et al., 1999). In *The Band*, the information model is derived from SEON and only components addressing the information model scope (or part of it) can be integrated. The information model represents the canonical/common data model to be used to share and exchange data between OBSs.
Semantic Data Integration Approach	Fusion and Supplementation	Fusion integration is performed to identify semantically equivalent entities coming from different sources. Supplementation integration, in turn, occurs when semantics is added to data to describe its content or semantic context (BUSSE et al., 1999). In *The Band*, semantic mappings are defined between SEON concepts and elements of the applications data models.
Transparency	Location transparency, Schema transparency, and Language transparency	In *The Band* users do not need to know the physical location of information (*location transparency*), the different denotations that entities or attributes have in different data sources (*schema transparency*), and do not need to cope with different query mechanisms and languages (*language transparency*).
Query Paradigm	Structured Queries	Structured queries assume some structure in the information which is used to specify data items in a query. DBMS are typical query-based systems (BUSSE et al., 1999). In *The Band*, the data items in the queries are specified according to the OBDRs schemas, which are based on the networked ontologies.
Engineering Approach	Top-down	A top-down approach starts from a global information need and, later, plugs in sources that can contribute to this need (BUSSE et al., 1999). In *The Band*, the integration scenario defining the ON extract to be used and applications to have data integrated is established based on information needs. Thus, global information needs are the basis for selecting the sources to be used.
Type of Integration	Materialized	In materialized integration sources are completely or partially materialized at the federation level. In *The Band*, data is persistently stored in each OBDR.

Data Access	Read-and-Write (partial Write capabilities supported)	Read-and-write access means that it is possible to read, insert or update data through the federation (BUSSE et al., 1999). *In The Band*, data stored in OBDRs can be read and when application data is changed, the modification needs to be propagated to all OBDRs that share the data. However, the user of the integration solution is not allowed to input data into the OBDRs.
Access Method	Query Language	In *The Band*, data can be accessed via SQL, REST, and GraphQL query.

5.7 Using *Immigrant*

When using *Immigrant*, the first step an organization must perform is Identify information needs to be met by integrated data. For that, the organization can use *California* or *Zeppelin* (or even other methods, such as (e.g., GQ(i)M (GOETHERT; FISHER, 2003) and OKR (TRINKENREICH et al., 2019)). Table 15 shows examples of information needs.

Table 15 – Example of the Information Needs.

#	**Information Need**
1	What is the average waiting time for a user story until development begins?
2	What is the average development time for a user story (from the start of its development until it is successfully completed)?
3	What is the average delivery time for a user story (since its creation)?
4	What is the average amount of rework during the development of a product version?

The next step is Identify available data sources, when the applications used by the organization to support the development process are selected and the available data related to the information needs is identified. Considering the identified information needs and the available data, it is possible to Define measures to be obtained from the available data to meet the information needs. It is outside the scope of this thesis to guide on measures definition. In (SOUZA, 2023), a work related to this thesis, a process called *Ramble ON* was proposed to help organizations derive measures from information needs and define them based on the conceptualization provided by networked ontologies. Table 16 presents examples of measures defined to meet information needs.

Table 16 – Examples of measures defined to meet the information needs.

#	**Information Need**	**Measure**	**Description**
1	What is the average waiting time for a user story until development begins?	Wait Time	How long a user story is in the Backlog until it is selected for development.

Table 16 – *Continued from previous page*

#	Information Need	Measure	Description
2	What is the average development time for a user story (from the start of its development until it is successfully completed)?	Cycle Time	How long a user story takes to be delivered.
3	What is the average delivery time for a user story (since its creation)?	Lead Time	How long, from its creation, a user story takes to be delivered.
4	What is the average amount of rework during the development of a product version?	Time to Repair	How long it takes for a defect to be resolved after it has been identified in the development and/or production-like environment.

The next step is Provide integrated data and it is when *The Band* is used. If the applications used by the organization are the ones considered in *The Band* implementation produced in this thesis and the provided integrated data (e.g., information provided in the produced dashboards) meet the organization's information needs, the organization can use *The Band* as it is to extract data from its applications and provide dashboards from the organization data. If the applications are the same but other information needs have to be met, then it is necessary to develop new searches and dashboards (i.e., update existing Data Access layer or develop new ones). If other applications need to be integrated, then it is also necessary to develop new ASAs and ETL components to extract data from the applications and store it in OBDRs.

It is worth pointing out that in *The Band* measures are defined at the semantic level, i.e., by using concepts from the networked ontologies instead of concepts from the applications. In this way, once data is extracted from the applications and stored in the OBDRs, it can be used to calculate values for the measures regardless of the source application. The semantic measurement formulas are defined in the OBDR command components of *The Band*. Table 17 describes some measurement formulas defined using *Continuum* concepts to the measures cited in Table 16.

Table 17 – Measurement formulas defined based on *Continuum* concepts

#	Measure	Measurement Formula
1	Wait Time	Time difference between adding it to a **Sprint Backlog** of a **Scrum project** and adding a **User Story** to the **Product Backlog** of a **Scrum Project**
2	Cycle Time	Time difference between the **End Date** of the last **Successfully Performed Scrum Development Task** related to a **User Story** and the **Start Date** of the First **Performed Scrum Development Task** related to that **User Story** in a **Scrum Project**.
3	Lead Time	Time difference between **End Date** of the last **Successfully Performed Scrum Development Task** related to that **User Story** in a **Scrum Project** and the addition of a **User Story** in a **Product Backlog** of a **Scrum Project**.
4	Time to Repair	Time difference between the **End Date** of the last **Successfully Performed Scrum Development Task** related to that **Defect** and the **Start Date** of a **Performed Scrum Development Task** performed to address a **Defect** in a **Scrum Project**

5.8 Related work

In the literature, there are some approaches using ontologies to integrate software development data. Some authors, such as Calhau & Falbo (2010), Fonseca, Barcellos & Falbo (2017), use ontologies as interlingua to integrate applications at the conceptual level. The ontology serves as a bridge to connect elements from different applications, assigning semantics to them. Similarly, in our proposal, ontologies also serve as a conceptual bridge to connect elements from different applications. Differently from the cited works, in our proposal, the ontology is transformed into an information model and it is used as an information resource to build software artifacts (e.g., databases and services) used to integrate application data. Moreover, the aforementioned works are focused on conceptual integration and induce design decisions that guide the developer to build a peer-to-peer connector between applications, while our proposal details an architecture that uses networked ontologies and ontology-based services to integrate application data. Furthermore, these works do not produce data integration software architectures with reusable components, i.e., they produce specific software solutions that can only be used in the context that were developed. This way, it is not possible to reuse the integration solution to integrate different applications (even if they meet the same information needs and refer to the same domain portion and ontologies used in the integration solution). In our proposal, we combine networked ontologies with FIS characteristics and provide an architecture containing reusable components that enable to connect different applications supporting the domain portions addressed by the OBSs and OBDRs.

In the work reported by Izza (2009), operational ontologies are used as solutions for semantically describing, discovering, and composing web services, using a single ontology, multiple ontologies, and hybrid ontology approaches. In the first approach, information sources (e.g., databases) are related to a global ontology, which can be applied only if all information sources to be integrated provide a very similar view of a domain. In the second approach, each information source is described by its own ontology and there is not a common vocabulary connecting the different ontologies or information sources. The third approach is similar to multiple ontologies (the semantics of each source is described by its own ontology) but there is a global shared ontology (or vocabulary) on which the sources are based. Differently from the cited work, our proposal focuses on data integration by using components (e.g., databases and services) developed based on an ON. Thus, we provide a data integration solution where each component shares a common conceptualization (given by the networked ontologies) that favors data exchange and sharing. Another difference is that we work with reference ontologies, which aims at providing a conceptualization independent of computational issues. Therefore, different technologies can be used to implement the solution. Izza (2009) focuses on operational ontologies, which are not developed from a foundational ontology that shares the same categorization with all ontologies used in the integration solution. Further, Izza (2009) does not use FIS characteristics to provide autonomous services in an architecture that favors

reuse and growth.

There have been some works addressing Software Process Improvement (SPI) that also propose to integrate data to support decision-making and, as such, are also related to ours. For example, Renault, Barcellos & Falbo (2018) use ontologies to integrate MantisBT[16] and Subversion[17], which are applications that support Issue Management and Software Configuration Management processes, respectively. Similarly to our work, the authors use ontologies to assign semantics at the conceptual level. However, differently from our work, the focus is on integration at the process level and supporting of decision-making to improve software processes, while we focus on integration at the data level to support data-driven software development.

We can also consider some Mining Software Repository (MSR) (HASSAN, 2008) works that aim at providing data to support decision-making in software development as related to ours. MSR aims to analyze and cross-link data present in software repositories (e.g., source control, bug repositories, deployment logs, source code repositories, and emails) to uncover actionable information about software and projects. It seeks to transform static record-keeping software repositories into active repositories that could provide information to support decision-making in software development (HASSAN, 2008). Some works addressing MSR to support software development include those of Mattila et al. (2017), Hassan (2008), Destefanis et al. (2016), Cubranic et al. (2005), Kim et al. (2006). Mattila et al. (2017) use data from Jira to guide decisions to decrease deviations between the planned and executed process. Hassan (2008), in turn, analyzes source code from code repositories to identify and propagate changes when a software artifact is modified in a project. Destefanis et al. (2016) explore data from Jira to investigate how social aspects (e.g., being polite) influence developers' productivity on agile software projects. Cubranic et al. (2005) and Kim et al. (2006) discuss that linking data from different and heterogeneous software repositories (e.g., email, source control repository, and chat) could improve data quality and, thus, provide a more complete view of a project.

Similarly to our work, these works aim to use and integrate existing data to provide useful information to support decision-making in software development. However, differently from our work, the authors of the aforementioned works were not concerned with semantic aspects explicitly. As we previously discussed, neglecting semantic aspects can lead to conflicts whenever the same information item is given divergent interpretations (WACHE et al., 2001).

Our work proposes the use of networked ontologies to assign semantics to data and structure services and repositories in the integration solution. The created OBSs and OBDRs are used to integrate applications. They work as systems of a FIS, making it easier to add new applications or change the ones that were integrated to others addressing similar scope (e.g., when the organization changes one application for another). Once semantics is assigned to

16 <https://www.mantisbt.org/>
17 <https://subversion.apache.org/>

applications' information items (e.g., class, attributes), it is possible to change an application data repository for another (e.g., from Microsoft Azure DevOps to Jira). The cited works, in turn, provide solutions considering the data structure of the used repositories, which makes it difficult to reuse them with different repositories. By using networked ontologies, our work not only supports the integration solution but also helps understand the domain of interest. We argue that ontology-based approaches such as the one proposed in this work could contribute to MSR solutions by providing comprehensive and well-founded conceptual models to combine and interpret data extracted from software repositories as well as support the linking of data from different repositories.

In conclusion, by analyzing all cited works, we notice that none of them use ON, provide characteristics enabled by an architecture based in FIS, produce ontology-based reusable software components, and address CSE aspects, as our proposal does. Moreover, when considering *Immigrant* as a whole (the previous discussion concerns mainly *The Band* component), our proposal is the only solution that aids data integration and supports identifying the information needs to be met by the resulting integrated data.

5.9 Final Considerations

This chapter introduced *Immigrant*, the main artifact produced in this work. It is an ontology-based approach to integrate application data aiming at enabling data-driven software development in the CSE context. *Immigrant* uses ontologies from SEON (RUY et al., 2016), particularly the ones from *Continuum*, a subnetwork developed in this work and devoted to CSE aspects. *Immigrant* has three components *California*, *Zeppelin*, and *The Band*, which have different and complementary roles.

Immigrant considers both, *top-down* and *bottom-up* perspectives. From the *top-down* perspective, the organization information needs must be identified. This can be supported by *California* and *Zeppelin*, which help understand the organization current status, identify CSE practices to be implemented or improved and, thus, derive information needs to be met. From the *bottom-up* perspective, available application data must be considered and integrated to meet the information needs. Data integration is supported by *The Band*, a data integration software solution that allows to capture and integrate data from applications supporting the software development process and provides integrated data in dashboards to aid software development and decision-making. This chapter presented an overview of the *Immigrant*, provided details about *The Band* and discussed related work.

The Band uses SEON and incorporates FIS characteristics to create OBSs and OBDRs that work as systems of a FIS. Some benefits of using an ON to develop FIS in a data integration solution are:

- The ON provides the "big picture" of which subdomains and concepts can be addressed in a data integration solution related to the ON domain. For example, *Continuum* helps understand which subdomains, concepts, and relations need to be considered (and, thus, integrated) in a data integration solution for the CSE domain.
- The ON structure helps define the FIS structure in which each networked ontology gives rise to components (here, OBSs plus OBDRs) that are systems of the FIS. The axioms help define integration rules to keep data consistency in the different systems (i.e., OBSs) that are part of the federation (i.e. *The Band*). In *The Band*, OBSs (plus respective OBDRs) are autonomous systems that can work alone (i.e., independently of other OBSs and can also share and exchange data with other OBSs of the federation. The relations between different OBSs are defined based on the relations between the networked ontologies. This helps identify data that must be shared.
- The ON architecture provides a basis to the integration architecture and supports different views of integrated data. For example, the ON layers can be used to categorize and organize OBSs in layers in the FIS architecture. This way, the FIS architecture can provide different viewpoints of data according to the aimed layer. For example, integrated data about software process performance can be provided in a core view by using SPO. On the other hand, specific data about agile software process can be provided in a domain-specific view by using SRO.
- The ON evolution enables the integrated solution evolution. If the ON evolves (by applying evolution mechanisms such as merging, reengineering, and aligning (SALAMON, 2018)), the FIS architecture can evolve accordingly. That is, new OBSs can be created to address the new domain portion added to the ON, making it possible to integrate new data from applications supporting that domain portion.

The next chapter presents a learning interaction performed to evaluate *Immigrant* in a software organization that desired to achieve a data-driven software development process.

6 Final Learning Iteration: Applying *Immigrant* in a Software Organization

Many times I've lied, many times I've listened, many times I've wondered how much there is to know.

Led Zeppelin, Over The Hills And Far Away

This chapter presents the fourth and last learning iteration performed in this work. It consisted of a participative case study in which *Immigrant* was used in a software organization that desires to implement a data-driven development software process. This chapter is related to the *Design Cycle*, since it regards evaluating the proposed artifact. Section 6.1 introduces the study context by describing the organization in which *Immigrant* was applied. Section 6.2 presents the study planning. Section 6.3 concerns the study execution and results. Section 6.4 discusses the results. Section 6.5 addresses threats to validity. Section 6.6 discusses what we learned in this last learning iteration. Finally, Section 6.7 presents the final considerations of the chapter.

6.1 Context

Organization X (its name was omitted for anonymity reasons) is a Brazilian software organization that decided to evolve from an agile to a data-driven software development process. Org X is a small 3-year-old fintech startup that provides financial solutions to small Brazilian financial funds. In addition to the directors, it has four software developers, one designer, and one financial expert. Org X adopts a software development process based on Scrum and other CSE practices (e.g., Continuous Integration, Continuous Delivery, and Continuous Deployment). Development team members have between one and three years of practical experience working in software organizations. Three out of the four developers started working as interns and after graduating were hired. The author of this thesis worked at Org X from January 2022 to March 2023 as a consultant.

The development team is responsible for the software requirements specification, design, coding, testing, deployment, and operation processes, i.e., the development team is responsible for performing every activity related to software development and operations. As a consequence of the many responsibilities, some problems emerged, such as projects delays and budget overruns, increased software defects, overloaded teams due to rework, and communication issues with clients. Aiming to minimize these problems, Org X decided to improve the software process by improving the performed CSE practices or implementing

new ones aiming at a data-driven software development. According to the director: "*the main difficulty is identifying problems during the process execution in an easy way and using data to support decision-making in development and business contexts*". In addition, the director commented that they "*desire to automate some aspects of the software process to reduce repetitive teamwork in activities related mainly to project management and software development (e.g., creating, relating and maintaining consistency of data regarding epics, user stories, and tasks in the tools)*". Considering this scenario, we proposed to use *Immigrant* to enable Org X to perform data-driven software development.

6.2 Study Planning

The study consisted of a participative case (BASKERVILLE, 1997) study whose **goal** was to evaluate *Immigrant* in a real context to verify if it is useful and if its use is feasible. It is important to observe that the *Immigrant* components have already been evaluated in isolation. As discussed in Chapter 4, we performed studies as learning iterations (LI) to evaluate *California* (LI2 in Figure 1) (SANTOS; BARCELLOS; CALHAU, 2022), *Zeppelin* (LI3 in Figure 1) (SANTOS et al., 2021) and *The Band* (LI1 in Figure 1) (SANTOS et al., 2022). The results suggests that, when used in isolation, each component is useful and its use in software organization is feasible. However, we have not evaluated the use of all components in the same organization. Moreover, after LI1 we evolved *The Band* (SANTOS; ALMEIDA; BARCELLOS, 2023) and we need to evaluate the current version. Thus, this study aimed to evaluate the use of all components of *Immigrant* in the same organization. The participative case study was selected as the research method in this study because the researcher (this thesis author) acted as a consultant in Org X . Together with other participants, he gathered information to understand the organization and defined strategies to enable data-driven software development. Thus, the researcher had some control over some intervening variables.

Aligned with the study goal, we defined the following main **research question**: *Is Immigrant useful and is its use in software organizations feasible?* To answer this research question, we decomposed it into other four: (i) Is *Zeppelin* useful and its use in software organizations feasible?; (ii) Is *California* useful and its use in software organizations is feasible?; (iii) Is *The Band* useful and its use in software organizations is feasible?; and (iv) Is the combined use of the three components of *Immigrant* useful and feasible?

The **procedure** followed in the study consisted of getting general information about the organization, applying *Immigrant*, and getting feedback about its use. For applying *Immigrant*, we followed the steps presented in Section 5.7:

Step1 - Identify information needs: in this step *Zeppelin* (SANTOS et al., 2022) is used by the participants, with the help of the researcher. The analytic report results are presented to the participants that must define information needs based on *Zeppelin*'s data. To complement

the identified information needs, *California* (SANTOS; BARCELLOS; CALHAU, 2020) is used to identify undesirable behaviors and its causes, define strategies to address them and, thus, derive information needs. To conclude this step, the information needs identified from *Zeppelin* and *California* data are prioritized and the ones to be considered in the study scope are selected.

Step2 - Identify available data sources: in this step the applications that support the development process in the organization are identified.

Step3 - Define measures: in this step the participants define the measures to meet the identified information needs and that can be obtained from the available data.

Step4 - Provide integrated data: in this step *The Band* (SANTOS; ALMEIDA; BARCELLOS, 2023) is used to collect data from the applications and provide integrated data enable data-driven software development. For that, first, it is verified whether *The Band* supports the applications, information needs, and measures identified in the previous steps. If new components are necessary, they are provided by the researcher.

6.3 Study Execution, Data Collection, and Results

In this section, we present information about the study execution and show some of the collected data and produced results. The study execution and data collection were performed from September to December 2023.

By following the study procedure, we started by getting general information about the organization. For that, we performed an interview with the director to gather information about the organizational environment, culture, relationship with partners, future plans, software development process, software development issues, and agile knowledge. The director was told to feel free to talk as much as they wanted to. The interview lasted about 60 minutes. The funnel questions technique was used, i.e., the interview started with general questions (e.g., *"What kind of software does the organization develop?"* and *"How is the software development process?"*), and then went deeper into more specific points (e.g., *"Tell me more about the software test activity"*). Among the information provided by the directors, he pointed out that "*He desires to implements a data-driven software culture inside of the development team, i.e., the decision-making and daily activities should be based on data and not only in the team's experience*".

The director said that "*the development team has difficulty developing a quick prototype to validate an idea with a client. The development team uses the same techniques to develop real and prototype software. As a consequence, the development team spends almost the same time in the development of a prototype and a real project. Thus, he has difficulty validating the ideas with clients*". Besides, He described that "*development team has difficulty understanding the problem domain concepts and implements CSE practices (e.g., automated testing with a modular architecture.)*".

Finally, the director expressed the desire "*to gain a comprehensive understanding of the current state of the projects development process*". This includes assessing whether the projects are on track for timely delivery, ensuring that the developers are producing high-quality software artifacts, and evaluating the investment made in each project. Furthermore, the director seeks "*to identify which features are most frequently utilized by clients*". According to the director, this information allows Org X make decisions to improve the software development process, products, and validate hypothesis about the client's needs.

After this overall view, we applied *Immigrant*. We started by *identifying information needs*. To understand the use of CSE practices in Org X, the development team members answered the *Zeppelin* questionnaire (SANTOS et al., 2022). We represented collected data in tables and graphs, analyzed them, presented the analysis results to the development team and director and, together, defined information needs based on *Zeppelin*'s data. After that, we applied *California* (SANTOS; BARCELLOS; CALHAU, 2020) to identify undesirable behaviors and its causes and, then, defined strategies to address them. Thus, together with the team and director we derived information needs. We prioritized the defined information needs and selected the ones to be addressed. Next, we *identified the available data sources* by identifying the applications Org X uses to support the development process, which are: Microsoft Azure DevOps, to support project management aspects, GitLab, a Source Code Repository and Continuous Integration Server, and Sonar Cloud, which supports code quality inspection. Then, considering data available in the application data sources and the identified information needs, together with the team we *defined the measures* to be represented in *The Band* dashboards. Finally, we checked whether *The Band* supported the identified applications, information needs, and measures, and we noticed that it would be necessary to develop new components to provide data related to some of the defined measures and information needs. We developed such components and, then, we used *The Band* to *provide integrated data*. In the following sections we present some of the results produced in each of these steps.

6.3.1 Identifying Information Needs from *Zeppelin*

After the interview with the director, we applied *Zeppelin* to identify information needs . *Zeppelin* Questionnaire was made available to the development team in June 2023. The questionnaire was answered by the development team one week after the day it was sent. After that, we prepared the *Zeppelin*'s Analytics Report and sent it to the team. Then, we performed a meeting with the team to discuss the results and clarify some points. Table 18 summarizes the adoption degree of CSE practices at each StH stage and Table 19 summarizes the adoption degree of CSE practices by Eye of CSE category.

Table 18 – Adoption Degree by StH stage.

StH Stage	**CSE Practices Adoption Degree (%)**
Agile Organization	16.15
Continuous Integration	21.33
Continuous Deployment	6.47
R&D as Innovation System	20.76

Table 19 – Adoption Degree by Eye of CSE category.

Eye of CSE category	**CSE Practices Adoption Degree (%)**
Technical Solution	36.66
Team	17.5
Development	17.27
Business	16.66
Quality	16.66
Software Management	16.25
Knowledge	13.63
User/Customer	5.71
Operation	0

The meeting lasted about 120 minutes. We spent 30 minutes explaining the *Zeppelin*‘s Analytics Report and the rest of the meeting was used to understand how the CSE practices were adopted. For that, we interviewed the development team to understand in detail the provided answers to *Zeppelin*’s Questionnaire. Concerning the *Agile Organization* practices, a developer said that the development process is based on Scrum. However, Scrum roles are not defined clearly, and it is common for a member to play different roles in the same project (e.g., Scrum Master and Developer). Another developer commented that “*he has difficulty identifying when the director wants a prototype or a real application*”. The developers commented that Org X has “*a culture to develop a proof of concept (POC) to understand the problem domain and create solutions*”. According to them, a POC is useful for understanding the project’s problem and designing a solution that brings value for a client and reduces rework. The developers commented that “*they are free to contact the client to resolve any doubts about the project*”. Regarding the development cycles, the sprint time box lasts five days (i.e., one week). The development team spends four days, from Monday to Thursday, developing and deploying a feature, and the last day, Friday, is dedicated to planning the next sprint. This sprint configuration has proved to be efficient. Microsoft Azure DevOps is used as a project management supporting application.

Concerning time estimates, the developers commented that “*the development team has difficulty estimating how long will take for a US to be materialized after being added in the Product Backlog or Sprint Backlog*”. The lack of this information brings difficulty to the director when he wants to show the product to potential clients because he does not know when the features will be incorporated into the product. Thus, it is common for the director to define the date

to deliver the features, and, thus, the development team develops the features based on that schedule. As a result, a feature can be deployed in the production environment without being tested sufficiently, as pointed out by a developer.

Regarding the problem domain that the organization works on, many of the developers have never worked with this domain before joining the organization. Thus, the developers said that there is an overhead when developing a solution, as they must learn about the domain at the same time they develop the solution. This causes a set of problems related to solution quality, acceptance criteria, and business rules. To alleviate these problems a little, the team produces small prototypes to validate the understanding of the problem.

Regarding *Continuous Integration* practices, the developers said that they "*tried to implement automated test when it is possible*". A developer said that *"sometimes it is not possible to develop automated tests because it is learning about the domain at the same time it is developing the solution*". In addition, a developer commented that "*it is common to create automated tests or review the code to improve some part of the project after the feature be delivered*". Another developer commented that desired to use test-driven approaches (e.g., Test-Driven Development -TDD- or Behavior-Driven Development - BDD) to validate the produced user stories and code.

The development team uses GitLab and Sonar Cloud to support the *Continuous Integration* process. GitLab is used as a source repository, Continuous Integration and deployment server; while Sonar Cloud is used as a code static inspection server. A CI pipeline was configured in GitLab where a code is tested by automated tests and verified by Sonar when a pull request is approved or a code is committed on the development branch. Org X uses Git Flow as the process to control the code integration. However, GitLab is not configured to stop the code integration when Sonar Cloud identifies a quality problem. According to the team, "*this has caused a lot of corrective maintenance problems throughout the projects*". Moreover, a developer commented that "*the development team used to discuss code quality, based on Sonar data, in retrospective meetings, but due to the increase in work demand this practice was no longer carried out*".

Concerning *Continuous Deployment* practices, Org X develops solutions using container technologies (e.g., Docker[1]). Each application is deployed, manually, in a production environment, in a cloud solution. Org X would like to automate this process. Thus, a developer was assigned to study Infrastructure as a Code concepts to create a production environment and deploy an application in an automatic way. According to a developer, "*Org X is studying to use technologies that implement the concepts of Infrastructure as Code[2] (e.g., Terraform[3]) and Continuous Deployment (e.g., ArgoCD[4]) to reduce deployment time in a production environment from hours to minutes*". Another benefit expected to be achieved is "*collecting data from deployment*

1 <https://www.docker.com/>
2 <https://martinfowler.com/bliki/InfrastructureAsCode.html>
3 <https://www.terraform.io/>
4 <https://argoproj.github.io/cd/>

and operation applications to improve the development process", as commented by a developer and director. According to them, data from deployment and operation can bring insights into how the development process and the products can be improved.

Regarding *R&D as Innovation System* practices, Org X does not have systematic practices to improve the products or create new ones based on users' feedback. According to the director and the developers, they use "*feedback and interview with the clients to identify what needs to be improved in the current products (e.g., develop a new feature) or to develop a new product*". In order to implement data collection in a production environment, a developer is studying how an Application Performance Monitor[5] tool (APM) (e.g., SkyWalking[6]) can provide data about the products in a production environment to business and development teams. The idea is to use the data to improve the products or create a new one based on data provided by the application. Finally, they are thinking of adding features to the application to enable identify which features are more used by the clients.

At the end of the meeting, the interviewer commented that some information provided by the interviewees was not in conformance with the answers provided in *Zeppelin*'s Questionnaire. This way, it was necessary to review the answers to get a correct overview of Org X. Thus, the development team reviewed the answers provided to *Zeppelin*'s Questionnaire, with the participation of the interviewer. As a result, a new CSE panoramic overview was obtained and it was presented to the team, which approved it. Table 20 and Table 21 present the adoption degree of CSE practices in Org X by StH stages and Eye of CSE category after the meeting.

Table 20 – CSE Practices Adoption Degree by StH' Stages, before and after interview.

StH Stage	**CSE Practices Adoption Degree before interview (%)**	**CSE Practices Adoption Degree after interview (%)**
Agile Organization	16.15	15
Continuous Integration	21.33	18
Continuous Deployment	6.47	7.6
R&D as Innovation System	20.76	2.3

Table 21 – CSE Practices Adoption Degree by Eye of CSE, before and after interview.

Eye of CSE's Dimensions	**CSE Practices Adoption Degree (%) before interview**	**CSE Practices Adoption Degree (%) after interview**
Technical Solution	36.66	46.66
Team	17.5	0

[5] <https://en.wikipedia.org/wiki/Application_performance_management>
[6] <https://skywalking.apache.org/>

Table 21 – *Continued from previous page*

Eye of CSE's Dimensions	**CSE Practices Adoption Degree (%) before interview**	**CSE Practices Adoption Degree (%) after interview**
Development	17.27	14.44
Business	16.66	6.7
Quality	16.66	6.7
Software Management	16.25	20
Knowledge	13.63	0.9
User/Customer	5.7	5.7
Operation	0	30

As can be observed in tables 20 and 21, the CSE practices adoption degrees changed after the meeting to clarify some questions and better understand the performed practices. This revealed that *Zeppelin* should be applied with the supervision of CSE specialist and requires improvements (i.e., clear guidelines about what each practice means) for other people to be able to properly apply it without the researchers' intervention.

Finally, considering the CSE panorama obtained from using *Zeppelin*, the interviewer, the development team, and the director defined information needs. Some of them are shown in Table 22.

Table 22 – Information needs identified based on the use of Zeppelin.

#	**Goal**	**Information Need**	**Description**
1	Improve time to deliver features	(1.a) What is the average waiting time for a user story from its creation until its development is initiated? (1.b) What is the average development time for a user story (from the beginning of its development until it is successfully completed)? (1.c) What is the average delivery time for a user story (from its creation)? (1.d) What is the average amount of rework during the development of a product version?	(1.a), (1.b), and (1.c) contribute to understanding the time spent on delivering new functionalities. An improvement in the value of (1.c) is a direct indicator of progress toward the goal of improving delivery time. (1.d) helps identify whether there has been a significant amount of rework that is impacting the delivery time of new functionalities.

Table 22 – *Continued from previous page*

#	Goal	Information Need	Description
2	Improve sprint planning	(2.a) What is the average time to successfully complete a task in a sprint? (2.b) What is the rate of tasks planned and not completed in a sprint?	(2.a) helps understand how long takes to accomplish a task recorded in the backlog. The scrum master/project manager can use this information to help estimate how many tasks can be included in the sprint backlog. (2.b) helps identify how much work has been planned for the sprint and not concluded. An improvement in values of (2.b) is a direct indication of the reach of the objective of improving sprint planning.
3	Improve the quality of the software artifacts	(3.a) Which types of code smells are produced in the software artifacts? (3.b) How many code smells are produced in the software artifacts? (3.c) How many code smells are produced by each developer?	(3.a) and (3.b) help understand potential quality code problems produced in software artifacts. This information helps identify and prevent code problems and also supports managers in defining training strategies for the development team in methods that improve quality. (3.c) helps identify if there is a predominance of any developers in producing code smells. This information can be used to identify training needs and improve the developer's skills.

6.3.2 Applying *California* to complement the Information Needs

After identifying information needs considering *Zeppelin's* data, we applied *California* to identify undesirable behaviors, their causes, set strategies and derive new information needs. Considering data gathered using *Zeppelin* Questionnaire and the interviews performed when discussing the Analytic Report with the team, we built systemic maps to detail some scenarios. Figure 71 illustrates a fragment of systemic maps. The elements in blue in the figure form a modeling pattern that reveals the presence of the archetype *Shifting the Burden*.

As previously said, there is a *lack of expertise in the development team* regarding both the organizational business and software development. Thus, developers often produce *Poorly defined requirements*. Additionally, there are *Communication problems* related to project subjects between the director (Business) and the developers (Development) . For example, it is common

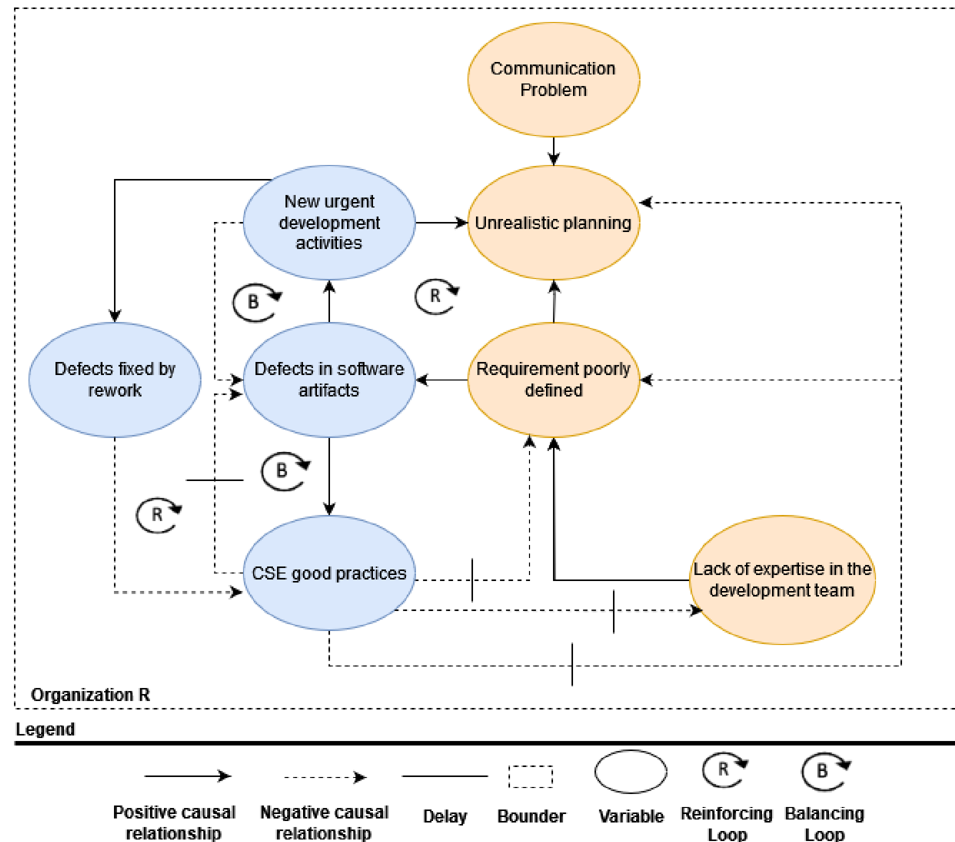


Figure 71 – Fragment of systemic map.

that the director does not express correctly that he desires a prototype to show an idea to a potential client and the development team understands that they desire a robust product in a few weeks. Therefore, *Unrealistic planning* (scope, goals and deadlines) is defined to the projects. .

Poorly defined requirements and *Unrealistic planning* contribute to increasing the number of *Defects in software artifacts*. This way, the development team needs to fix defects by performing *New urgent development activities*, as fast as possible (*Defects fixed by rework*), which decreases the number of *Defects in software artifacts*. The side effect is that the *CSE good practices* take time to be implemented in Org X. These variables and the relations between them characterize the *Shifting the Burden* archetype. It is a complex behavior structure because the balancing and reinforcing loops move the system (Org X) in a direction (*New urgent development activities*) usually other than the one desired (*CSE good practices*). *New urgent development activities* contribute to increasing problems relate to scope, goal, deadline in a project (*Unrealistic planning*) because these activities were not initially planned in the project.

Based on the systemic maps, *Zeppelin* data and interviews, it was possible to identify the *undesirable behaviors* presented in Table 23. They were prioritized by using GUT matrix, as we explain in (SANTOS; BARCELLOS; CALHAU, 2022)

Table 23 – Some of the identified undesirable behaviors

#	**Undesirable Behaviors**	**G**	**U**	**T**	**GxUxT**
UB1	Use of the same software development methods and technologies to develop a prototype to validate an idea and to create a software product.	5	5	5	125
UB2	Development of software artifacts based on misunderstood requirements.	5	5	5	125
UB3	Continuous Integration and Continuous Deployment techniques (e.g., automated test and automated quality inspection) are not often applied to build software artifacts.	5	5	4	100

For each undesirable behavior, we identified its causes. Table 24 presents the main causes of the undesirable behaviors.

Table 24 – Causes of Undesirable Behaviors.

#	**Causes**	**UB1**	**UB2**	**UB3**
C1	The software development process is the same to develop a prototype and a final product	x	x	-
C2 -	Poor communication between director and development team	-	x	-
C3	Lack of knowledge of the problem domain	-	x	-
C4	Organization members have different experiences with CSE practices	x	-	x
C5	CSE concepts and practices are not well-known by the organization	x	x	x

C1 and C2 were identified directly from the interviews. Regarding C1, the director and development team described that the software development process does not have activities, methods, or techniques that guide the developers to build a prototype. Although there is an open channel between the director and development team, there is a lot of miscommunication between them. Many times, the developers understand that they should build a product, while a prototype was required (C2) and vice-versa.

C2 *Poor communication between director and development team* and C3,*Lack of knowledge in problem domain*, are causes of UB2 *Development of software artifacts based on misunderstood requirements*, because fails in the communication between the director and the developers affect understand the requirements, which can be worsened by a lack of knowledge of the domain (e.g., concepts and rules). Regardig C3, It happens because Org X application domain is very specific (financial) and requires deep knowledge to properly understand all the concepts, requirements and rules. Thus, as the team is not expert in the domain, the lack of knowledge of the application domain (C3) causes inconsistencies. It is necessary that the developers knows the application domain to build a proper solution.

Regarding UB3 *Continuous Integration and Deployment techniques (e.g., Automated Test and Automated quality inspection) are not often applied to build software artifacts*, some members of the organization had previous experience with CSE practices in other companies, while

others have never had. Most of the members were not sure about CSE concepts and practices. Therefore, this undesirable behavior is caused by C4 *Organization's members had different experiences with CSE practices* and C5 *CSE concepts and practices are not well-known by the organization*.

Once the causes were identified, we defined strategies to implement data-driven software development by addressing such causes to leverage results. Table 25 presents the proposed strategies.

Table 25 – Strategies and Causes.

#	Strategies	Causes
S1	Define a new software development process that guides about prototyping and developing software products adopting CSE practices.	C1, C2, and C4
S2	Implement measurement culture to enable data-driven software development and improve process and product quality.	C4
S3	Define an on-boarding process to new members.	C3, C4, and C5
S4	Adopt a CSE common conceptualization.	C4 and C5
S5	Automate part of the development process by implementing CSE practices	C4 and C5
S6	Hire a CSE consultant or specialist.	C4 and C5

In order to provide useful information and support monitoring the defined strategies, information needs were identified. Table 26 presents some of them, related to the strategy S5.

Table 26 – Information needs identified based on the use of *California*.

#	Goal	Information Need	Description
4	Implement and improve CSE practices	(4.a) How many projects have adopted CI and CD practices (e.g., automatic testing, and automatic deployment)? (4.b) What is the rate of CI pipelines successfully completed? (4.c) What is the rate of CD pipelines successfully completed?	(4.a) contributes to understand if CSE practices have been adopted in the organization projects; while (4.b) and (4.c) help to understand if CI/CD pipelines have been successfully applied in the projects.

The set of information needs defined (tables 25 and 26) was discussed with the development team and director in a meeting and the ones to be addressed in this study were selected. The selection took into account mainly time constraints to perform the study in the context of this thesis. Thus, in this study the following information needs will be considered:

- (1.a) *What is the average waiting time for a user story from its creation until its development is initiated ?*
- (2.a) *What is the average time to successfully complete a task in a sprint*?
- (3.a) *Which types of code smells are produced in the software artifacts?*

- (3.b) *How many code smells are produced in the software artifacts?*
- (4.b) *What is the rate of CI pipelines successfully completed?*

6.3.3 Identifying the Available Sources

The following applications are used by Org X and were selected to provide data to meet the identified information needs. as an available source: Microsoft Azure DevOps, GitLab, and Sonar Cloud. Microsoft Azure DevOps provides data related to project management using Scrum, such as Project, Person, Development Team, User Story, Sprint, and Task. GitLab provides data about Source Repository, Software Artifacts, Commits, Branch, Developer that created or modified a Software Artifact and when a CI and Delivery pipeline was performed. Finally, Sonar Cloud provides data related to the quality of software artifacts created in software project. Thus, it stores data such as quality issues of a software artifact, the developer that created a software artifact and the developer that fixed a quality issue.

6.3.4 Defining Measures

Considering the available data and the identified information needs, the measures presented in Table 27 were defined.

Table 27 – Measures defined to meed the Information Needs.

#	Information Need	Measure	Description
(1.a)	What is the average waiting time for a user story from its creation until its development is initiated?	User Story Wait Time	The average time a user story has been waiting since it was created until its development starts in a sprint.
(2.a)	What is the average time to successfully complete a task?	Task Cycle Time in a project	The average time a task take to be completed in a project (from when it is started to when it is finished)
(3.a)/ (3.b)	Which types of code smells are produced in the software artifacts ?/How many code smells are produced in the software artifacts?	Number of Code Smells (per type)	Quantity of code smells (per type) identified in the software artifacts produced in a project.
(4.b)	What is the rate of CI pipelines successfully completed?	CI Pipeline Success Rate	Rate of CI pipeline that were performed with success in a project (i.e., reason between the number of successfully performed CI pipelines and the number of performed CI pipelines) .

6.3.5 Providing Integrated Data using *The Band*

After the meeting with the director and developers, we received the access credentials of Microsoft Azure DevOps, GitLab, and Sonar Cloud to extract data and provide integrated data by using *The Band*.

The *The Band* instance that we have implemented (see Section 5.5) considered the three applications used by Org X. That means that *The Band* is able to extract data from these applications and store it in the OBDRs. However, as Org X had information needs that were not met by the dashboards already implemented in *The Band*, we needed to add new data access components to collect data from the OBDRs, integrate it and show it in dashboards to Org X. To provide data to meet the information needs, we used concepts from OBDRS related to SRO, CIRO and, RSMO. Next, we illustrate some fragments of the produced dashboards. After the director and the development team access the dashboards, the researcher discussed with them some questions raised from the data.

Figure 72 presents a fragment of the dashboard that provides data to the measure *User Story Wait Time* related to the information need 1.a.

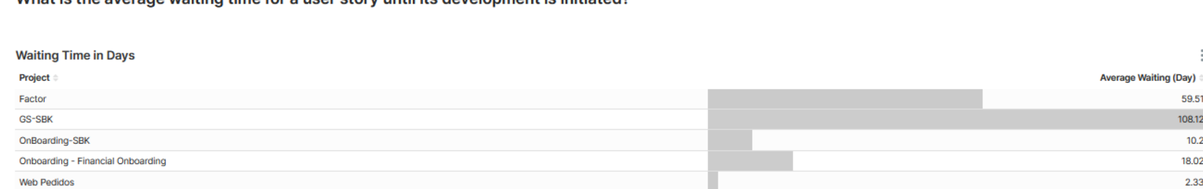


Figure 72 – Fragment of the dashboard focusing on User Story Wait Time.

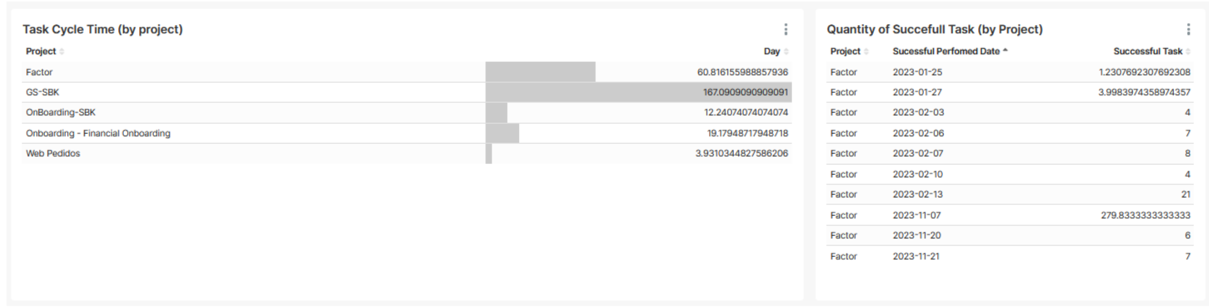
As can be observed in Figure 72, the user story average wait time in Org X projects varies from 2 days to 108 days. The director and the development team were questioned about that and, according to them, there were problems ranging from not understanding the requirements addressed in the user story, automatic generation of user stories and even not updating the conclusion of the user story in the project management application.

To understand a little more about this issue, another graph was provided to visualize the number of defined user stories over time (Figure 73). By analyzing the figure it is possible to visualize peaks and drops. The peaks refer to periods when a consultant worked at the organization and helped the team improve software engineering practices. The consultant gradually decreased its participation and the team abandoned practices that had been implemented (e.g., formally recording the user stories creation, start and end dates). In Figure 73 it is noted that in some projects there are periods in which no use story was recorded. The team commented that new features were developed, but the user stories were not recorded in the Microsoft Azure DevOps because the developers were overloaded.

Figure 74 represents data related to *Task Cycle Time (by Project)* and *Task Cycle Time*

Figure 73 – Fragment of the dashboard showing number of US (per type) defined over time.

(in an Project), which are related to the information need 2.a.



Task Cycle Time (by project)

Project	Day
Factor	60.816155988857936
GS-SBK	167.0909090909091
OnBoarding-SBK	12.24074074074074
Onboarding - Financial Onboarding	19.17948717948718
Web Pedidos	3.9310344827586206

Quantity of Succefull Task (by Project)

Project	Sucessful Perfomed Date	Successful Task
Factor	2023-01-25	1.2307692307692308
Factor	2023-01-27	3.9983974358974357
Factor	2023-02-03	4
Factor	2023-02-06	7
Factor	2023-02-07	8
Factor	2023-02-10	4
Factor	2023-02-13	21
Factor	2023-11-07	279.8333333333333
Factor	2023-11-20	6
Factor	2023-11-21	7

Figure 74 – Fragment of the dashboard showing the *Task Cycle Time* by the projects.

As can be observed in Figure 74, some projects (e.g., Factor and GS-SBK) have a *Task Cycle Time* superior to 30 days and, when we asked the development team "Why do tasks on average take more than 30 days to be completed successfully?" a developer reported that *"the team member forgot to update the conclusion of a task in the management application"*. He added that "*successfully completed tasks were recorded correctly when the consultant worked at Org X and again during the case study*.". It can be observed on *Task Cycle Time (an Project)* measure, that the tasks were launched and successfully completed in the period in which the consultant was in Org X (February and between September and December 2023). This showed that the development team needs to develop software management skills and improve conformance in following recording procedures.

Figure 75 presents data regarding code smells (information needs 3.a and 3.b and respective measures) .

When we asked the development team about the high quantity of the code smell *Function and methods should not be empty*, a developer commented that Org X developed an internal MDD solution that produces files with empty standard methods that can mostly be filled with business rules, if necessary. For example, file *signals.py* is an example of a file produced by the internal MDD solution that has 88 code smells of that type.

Regarding the information need 4.b (What is the rate of CI pipelines successfully completed), we developed two graphs to provide information related to it. The first one presents data about *CI Pipeline Success Rate* in a project (Figure 76). The second shows data about *CI*

Number of Code Smells (per type)

Code Smell	Quantity
Functions and methods should not be empty	265
Sections of code should not be commented out	94
String literals should not be duplicated	78
Comma operator should not be used	29
"Exception" and "BaseException" should not be raised	27
Multiline blocks should be enclosed in curly braces	26
Track uses of "TODO" tags	20
Method names should comply with a naming convention	19
Local variable and function parameter names should comply with a naming convention	18
Wildcard imports should not be used	18
Unused local variables should be removed	17

Number de Code Smells (per type and artifact)

Software Artifact	Code Smell	Quantity
recomb_social-entity:recomb_socialentity/signals.py	Functions and methods should not be empty	88
recomb_financial-agent:recomb_financialagent/signals.py	Functions and methods should not be empty	85
recomb_onboarding-business-backend:onboarding/signals.py	Functions and methods should not be empty	68
recomb_webpedidos-integration-webpedido-connect:index.html	Comma operator should not be used	29
recomb_webpedidos-integration-webpedido-connect:index.html	Multiline blocks should be enclosed in curly braces	26
recomb_webpedidos-websystem-webpedidos:apps/negociacao/api_views.py	Sections of code should not be commented out	14
recomb_onboarding-integration-wba:wba_integration/mapper.py	Method names should comply with a naming convention	13

Figure 75 – Fragment of the dashboard showing the Number of Code Smells (per type and artifact).

Pipeline Success Rate in all projects (Figure 77).

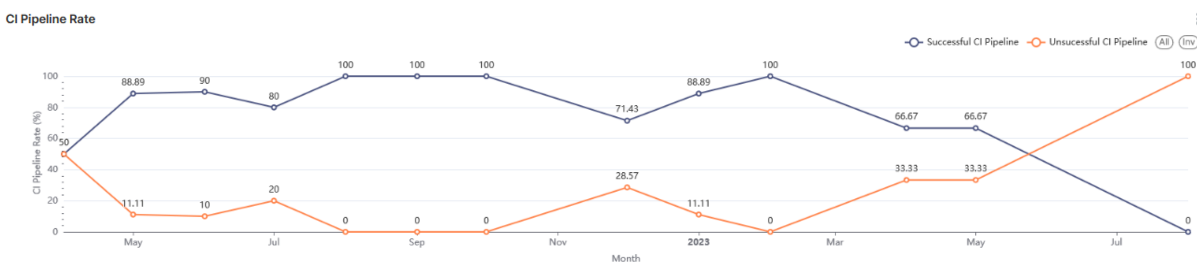


Figure 76 – Fragment of the dashboard showing data regrading CI Pipeline Success Rate in a project.

In Figure 76 it is possible to see the points in the development process in which there were problems in the CI pipeline. Using this graph, the development team can check if there were problems creating or maintaining a software artifact or even problems related to the configuration of the CI process, for example.

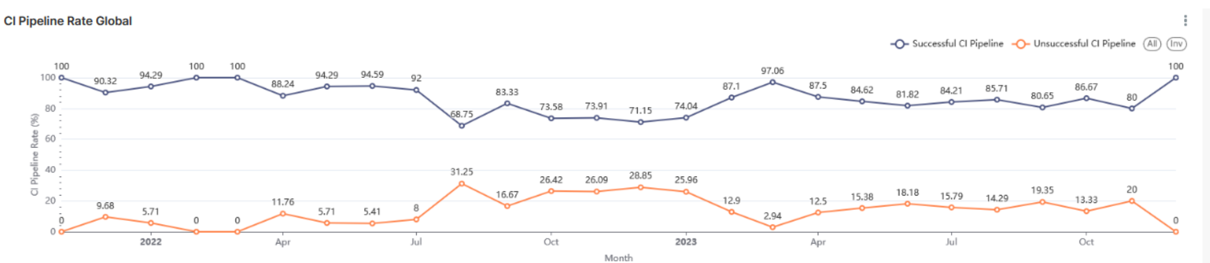


Figure 77 – Fragment of the dashboard focusing on CI Pipeline Success Rate in all projects.

As can be observed in Figure 77, most projects have a *CI Pipeline Success Rate* above 60%. However, although the *CI Pipeline Success Rate* is high, it is important to note that most projects do not implement automated tests, just code verification with Sonar Cloud.

It is worth pointing out that we faced some challenges in providing integrated data due to limitations in application data. For instance, Microsoft Azure DevOps, GitLab, and Sonar Cloud do not record some data we needed to provide meaningful integrated data. For example, Microsoft Azure DevOps does not provide data about the projects start and end date. Thus, it was necessary to create queries to estimate when the project started based on the

date of the first created user story and the first performed commit. Another challenge is data inconsistency in the applications. For example, the same project was recorded with different names in *Microsoft Azure DevOps* and in *GitLab*, making it difficult to identify data related to the same project. Moreover, sometimes, the same entity in the real world is represented by different entities in application data. For example, *Gitlab* allows the recording of the same team member more than once if we change the associated email. In some cases, data was missing. For example, in *GitLab*, it was common branch names and commit messages without reference to the Microsoft Azure DevOps user stories in which they originated.

To solve the aforementioned issues, we implemented in *The Band* a mechanism to align data of different OBDRs. To identify possible duplicate instances of an entity, we developed a script that implements a string matching (e.g., we used a script to find the instances of a *Person* with different names that represent the same person in the real world). After that, it was necessary to perform manual alignment to ensure data consistency.

6.3.6 Getting Feedback about *Immigrant*

After applying *Immigrant*, we conducted an interview with the director[7] to obtain his perception of the usefulness and feasibility of using our proposal. The interview lasted about 60 minutes. The interviewer talked briefly about each component of *Immigrant*, summarized how they were applied, and the main results produced when applying them. The study research questions were used as the main questions of the interview (i.e., we aimed to verify if *Immigrant* and each one of its components were useful and if their use was feasible). The interviewer explained the interview purpose to the director and asked the interviewee to talk freely about his perceptions of using each of the *Immigrant* components and the approach as a whole, pointing out the main advantages and disadvantages of using them. Next, we present some aspects pointed out by the director.

Regarding *Zeppelin*, the director found it useful and easy to use. He commented that an advantage of applying *Zeppelin* was that it provided a quantitative view of CSE practices adoption that, besides helping the organization reflect on how much and how well it has performed CSE practices, it supports investigating "*how much the development team knows about CSE practices that should be applied in Org X*". According to the director, the organization intends to use the panoramic view provided by *Zeppelin* to aid in identifying knowledge needs and offering training to improve the developers knowledge about CSE practices. Concerning disadvantages, the director commented that "*the questionnaire seems generic for all types of software organizations*" and, according to him, some practices may not be suitable (or realistic) for some types. The director questioned if it would be possible to create a version of *Zeppelin* focused on early-stage startups. The interviewer informed him that we intend to work on that as a future work.

[7] The development team was not available to be interviewed.

As for *California*, the director pointed out that the main strength is that "*it provides insight of causes and effects that are benefiting or causing problems in the development process. Knowing the causes of problems is important to create strategies that help remove or mitigate their effects*". The director said that he wants to propose a training path focusing on the financial domain because by analyzing information provided by *California*, "*the lack of developers knowledge of the financial domain became evident and it is a cause of several problems.*".The Director commented that he liked *California* and found it useful but he also found it complex to be replicated in the organization without a System Thinking and CSE specialist. It considered this the main weakness of *California*. The interviewer informed him that there is an ongoing research project creating guidelines and tool support to help other people use *California*.

About information needs identification, the director commented that" *Zeppelin and California provided data that enabled him to make questions about the software development process and the necessary data to improve the process and the team*". For example, he said that *"Knowing how long a US takes to be done is important when he is negotiating a contract with a client. And knowing which types of code smells a developer is "creating" it is important to promote training to improve the team"*. After this comment, he quoted the phrase from W. Edwards Deming[8]: "*What cannot be measured, cannot be managed*". He also asked "*whether there is a possibility of Zeppelin and California suggesting information needs based on the data analysis data*". The interviewer informed him that the current version of those Immigrant's components does not support that, but we can consider it as future work.

Concerning *The Band*, the director said that it was very useful. He said that "*it provided useful information about the software development process and development team that can be used to improve both of them*". He commented that "*from data provided by The Band, it was possible to note that some internal initiatives (e.g., Model-Driven Development (MDD) and Design Structure Matrix (DSM)) used to improve the development and project management processes have been successful*". It also highlighted: "*what I found most useful was that The Band provided realistic data that showed us what is happening in the organization*". According to him, understanding what is happening is important to identify what needs to be improved and take actions in that direction. He exemplified: "*For example, using the dashboard presenting data about the Number of Code Smells (per type), it is possible to visualize points of improvement in the MDD internal solution. For example, we can identify the software artifacts with a high number of code smells and who created them and we can investigate what happened (if the problem is in the applied method or if it was caused by the developer that created the artifact). Thus, we can improve practices to reduce code smells and even improve the developer's technical knowledge to avoid these problems.*". He also pointed out that data presented in *The Band* dashboards comes from data created by some action of a team member in projects carried out in the organization. Thus, *The Band* also helps identify problems in the process of collecting and recording such data. For example, it

[8] <https://en.wikipedia.org/wiki/W._Edwards_Deming>

was possible to identify that developers stopped recording user stories after the consulting left the organization, which hampered project planning. Knowing that also helps improve the software process.

The director did not face any difficulty in using *The Band*. It is important to notice that we refer to the use of the implemented solution, i.e., the use of the produced dashboards. In this study, the new components necessary for *The Band* to provide data to meet the identified information needs were developed by the author of this thesis. Thus, no effort was needed from the organization. The director questioned whether *The Band* would suggest points for improvement based on the integrated data: "*What actions should I take to improve the values of the measures? What is the most relevant measure? Where do I attack the problem?*". The interviewer clarified that *The Band* purpose is to provide data to *enable* data-driven software development. Thus, how data will be used and what are the actions to be taken are out of *The Band's* scope.

Finally, when asked about the combined use of the three components, the director said that it was useful because by using *Immigrant* he had a panoramic view of CSE practices in Org X, going from a global view (*Zeppelin* and *California*) to a more detailed view based on quantitative data (*The Band*). In this way, it was possible to envision a path to data-driven software development.

6.4 Discussion

The participative study case results and the director's feedback provide us with information about how the *Immigrant*'s components (*Zeppelin*, *California*, and *The Band*) work together and enable data-driven software development in a software organization.

Under the *Immigrant's* top-down perspective, *Zeppelin* and *California* offered an overview of Org X and helped identify information needs. *Zeppelin* provided a quantitative data view of the CSE practices adoption, while *California* supported understanding undesirable behaviors and their causes. By combining both components Org X defined a set of information needs to guide its first steps towards data-driven software development.

Another benefit brought by these Immigrant's components was the management of expectations about the quality of the application data in Org X. After all, the low level of knowledge and experience with CSE practices of the team members identified from *Zeppelin* data would be reflected in the use of the applications that support the development process and, consequently, in the data stored in the applications. This was confirmed when *The Band* was used to provide integrated data to meet the information needs. For example, by analyzing data related to *Task Cycle Time* it was noticed that the development team only recorded data in the Microsoft Azure DevOps when the consultant was working at Org X.

Concerning *Immigrant's* bottom-up perspective, *The Band* was useful to validate the CSE practices adoption overview provided by *Zeppelin* and *California*, as commented by the director. In this case study, it is possible to observe a relation between the quality of CSE practices adoption and the quality of application data, as commented before. Additionally, *The Band* allows the director to identify opportunities for improvement in specific areas of the organization (e.g., training some developers, to create less code smells and verify that some internal initiatives (MDD solution and DSM) were fruitful).

Although the challenges faced when implementing *Immigrant* with all components (e.g., it requires time to use all the components, particularly if new components need to be added to *The Band*), it was possible to observe that it was useful and its use is feasible to enable data-driven software development.

6.5 Threats to Validity

Next, we discuss some threats considering the classification proposed in (RUNESON et al., 2012). The definition of each validity type was presented in Chapter 4.

Regarding *Construct Validity*, some threats observed in the previous learning iterations, when we used *Immigrant* components in isolation, are also present in this study, such as the statements used to identify CSE practices in the *Zeppelin* diagnosis questionnaire (which may have led to misunderstandings) and the weights assigned by the researchers to the adoption levels are threats. To resolve misunderstandings, we performed interviews with the participants to validate the answers. This gave us an opportunity to calibrate the answers. Another threat is that using *California* may be tiresome and requires knowledge of the organization and system thinking tools. Thus, participants could provide incorrect or incomplete information. To minimize this threat we applied *California* after *Zeppelin*, which accelerated its use. Moreover, the researcher guided the participants through *California* steps and provided the necessary system-thinking knowledge to accomplish the tasks. Another threat is related to the quality of the available data. The successful use of *The Band* relies on the quality of application data. Therefore, the study results also depend on the quality of data available in the organization. *The Band* does not have any mechanism to ensure application data quality (this is out of its scope). If there is a lack of data, data is incorrect or incomplete, the quality of the integrated data will be affected and, as a consequence, the perceptions of the use of *The Band* as well. To minimize this threat, we assign semantics to data in *The Band* by using concepts from an ontology network (i.e., we extract data from the applications and transform it into instances of ontological concepts). Even so, it is not possible to guarantee data quality because we did not influence how data was created. The participation of the researcher in the study also threatens construct validity because as he is one of the creators of *Immigrant*, he knows what was addressed in *The Band* instance that was already implemented. This familiarity may

have influenced the decision on which information needs and the measures to define in the study (e.g., by directing to the ones that could be satisfied by the implemented instance of *The Band*). To minimize this threat, information needs and measures were defined together with the director and developers, supported by information provided when using *Zeppelin* and *California*. Another threat is that so far the results from using our approach are based on the perception of the director, collected through an interview. We still lack quantitative data to provide more robust evidence of the use of *Immigrant*.

Concerning *Internal Validity*, the main threat regards the time to carry out the study. We started working with Org X in September 2023 and we believed that we would have enough time to carry out the study without any hurry. However, the author of this thesis needed to drastically decrease the number of hours dedicated to the study and Org X also faced issues that delayed some activities. As a result, the use of *The Band* to extract data and provide integrated data was performed only in November 2023. As a consequence, it was not possible to implement the defined strategies to quantitatively monitor them by using data provided by *The Band*. The active participation of the author in the study is also a threat intrinsic to participative case studies. The previously discussed problems in the quality of application data also affect internal validity (different data could lead to different results in the same organization).

As for *External Validity*, the main threat is that *Immigrant* was applied in only one organization and the author of this thesis participated in the study. Moreover, the organization profile is very specific (a small three-years-old fintech startup). To generalize results, ideally, the proposal should be applied in other software organizations and also by someone other than the author of this work.

Concerning *Reliability Validity*, the main threat is the influence of the author in the study and the fact that data analysis was performed by the researchers. To minimize this threat, analysis was carried out by the author and discussed/refined together with the supervisors. Discussions were performed until consensus.

Considering all the aforementioned threats, the obtained results must be considered preliminary evidence that needs other studies to be confirmed.

6.6 What did we learn?

To apply *Immigrant* in OrgX, we started by using *Zeppelin* and, after that, we applied *California*. We learned that by doing that it was possible to understand the organization faster than when *California* was applied alone, as described in (SANTOS; BARCELLOS; CALHAU, 2020). It was possible to quickly identify the undesirable behaviours, define strategies, and information needs.

Another important lesson learned is that, when answering *Zeppelin* questionnaire,

people can misunderstand the statements and provide incorrect answers. Thus, conducting an interview helps understand how CSE practices have been performed in the organization, remove gaps in the understanding of the statements, and produce more realistic and accurate answers. As can be noted in the Table 20 and Table 21, the adoption degrees changed after the interview. This shows that *Zeppelin* can be improved to better support its users to properly understand the statements and the levels at which each practice can be performed.

We also learned that in *The Band* data integration is sensitive to the quality of application data. Moreover, quality of such data is directly related to the quality of the process that produce it and to the actions performed by the development team. Therefore, to improve the quality of data integration it is necessary to improve the development process and its execution by the development team. Performing a quality software process would contribute to provide standardized and quality data.

Finally, we learned that although there are some challenges in applying *Immigrant*, it is a promising approach to enable data-driven software development.

6.7 Final Considerations

This chapter presented a participative case study whose goal was to evaluate *Immigrant* in a real context to verify whether it is useful and whether its use is feasible. In a nutshell, initially, *Zeppelin* was used to gain a comprehensive understanding of the current CSE practices implemented in the organization. Subsequently, we employed *California* to pinpoint specific information requirements essential for the organization to formulate strategies enabling the implementation of a data-driven software process. From *Zeppelin* and *California* results, we identified information needs to guide data-driven software development. Last, we used *The Band* to deliver cohesive and integrated data to meet the identified information needs. Due to time constraints, it was not possible to implement and evaluate the strategies defined to Org X. Even so, *Immigrant* was considered useful by the director to enable data-driven software development. It provided useful data for the identification of leverage points in the current software development process, improvement of skills of the developers, and validation of internal initiatives, for instance.

In the study presented in this chapter, *Immigrant* was used in an organization that had information needs not addressed by the instance of *The Band* that was implemented. Thus, it was necessary to add new components to it, which demands development effort (the author developed such components). It is important to notice that an organization can use *The Band* as it is, i.e., considering the applications and data the current version is able to integrate. In this way, no development effort would be necessary.

Immigrant deals with a complex process (from the organization information needs to integrated data aiming data-driven software development). Thus, it is natural that some

challenges arise when applying the approach. These challenges reveal that there are several aspects involved in the *Immigrant* components that can be improved. Some of them are discussed in the next chapter, which presents the research contributions, limitations, and envisioned future works.

7 Final Considerations

Rest now within the peace. Take of the fruit, but guard the seed.

Led Zeppelin, Carouselambra

This chapter presents the final remarks and conclusions regarding the research presented in this thesis. Section 7.1, summarizes the main aspects of the work. Section 7.2 describes the produced contributions, relating them to the proposed objectives and the published papers. Section 7.3 discusses the limitations of the work. Finally, Section 7.4 presents recommendations for future improvements and further research.

7.1 Summary of the Research

This research addresses an ontology-based approach, called *Immigrant*, which uses networked ontologies to integrate application data aiming at enabling data-driven software development in the CSE context. *Immigrant* considers the organization's information needs and the available data to provide meaningful data integration to support CSE. It is composed of the following components: (i) *California*, (ii) *Zeppelin*, and (iii) *The Band*. *California* and *Zeppelin* are used to understand the organization and identify its information needs, while *The Band* is used to provide ontology-based integrated data to meet the identified information needs. *The Band* uses the ontologies in the *Continuum* ontology (sub)network that has been proposed as part of this thesis in alignment with SEON (RUY et al., 2016).

As previously discussed, this research was motivated by the fact that in the last years it has been noticed that for producing products that properly meet customers' needs, making well-informed decisions, and identifying business opportunities, new practices should be combined with agile development to enable continuous actions of planning, building, operation, and evaluation (FITZGERALD; STOL, 2017; BARCELLOS et al., 2022). Hence, organizations should evolve to continuous and data-driven development in a Continuous Software Engineering (CSE) approach (BOSCH, 2014). CSE consists of a set of practices and tools that supports a holistic view of software development to make it faster, iterative, integrated, continuous, and aligned with the business (FITZGERALD; STOL, 2017; BARCELLOS et al., 2022). Software organizations often use different applications to support different aspects of software development. However, many of them do not use data present in the applications to support data-driven software development (SANTOS et al., 2022).

One of the reasons for that is the difficulty in accessing, integrating, analyzing, and viewing data handled by heterogeneous applications (CALHAU; FALBO, 2010). This difficulty

can result in semantic conflicts whenever divergent interpretations are given to the same information item, a situation that may not even be detected (WACHE et al., 2001). Ontologies can be used to deal with these issues in semantic integration initiatives (NARDI; FALBO; ALMEIDA, 2013). They establish a common conceptualization about the applications subject domains and support communication and integration.

In this work, we propose to use networked ontologies to establish an integration architecture and assign semantics to application data. Moreover, we advocate that integrated data should be aligned to the organization information needs. The work followed the Design Science Research paradigm and involved four learning iterations (BARCELLOS et al., 2022).

In order to understand how ontologies can be used to integrate data on the CSE context, in the first learning iteration we explored the use of an ontology on agile development with Scrum to integrate software development data spread across applications and thereby support data-driven software development. It was the first *Immigrant* and *The Band* version. The results were promising; by using the proposed approach and the resulting integrated solution, improved estimates and product quality were reported in a Brazilian public software development unit (SANTOS et al., 2021).

With this study, we confirmed in practice the necessity of properly identifying the organization's information needs to guide data integration and we noticed that organizations may face difficulties in identifying such information. Moreover, we observed that understanding how the organization works and the CSE practices it adopts is useful to identify the organization's information needs. Thus, we performed the second learning iteration, a case study in a Brazilian software house to understand this phenomenon (SANTOS; BARCELLOS; CALHAU, 2020; SANTOS; BARCELLOS; CALHAU, 2022). By using a system-thinking-based process (*California*) it was possible to define some information needs related to strategies defined to address causes of undesirable behaviors.

Although *California* can be useful in this matter, it may demand knowledge and time to be applied. Thus, we performed the third learning iteration, a multiple case study with five Brazilian software organizations to evaluate a diagnostic instrument, called *Zeppelin*, to support organizations to get a panoramic view of CSE adoption (SANTOS; BARCELLOS; RUY, 2021). The results indicate that *Zeppelin* provides a panoramic view that describes the current state of the adopted CSE practices in an organization, helping to define its information needs.

Despite the results of the first learning iteration being promising, data from applications addressing certain CSE processes, such as continuous integration and continuous deployment, was not included in the first version of the proposed solution. Aiming to build a more comprehensive data integration solution that covers several CSE processes, we decided to expand the scope of integrated data using an ontology network. Thus, we developed *Continuum*, an ontology network that addresses CSE aspects and is integrated into the Software Engineering Ontology Network (SEON) (RUY et al., 2016). The current version of *Continuum* is composed

of the Scrum Reference Ontology (SRO), Continuous Integration Reference Ontology (CIRO), and Continuous Deployment Reference Ontology (CDRO). Thus, we evolved the software solution proposed in (SANTOS et al., 2021) to a data integration solution based on the concepts of ontology network and Federated Information System (FIS), called *The Band* (SANTOS; ALMEIDA; BARCELLOS, 2023). *The Band* leverages *Continuum* as the basis to build reusable and autonomous software components, and organize them in a FIS architecture in which all components work together in the federation. As a result, data from different applications can be integrated and visualized to provide meaningful information to aid in software development activities and decision-making.

To evaluate *Immigrant*'s usefulness and practical feasibility, we performed the fourth learning iteration, a case study case in a software organization where *Zeppelin* and *California* were used to help identify information needs and *The Band* was used to provide integrated data from Microsoft Azure DevOps, GitLab, and Sonar Cloud to meet the information needs. The results provide preliminary evidence that *Immigrant* is useful and its use in a practical setting is feasible.

7.2 Research Contributions

This work is related to two main research areas: Continuous Software Engineering (CSE) and Ontologies. We see its general objective – *to propose an ontology-based approach that uses networked ontologies to integrate application data aiming at enabling data-driven software development in the CSE context* – as an application of ontologies to solve data integration problems in CSE. We enumerate here the contributions, spanning the two research areas:

- **California**, a *Systems Theory-based process* that helps to understand how the organization behaves and to define proper strategies to implement or improve CSE practices according to the organization's behavior and needs (SANTOS; BARCELLOS; CALHAU, 2020; SANTOS; BARCELLOS; CALHAU, 2022). *California* provides means to (i) understand how different organizational aspects are interrelated, (ii) create strategies to change undesirable behaviors, (iii) prioritize the undesirable behaviors to be changed first, (iv) create a common communication among project stakeholders, and (v) address undesirable behaviors by applying CSE practices and concepts. In the *Immigrant* context, it can be used to aid the identification of the organization's information needs to be met by integrated data. This contribution is related to the specific objective SO1 (*Establish mechanisms to identify the organization information needs*) of this research.
- **Zeppelin**, a *diagnostic instrument* that provides a panoramic view of the current state of the adoption of CSE practices in an organization, supporting the identification of weaknesses and strengths as well as aiding in decision-making about which aspects

should be addressed in improvement actions (SANTOS; BARCELLOS; RUY, 2021). In the *Immigrant* context, it can be used to aid the identification of information needs and data sources that will provide data to meet the information needs.

- **Continuum**, an *ontology (sub)network* that aims at representing the conceptualization related to the processes involved in CSE. In this work, the following processes are considered: Agile Development, Continuous Integration, Continuous Delivery and Continuous Deployment. Currently, *Continuum* comprises the *Scrum Reference Ontology* (SRO) (SANTOS et al., 2021), the *Continuous Integration Reference Ontology* (CIRO), and *Continuous Deployment Reference Ontology* (CDRO). In the SE big picture, CSE appears as a (large) subdomain involving other subdomains. Thus, *Continuum* has been developed as a subnetwork of SEON (RUY et al., 2016). In the *Immigrant* context, *Continuum* establishes a comprehensive conceptualization that can serve as an interlingua to assign semantics and allow the integration of data from different applications. *Continuum* can also be used to support other ontology applications, such as communication and learning, standards harmonization, semantic documentation, and among others. This contribution is related to the specific objective SO2 (*Develop networked ontologies about CSE subdomains*) of this research.

- **The Band**, an *integration software solution* that uses networked ontologies and Federated Information System architectures to provide integrated data. *The Band* provided means to extract, store data from applications and provide integrated data without semantic issues. It enables *Immigrant* to meet requirements R2 (*the approach must address semantic issues involved in data integration in such a complex domain*), R3 (*the approach must consider data available in the organization's applications*) and R4 (*the approach must provide integrated and meaningful data, considering the organization's information needs and available data*). It is related to SO3 (*Create an ontology-based approach to integrate application data*), because it is the *Immigrant* component responsible for data integration. *The Band* was developed following the process we have proposed called *The Journey*, employing model-driven techniques that leverage information present in networked ontologies to create data integration components. We believe that *The Journey* and its elements (model transformations, for example) may also be used in other settings; a hypothesis which may be further examined in future work.

- And, finally, **Immigrant**, an *ontology-based approach* that uses networked ontologies from *Continuum* to integrate application data aiming at enabling data-driven software development in CSE context. *Immigrant* considers the organization's information needs and the available data to provide integrated data in dashboards. It is composed of *California*, *Zeppelin* and *The Band*, and is related to the specific objective SO3 (*Create an ontology-based approach to integrate application data*).

These results contribute to fulfilling the established objectives of this work. Table 28 relates the presented contributions to the specific objectives of this thesis.

Table 28 – Contributions versus Specific Objectives.

General Objective	Specific Objective	Contribution
An ontology-based approach that uses networked ontologies to integrate application data aiming at enabling data-driven software development in the CSE context	*SO1. Establish mechanisms to identify an organization's information needs*	*California*, a *Systems Theory-based process* that helps understand how the organization behaves and defines proper strategies to implement or improve CSE practices, and *Zeppelin*, a diagnostic instrument that provides a panoramic view of the current state of the adoption of CSE practices in an organization.
	SO2. Develop networked ontologies on CSE subdomains	*Continuum*, an ON that aims at representing the conceptualization related to the processes involved in CSE.
	SO3. *Create an ontology-based approach to integrate application data*	*The Band*, the *Immigrant* component responsible for data integration. It is an ontology-based approach that uses networked ontologies from *Continuum* to integrate application data.
Immigrant, the proposed solution to enable data-driven software development in CSE context.		
	SO4. Apply the proposed approach in a real context	Fourth studies (the fourth learning iterations) performed with software organizations involving the *Immigrant* components.

The contributions cited above and others produced along this work were published in the following papers (in chronological order):

i. P. S. Santos Jr, M. P. Barcellos, and R. F. Calhau, *"Am I Going to Heaven? First Step Climbing the Stairway to Heaven Model – Results from a Case Study in Industry"*, in 34th Brazilian Symposium on Software Engineering (SBES 2020), 2020, p. 309–318, <http://dx.doi.org/10.1145/3422392.3422406>. *Presents the exploratory study that resulted in California. Best Paper Award.*

ii. P. S. Santos Jr, M. P. Barcellos, and F. Ruy, *"Tell me: Am I going to Heaven? A Diagnosis Instrument of Continuous Software Engineering Practices Adoption"*, in 25th International Conference on Evaluation and Assessment in Software Engineering (EASE 2021), 2021,

<https://doi.org/10.1145/3463274.3463324>. *Presents Zeppelin and the multiple case studies performed to evaluate it.*

iii. P. S. Santos Jr, M. P. Barcellos, R. A. Falbo, and J. P. A. Almeida, *"From a Scrum Reference Ontology to the Integration of Applications for Data-Driven Software Development"*, in Information and Software Technology, vol. 136, 2021, <https://doi.org/10.1016/j.infsof.2021.106570>. *Presents the first version of Immigrant (focusing on The Band).*

iv. P. S. Santos Jr, M. P. Barcellos, and J. P. A. Almeida, *"An Ontology-based Approach to Enable Data-Driven Decision-Making in Agile Software Organizations"*, in V Doctoral and Masters Consortium on Ontologies (WTDO 2021), Ontobras 2021, 2021. <https://ceur-ws.org/Vol-3050/>. *Provides an overview of the Doctoral research project.*

v. P. S. Santos Jr, M. P. Barcellos, and R. F. Calhau, *"Am I Going to Heaven? First Step Climbing the Stairway to Heaven Model – Results from a Case Study in Industry"*, Journal of Software Engineering Research and Development, 2022, vol. 10, <https://doi.org/10.5753/jserd.2021.1992>. *Extended version of the paper published in SBES 2020.*

vi. P. S. Santos Jr, M. P. Barcellos, F. B. Ruy, and M. S. Omêna, *"Flying over Brazilian Organizations with Zeppelin: A Preliminary Panoramic Picture of Continuous Software Engineering"*, In Proceedings of the XXXVI Brazilian Symposium on Software Engineering (SBES '22), 2022, p. 279–288, <https://doi.org/10.1145/3555228.3555234>. *Presents a survey investigating the adoption of CSE practices in 28 Brazilian organizations. Distinguished Paper Award.*

vii. P. S. Santos Jr, J. P. A. Almeida, and M. P. Barcellos. 2023. *"Towards Federated Ontology-Driven Data Integration in Continuous Software Engineering"*. In XXXVII Brazilian Symposium on Software Engineering (SBES 2023), September 25–29, 2023, Campo Grande, Brazil. <https://doi.org/10.1145/3613372.3613380>. *Presents an overview of the current version of The Band.*

viii. C. E. Correa Braga, P. S. Santos Jr, and M. P. Barcellos. 2023. *Help! I need somebody. A Mapping Study about Expert Identification in Software Development.* In XXXVII Brazilian Symposium on Software Engineering (SBES 2023), September 25–29, 023, Campo Grande - MS, Brazil. <https://doi.org/10.1145/3613372.3613389>. *Collaboration with a master's student to investigate how experts have been identified in software development. The student intends to use The Band to integrate data regrading results produced by developers to help identify "who knows what" in an organization.*

Finally, the following paper is currently under review:

- P. S. dos Santos Jr, M. P. Barcellos, F. B. Ruy, and M. S. Omêna, *"Preliminary Panoramic View of Continuous Software Engineering Adoption in Brazilian Organizations"*, *Submitted*

to Journal of Software Engineering Research and Development. Extended version of the paper published in SBES 2022.

In Chapter 1 we presented the hypothesis established at the beggining of this work as follows:

The use of ontologies in an ontology network that addresses Continuous Software Engineering (CSE) aspects facilitates the semantic integration of data stored in diverse applications, and can thereby enable data-driven software development.

The artifacts produced in this work, particularly *Continnum* and *The Band*, along with the studies results have shown that using *Continnum* helped us assign semantics to application data and create an ontology-based FIS architecture (*The Band*) to integrate data and support data-driven software development. This corroborates our hypothesis [1].

7.3 Research Limitations

Like any research, this work has limitations. Throughout the chapters, we discussed some specific limitations. Here, we summarize them and discuss overall caveats that should be considered in the use of the proposed artifacts.

- **California**: although the proposed process is useful, it involves a lot of tacit knowledge and judgment, as well as knowledge of *System Thinking tools*, *GUT matrix*, and *Reference ontologies*. Moreover, it may demand much time to be applied. Hence, depending on the application scenario to be considered, it may be difficult or even unfeasible to use it fully.
- **Zeppelin**: the proposed instrument relies on a questionnaire to create a panorama of the adoption of CSE practices in an organization; as such, this overview is based on the perception and knowledge of those who answered the questionnaire. Therefore, this can provide a biased view of a software organization. In the fourth learning iteration, we noticed divergences between the answers provided by the developers and the observed organizational reality. As a means to mitigate these divergences, interviews (or other forms of raising evidence) concerning the adoption of the various CSE practices may be required. Beyond biases, the divergences may also be the result of different interpretations of the statements contained in the questionnaire.
- **Continuum**: up until now, the proposed network's development and growth have been done exclusively by our group. It is necessary to allow other researchers to contribute to *Continuum*. Besides that, the Continuous Deployment and Delivery domains needs to be more explored in further depth.

[1] This conclusion is based on the researchers' experience in this work and on the studies results. We did not run

- **The Band**: the proposed solution has a modular and complex data integration architecture based on a Federated Information System and ontology networks. Therefore, knowledge about *Continuum*'s subontologies is required to fully grasp the integrated data provided by *The Band*. Another limitation is related to the quality data issue of data stored in the applications that support the development process. This may negatively affect the integration and exchange of data between *The Band* and applications.

- **Immigrant**: the proposed approach considers both *top-down* and *bottom-up* perspectives. Therefore, it is necessary that whoever is applying it has the ability to look at the organization holistically and navigate between the different organizational levels in a fluid way and connect the different organizational elements (e.g., process, good practices, and data), to understand how the organization implements CSE. While there are various supporting elements in the proposed approach, these may be considered heavy requirements for the full application of the approach. It is also worth say that although we have carried out several studies in this work, it is still necessary to apply Immigrant in other practical settings. The results obtained so far are promising, but they are not robust enough to allow for generalization. Thus, new studies are necessary to better understand Immigrant strengths and weaknesses.

7.4 Perspectives of Future Works

The results presented in this thesis establish the basis for several future works. We believe that these works can solve some of the aforementioned limitations. Others are improvements for the work done and contributions in the context of the research lines explored in this thesis. Following, we present some possible future works. The list focuses on the main improvement opportunities we identified, but it is not exhaustive.

- **California**: to have a better understanding of how the different software engineering aspects (i.e., variables) are connected to and influence each other in an organization, we propose to study other *Systems Thinking theory* tools (e.g., Behavior over Time Graphs, Dynamic System Model, and Iceberg Model) and combine them with *Enterprise Architecture Models*, so that it will be possible to connect system variables, undesirable behaviors, and causes to elements of the organization architecture. Regarding undesirable behaviors, it is difficult to identify them and relate them to *archetypes* (e.g., Limits to Success, Shifting the Burden, Fixes that Fail, and Success to the Successful) and *leverage points*. Thus, an future work is defining guidelines to help their identification. For example, define a guideline that teaches how create a systemic models using concepts of CSE. Finally, *California* needs clear guidance and computational support. These aspects have

hypothesis testing.

been addressed in works developed by other students of our research group.

- **Zeppelin**: from the performed studies, we noticed that some statements raised doubts in those who answered the questionnaire. Therefore, it is necessary to improve *Zeppelin* by providing detailed information about each statement/practice to ensure its proper understanding. It is also possible to evolve *Zeppelin* to better address some CSE aspects, such as Continuous Operation and Continuous Use. CSE involves several processes and some of them are not addressed by *Zeppelin*. From the studies we also realize that just one person answering *Zeppelin* can give a biased perspective on how the organization implements CSE practices. Therefore, it would be interesting to create a way for more than one person in the organization to fill out the *Zeppelin* questionnaire or create a guideline that allow us to identify and minimize the biased perspective and, thus, present an analytical report of this shared vision. Furthermore, an important challenge when using *Zeppelin* is to verify whether the answers to the statements are in accordance with the organization's reality. It would be positive to make such verification without the need for an interview or asking the organization to send proof artifacts. Moreover, it is necessary to calibrate the weights defined to the answers provided to each statement. In a recent collaboration with researchers from the Laboratory for Software Engineering and Reliability (LASER) at UNICAMP, it was noticed that the weights may differ depending on the organization type (e.g., Startup, Software House, and Organization with IT Departure). Furthermore, as different organization types may perform different CSE practices, it can be necessary to create a different version of *Zeppelin* for each type of organization. Last, it would be useful to provide guidelines on how to implement the CSE practices addressed in *Zeppelin* (this has been addressed in the work of a master's student of LASER).

- **Continuum**: the *Continuum* body of ontologies may be enlarged, allowing better coverage of the CSE domain. Ontologies covering other subdomains should be developed and added to *Continuum*, e.g., concerning Continuous Planning, Continuous Use, Continuous Trust. Further, we believe that the body of ontologies composing *Continuum* can be applied to CSE learning. CSE disciplines can be better understood with the support of consolidated conceptual models such as those that are part of *Continuum*, potentially complementing CSE teaching. Teachers can adopt ontologies as complementary material in their disciplines, present the graphical model and use the conceptualization provided to minimize students' difficulties in understanding the domain, use instantiations to demonstrate their real functioning and create new instantiations to stimulate students' thinking. An example of how *Continuum* was used to teach concepts from the CSE domain in an organization was presented in (SANTOS; BARCELLOS; CALHAU, 2022).

- **The Journey**: can be characterized as an *ontology-driven software development process* (PAN et al., 2012) that uses ontologies' concepts to produce data integration software artifacts (e.g., database, webservices, and code libraries). However, no in-depth study has

been carried out on the characteristics of ontology-driven software development process literature. Therefore, it would be interesting to carry out a literature review on that topic to improve the use of ontologies in *The Journey*. Although some code generation tools based on MDD techniques have been developed, there are still software artifacts (e.g., dashboards and data view) generated during *The Journey* that the tools do not address. Further automating the process may be an interesting line of work.

- **The Band**: like any data integration solution, *The Band* deals with data quality issues which can influence the results. Therefore, it is necessary to create mechanisms for *The Band* to identify and fix (when possible) data quality issues, as well as indicate in which part of the software process data with quality problems is produced. *The Band* was developed to integrate data in CSE context. However, other domains can take advantage of this data integration solution to support data-driven decision-making. For example, in the context of a master research project developed by a student of our research group, *The Band* is being used on integration of application data and, thus, provide information useful to identify team member‘s skills to allocate people in teams. Therefore, studies can be performed to investigate how other domains can take advantage of *The Band*.

- **Immigrant**: Finally, it is important to comment that all the future work mentioned above has a positive impact on the Immigrant evolution, as they are related to the *Immigrant*'s components. In addition to the mentioned works, we believe that it is necessary to detail the process of using *Immigrant* by providing clear guidelines to support a software engineer applying it in a software organization. Currently, the process contains four activities that can be detailed to better support third-part use of *Immigrant*.

Bibliography

ALMEIDA, J. P. A. et al. *gUFO: A Lightweight Implementation of the Unified Foundational Ontology (UFO)*. 2020. Disponível em: <http://purl.org/nemo/gufo>. Cited on page 46.

ALMEIDA, J. P. A.; FALBO, R. A.; GUIZZARDI, G. Events as entities in ontology-driven conceptual modeling. In: SPRINGER. *Conceptual Modeling: 38th International Conference, ER 2019, Salvador, Brazil, November 4–7, 2019, Proceedings 38*. [S.l.], 2019. p. 469–483. Cited on page 45.

AMBLER, S. W.; LINES, M. *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. 1st. ed. [S.l.]: IBM Press, 2012. ISBN 0132810131. Cited on page 34.

ATLASSIAN. *Bitbucket*. 2023. Disponível em: <https://www.atlassian.com/br/git/tutorials/using-branches>. Cited 2 times on the pages 10 and 40.

AYED, H.; VANDEROSE, B.; HABRA, N. A metamodel-based approach for customizing and assessing agile methods. In: *2012 Eighth International Conference on the Quality of Information and Communications Technology*. [S.l.: s.n.], 2012. p. 66–74. Cited on page 102.

BARCELLOS, M. P. Towards a framework for continuous software engineering. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 626–631. ISBN 9781450387538. Disponível em: <https://doi.org/10.1145/3422392.3422469>. Cited 6 times on the pages 10, 19, 31, 33, 34, and 126.

BARCELLOS, M. P.; FALBO, R. de A.; FRAUCHES, V. Towards a measurement ontology pattern language. In: *ONTO.COM/ODISE@FOIS*. [S.l.: s.n.], 2014. Cited on page 48.

BARCELLOS, M. P. et al. Organizing empirical studies as learning iterations in design science research projects. *Simpósio Brasileiro de Qualidade de Software (SBQS)*, Nob. 2022. Cited 9 times on the pages 10, 11, 25, 29, 106, 107, 129, 180, and 181.

BASKERVILLE, R. *What design science is not*. [S.l.]: Springer, 2008. 441–443 p. Cited on page 106.

BASKERVILLE, R. L. Distinguishing action research from participative case studies. *Journal of systems and information technology*, MCB UP Ltd, 1997. Cited 5 times on the pages 114, 116, 123, 130, and 158.

BECK, K. *Extreme Programming Explained: Embrace Change*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201616416. Cited on page 115.

BECK, K. *Extreme programming explained: embrace change*. [S.l.]: addison-wesley professional, 2000. Cited on page 20.

BERCLAZ, D. *8 Deployment Strategies Explained and Compared*. 2023. Disponível em: <https://www.apwide.com/8-deployment-strategies-explained-and-compared/>. Cited on page 43.

BOSCH, J. Continuous software engineering: An introduction. In: *Continuous software engineering*. [S.l.]: Springer, 2014. p. 3–13. Cited 4 times on the pages 19, 20, 35, and 180.

BRANK, J.; GROBELNIK, M.; MLADENIC, D. A survey of ontology evaluation techniques. In: CITESEER LJUBLJANA SLOVENIA. *Proceedings of the conference on data mining and data warehouses (SiKDD 2005)*. [S.l.], 2005. p. 166–170. Cited 3 times on the pages 73, 91, and 99.

BRINGUENTE, A.; FALBO, R.; GUIZZARDI, G. Using a foundational ontology for reengineering a software process ontology. *Journal of Information and Data Management,*, v. 2, p. 511–526, 01 2011. Cited 2 times on the pages 48 and 104.

BRYNJOLFSSON, E.; HITT, L. M.; KIM, H. H. Strength in numbers: How does data-driven decision making affect firm performance? *O&M: Decision-Making in Organizations eJournal*, 2011. Cited on page 20.

BUSSE, S. et al. *Federated Information Systems: Concepts, Terminology and Architectures*. [S.l.], 1999. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.5830>. Cited 6 times on the pages 10, 58, 59, 149, 150, and 151.

CALHAU, R. F.; AZEVEDO, C. L. B.; ALMEIDA, J. P. A. Towards ontology-based competence modeling in enterprise architecture. In: *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*. [S.l.: s.n.], 2021. p. 71–81. Cited on page 46.

CALHAU, R. F.; FALBO, R. d. A. An Ontology-based Approach for Semantic Integration. In: *Proceedings 14th IEEE International Enterprise Distributed Object Computing Conference*. [S.l.]: IEEE Computer Society, 2010. p. 111–120. Cited 7 times on the pages 21, 22, 23, 58, 137, 153, and 180.

CALHAU, R. F.; FALBO, R. D. A. A Configuration Management Task Ontology for Semantic Integration. In: *Proceedings of the 2012 ACM Symposium on Applied Computing*. [S.l.: s.n.], 2012. Cited 3 times on the pages 48, 79, and 104.

CALVACHE, C. et al. A reference ontology for harmonizing process-reference models. *Revista Facultad de Ingeniería*, v. 1, p. 29–42, 12 2014. Cited 2 times on the pages 103 and 104.

CARRARETTO, R. *Separating Ontological and Informational Concerns: A Model-driven Approach for Conceptual Modeling*. Dissertação (Mestrado) — Universidade Federal do Espírito Santo, 2012. Cited 2 times on the pages 110 and 136.

CHEN, L. Continuous delivery: Huge benefits, but challenges too. *IEEE software*, IEEE, v. 32, n. 2, p. 50–54, 2015. Cited on page 41.

COHN, M. *Succeeding with agile: software development using Scrum*. [S.l.]: Pearson Education, 2010. Cited 2 times on the pages 63 and 70.

CONSERVANCY, S. F. *Git*. 2023. Disponível em: <https://git-scm.com/site>. Cited 6 times on the pages 10, 37, 38, 39, 40, and 80.

COSSENTINO, M. et al. A holonic metamodel for agent-oriented analysis and design. In: . [S.l.: s.n.], 2007. ISBN 978-3-540-74478-8. Cited on page 102.

COSTA, S. D. et al. A core ontology on the human–computer interaction phenomenon. *Data & Knowledge Engineering*, v. 138, p. 101977, 2022. ISSN 0169-023X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0169023X21000951>. Cited 2 times on the pages 48 and 104.

CUBRANIC, D. et al. Hipikat: a project memory for software development. *IEEE Transactions on Software Engineering*, v. 31, n. 6, p. 446–465, 2005. Cited 2 times on the pages 22 and 154.

DAMIANI, E. et al. A metamodel for modeling and measuring scrum development process. In: CONCAS, G. et al. (Ed.). *Agile Processes in Software Engineering and Extreme Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 74–83. ISBN 978-3-540-73101-6. Cited on page 102.

DEAN, L. Safe® 4.0 reference guide: Scaled agile framework® for lean software and systems engineering. *Google Scholar Google Scholar Digital Library Digital Library*, 2016. Cited on page 118.

DEBOIS, P. et al. Devops: A software revolution in the making. *Journal of Information Technology Management*, v. 24, n. 8, p. 3–39, 2011. Cited on page 20.

DESTEFANIS, G. et al. Software development: do good manners matter? *PeerJ Computer Science*, PeerJ Inc., v. 2, p. e73, 2016. Cited on page 154.

DRESCH, A.; LACERDA, D. P.; JÚNIOR, J. A. V. A. *Design science research: método de pesquisa para avanço da ciência e tecnologia.* [S.l.]: Bookman Editora, 2015. Cited on page 108.

DUARTE, B. B. et al. Towards an Ontology of Software Defects, Errors and Failures. In: *Conceptual Modeling.* [S.l.]: Springer-Verlag, 2018. Cited 3 times on the pages 48, 56, and 104.

DUARTE, B. B. et al. Ontological foundations for software requirements with a focus on requirements at runtime. *APPLIED ONTOLOGY (ONLINE)*, v. 13, p. 73–105, 2018. ISSN 18758533. Cited 2 times on the pages 48 and 104.

DURIEUX, T. et al. Empirical study of restarted and flaky builds on travis ci. In: *Proceedings of the 17th International Conference on Mining Software Repositories.* New York, NY, USA: Association for Computing Machinery, 2020. (MSR '20), p. 254–264. ISBN 9781450375177. Disponível em: <https://doi.org/10.1145/3379597.3387460>. Cited on page 102.

DUVALL, P. M.; MATYAS, S.; GLOVER, A. *Continuous integration: improving software quality and reducing risk.* [S.l.]: Pearson Education, 2007. Cited 7 times on the pages 10, 36, 37, 79, 82, 83, and 87.

DYBÅ, T.; DINGSØYR, T. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, v. 50, n. 9, p. 833–859, 2008. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584908000256>. Cited on page 34.

FALBO, R. SABiO: Systematic approach for building ontologies. *CEUR Workshop Proceedings*, v. 1301, 01 2014. Cited 7 times on the pages 44, 48, 62, 73, 91, 99, and 104.

FALBO, R. D. A.; NARDI, J. C. Evolving a Software Requirements Ontology. In: *Anales de XXXIV Conferencia Latinoamericana de Informática.* [S.l.: s.n.], 2008. p. 300–309. Cited 2 times on the pages 48 and 104.

FENSEL, D. et al. Introduction: What is a knowledge graph? In: ____. *Knowledge Graphs: Methodology, Tools and Selected Use Cases.* Cham: Springer International Publishing, 2020. p. 1–10. ISBN 978-3-030-37439-6. Disponível em: <https://doi.org/10.1007/978-3-030-37439-6_1>. Cited on page 102.

FIELDING, R. T.; TAYLOR, R. N. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, ACM New York, NY, USA, v. 2, n. 2, p. 115–150, 2002. Cited on page 138.

FITZGERALD, B.; STOL, K.-J. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, v. 123, p. 176–189, 2017. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121215001430>. Cited 11 times on the pages 10, 19, 20, 31, 32, 33, 36, 41, 126, 129, and 180.

FONSECA, V.; BARCELLOS, M.; FALBO, R. An ontology-based approach for integrating tools supporting the software measurement process. *Science of Computer Programming*, v. 135, 10 2016. Cited on page 58.

FONSECA, V.; BARCELLOS, M.; FALBO, R. d. A. An ontology-based approach for integrating tools supporting the software measurement process. *Science of Computer Programming*, v. 135, p. 20–44, 2017. ISSN 0167-6423. Special Issue on Advances in Software Measurement. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642316301599>. Cited 4 times on the pages 21, 22, 23, and 153.

FOWLER, M. *Continuous Integration*. 2006. <https://www.martinfowler.com/articles/continuousIntegration.html>. [Online; accessed 05-April-2022]. Cited on page 79.

FOWLER, M. *Continuous Integration.(2006) http://www.martinfowler. com/articles/continuousIntegration.html*. [S.l.]: Stand, 2011. Cited on page 36.

FOWLER, M. *Patterns of Enterprise Application Architecture: Pattern Enterprise Applica Arch.* [S.l.]: Addison-Wesley, 2012. Cited 3 times on the pages 141, 144, and 145.

FOWLER, M. *Canary Release*. 2014. Disponível em: <https://martinfowler.com/bliki/CanaryRelease.html>. Cited 2 times on the pages 42 and 43.

FOWLER, M. *Feature Toggles: What Are They and How to Use Them*. 2023. Disponível em: <https://martinfowler.com/articles/feature-toggles.html>. Cited on page 43.

GALLAND, S.; GAUD, N.; KOUKAM, A. Towards a multilevel simulation approach based on holonic multiagent systems. In: . [S.l.: s.n.], 2008. Cited on page 102.

GARCÍA, F. et al. Towards a consistent terminology for software measurement. *Information and Software Technology*, Elsevier, v. 48, n. 8, p. 631–644, 2006. Cited 2 times on the pages 103 and 104.

GitHub. *Git Guide*. 2023. Disponível em: <https://github.com/git-guides>. Cited 2 times on the pages 38 and 39.

GOETHERT, W.; FISHER, M. *Deriving Enterprise-Based Measures Using the Balanced Scorecard and Goal-Driven Measurement Techniques*. Pittsburgh, PA, 2003. Disponível em: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6507>. Cited on page 151.

GRUBER, T. R. A translation approach to portable ontology specifications. *Knowledge Acquisition*, v. 5, n. 2, p. 199–220, 1993. ISSN 1042-8143. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1042814383710083>. Cited 2 times on the pages 21 and 43.

GUERRERO, J.; CALVACHE, C.; OROZCO, C. Devops ontology - an ontology to support the understanding of devops in the academy and the software industry. *Periodicals of Engineering and Natural Sciences (PEN)*, v. 11, p. 207–220, 04 2023. Cited 2 times on the pages 103 and 104.

GUIDONI, G. L.; ALMEIDA, J. P. A.; GUIZZARDI, G. Transformation of ontology-based conceptual models into relational schemas. In: SPRINGER. *Conceptual Modeling: 39th International Conference, ER 2020, Vienna, Austria, November 3–6, 2020, Proceedings 39*. [S.l.], 2020. p. 315–330. Cited on page 136.

GUIZZARDI, G. *Ontological foundations for structural conceptual models*. Tese (Doutorado) — University of Twente, out. 2005. Cited 7 times on the pages 31, 44, 45, 46, 48, 62, and 104.

GUIZZARDI, G. Conceptualizations, modeling languages, and (meta) models. In: IOS PRESS. *Databases and Information Systems IV: Selected Papers from the Seventh International Baltic Conference, DB&IS'2006*. [S.l.], 2007. v. 155, p. 18. Cited 3 times on the pages 26, 44, and 115.

GUIZZARDI, G. et al. Ufo: Unified foundational ontology. *Applied Ontology*, IOS Press, v. 17, n. 1, p. 167–210, 2022. Cited 4 times on the pages 44, 45, 46, and 62.

GUIZZARDI, G.; FALBO, R. D. A.; GUIZZARDI, R. S. S. Grounding Software Domain Ontologies in the Unified Foundational Ontology (UFO): The case of the ODE Software Process Ontology. In: *Proceedings of the XI Iberoamerican Workshop on Requirements Engineering and Software Environments (IDEAS?2008)*. [S.l.: s.n.], 2008. Cited 3 times on the pages 44, 45, and 47.

GUIZZARDI, G. et al. Towards ontological foundations for conceptual modeling: The unified foundational ontology (ufo) story. *Applied ontology*, IOS Press, v. 10, n. 3-4, p. 259–271, 2015. Cited on page 44.

GUIZZARDI, G. et al. Towards ontological foundations for the conceptual modeling of events. In: . [S.l.: s.n.], 2013. v. 8217, p. 327–341. ISBN 978-3-642-41923-2. Cited 2 times on the pages 44 and 45.

GUIZZARDI, R. S.; GUIZZARDI, G. *Ontology-Based Transformation Framework from Tropos to AORML*. 2011. Cited on page 45.

GUSTAFSSON, J. *Single case studies vs. multiple case studies: A comparative study*. 2017. Cited on page 125.

HARMSEN, F.; BRINKKEMPER, S.; OEI, H. Situational method engineering for information systems project approaches. In: Verrijn Stuart, A.; OLLE, T. (Ed.). *Methods and Associated Tools for the Information Systems Life Cycle*. Netherlands: North Holland, 1994. (IFIP Transactions A), p. 169–194. ISBN 0-444-82074-4. IFIP WG 8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle 1994 ; Conference date: 26-09-1994 Through 28-09-1994. Cited on page 102.

HARTIG, O.; PÉREZ, J. Semantics and complexity of graphql. In: *Proceedings of the 2018 World Wide Web Conference*. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018. (WWW '18), p. 1155–1164. ISBN 9781450356398. Disponível em: <https://doi.org/10.1145/3178876.3186014>. Cited 2 times on the pages 138 and 145.

HASSAN, A. E. The road ahead for mining software repositories. In: *2008 Frontiers of Software Maintenance*. [S.l.: s.n.], 2008. p. 48–57. Cited on page 154.

HEVNER, A. R. The three cycle view of design science research. *Scandinavian Journal of Information Systems*, v. 19, n. 2, p. 87–92, 2007. Cited on page 24.

HUMBLE, J. *Continuous Delivery vs Continuous Deployment, Available*. 2010. Disponível em: <https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/>. Cited 2 times on the pages 41 and 42.

HUMBLE, J. *What is Continuous Delivery?* 2022. Disponível em: <https://continuousdelivery.com/>. Cited on page 41.

HUMBLE, J.; FARLEY, D. *Continuous delivery: reliable software releases through build, test, and deployment automation*. [S.l.]: Pearson Education, 2010. Cited 7 times on the pages 36, 37, 42, 43, 79, 95, and 98.

HUMBLE, J.; KIM, G. *Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations*. [S.l.]: IT Revolution, 2018. Cited 3 times on the pages 95, 98, and 103.

IZZA, S. Integration of industrial information systems: from syntactic to semantic integration approaches. *Enterprise Information Systems*, Taylor & Francis, v. 3, n. 1, p. 1–57, 2009. Disponível em: <https://doi.org/10.1080/17517570802521163>. Cited 4 times on the pages 57, 58, 112, and 153.

JOHANSSEN, J. O. et al. Continuous software engineering and its support by usage and decision knowledge: An interview study with practitioners. *Journal of Software: Evolution and Process*, v. 31, 05 2019. Cited 6 times on the pages 10, 31, 32, 33, 126, and 129.

JONES, C. L. et al. Practical software and systems measurement continuous iterative development measurement framework. *Version*, v. 1, p. 15, 2020. Cited on page 113.

JULIAN, B.; NOBLE, J.; ANSLOW, C. Agile practices in practice: Towards a theory of agile adoption and process evolution. In: *XP*. New York, NY, USA: Agil. Process. Softw. Eng. Extrem. Program, 2019. p. 3–18. Cited on page 19.

KARVONEN, T. et al. Hitting the target: Practices for moving toward innovation experiment systems. In: FERNANDES, J. M.; MACHADO, R. J.; WNUK, K. (Ed.). *Software Business - 6th International Conference, ICSOB 2015, Braga, Portugal, June 10-12, 2015, Proceedings*. Springer, 2015. (Lecture Notes in Business Information Processing, v. 210), p. 117–131. Disponível em: <https://doi.org/10.1007/978-3-319-19593-3_10>. Cited 5 times on the pages 24, 26, 31, 33, and 115.

KASAULI, R. et al. Agile islands in a waterfall environment: Challenges and strategies in automotive. In: *Proceedings of the Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (EASE '20), p. 31–40. ISBN 9781450377317. Disponível em: <https://doi.org/10.1145/3383219.3383223>. Cited on page 24.

KEPNER, C. H.; TREGOE, B. B. The new rational manager. Princeton research press Princeton, NJ, 1981. Cited on page 26.

KIM, D. *System Archetypes I: Diagnosing Systemic Issues and Designing High-Leverage Interventions. Toolbox Reprint Series. Cambridge, MA: Pegasus Communications*. [S.l.]: Inc, 1993. Cited on page 115.

KIM, S. et al. Automatic identification of bug-introducing changes. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. [S.l.: s.n.], 2006. p. 81–90. Cited on page 154.

KIV, S. et al. Agile methods knowledge representation for systematic practices adoption. In: KRUCHTEN, P.; FRASER, S.; COALLIER, F. (Ed.). *Agile Processes in Software Engineering and Extreme Programming*. Cham: Springer International Publishing, 2019. p. 19–34. ISBN 978-3-030-19034-7. Cited on page 102.

KNIBERG, H. *Scrum and XP from the Trenches*. [S.l.]: Lulu. com, 2015. Cited 2 times on the pages 19 and 34.

KNIBERG, H.; SKARIN, M. *Kanban and Scrum-making the most of both*. [S.l.]: Lulu. com, 2010. Cited 3 times on the pages 19, 34, and 36.

LARMAN, C.; VODDE, B. *Large-Scale Scrum: More with LeSS*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2016. ISBN 0321985710. Cited on page 34.

LEITE, L. et al. A survey of devops concepts and challenges. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 52, n. 6, nov 2019. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3359981>. Cited on page 104.

LIN, Y. et al. Scrum conceptualization using k-crio ontology. In: ABERER, K.; DAMIANI, E.; DILLON, T. (Ed.). *Data-Driven Process Discovery and Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 189–211. ISBN 978-3-642-34044-4. Cited on page 102.

MACARTHY, R. W.; BASS, J. M. An empirical taxonomy of devops in practice. In: IEEE. *2020 46th euromicro conference on software engineering and advanced applications (seaa)*. [S.l.], 2020. p. 221–228. Cited on page 103.

MARTIN, R. C. *Clean code: a handbook of agile software craftsmanship*. [S.l.]: Pearson Education, 2009. Cited 2 times on the pages 37 and 83.

MATTILA, A.-L. et al. Discovering software process deviations using visualizations. In: BAUMEISTER, H.; LICHTER, H.; RIEBISCH, M. (Ed.). *Agile Processes in Software Engineering and Extreme Programming*. Cham: Springer International Publishing, 2017. p. 259–266. ISBN 978-3-319-57633-6. Cited 2 times on the pages 22 and 154.

MEADOWS, D. H. *Thinking in systems: A primer*. [S.l.]: chelsea green publishing, 2008. Cited 2 times on the pages 26 and 115.

MORICONI, F. et al. Automated identification of flaky builds using knowledge graphs. 2022. Cited on page 102.

MUKHERJEE, J. *Continuous Delivery: Business Value, Benefits, Challenges & Metrics*. 2016. Disponível em: <https://www.atlassian.com/continuous-delivery/principles/business-value>. Cited on page 42.

NARDI, J. C.; FALBO, R. D. A.; ALMEIDA, J. P. A. Foundational Ontologies for Semantic Integration in EAI: A Systematic Literature Review. In: *Proceedings 12th IFIP WG 6.11 Conference on e-Business, e-Services, and e-Society, I3E 2013*. [S.l.]: Springer, 2013. p. 238–249. Cited 2 times on the pages 21 and 181.

OLSSON, H. H.; ALAHYARI, H.; BOSCH, J. Climbing the "stairway to heaven" – a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. [S.l.: s.n.], 2012. p. 392–399. Cited 10 times on the pages 10, 20, 26, 31, 32, 33, 114, 115, 126, and 129.

PAN, J. Z. et al. *Ontology-driven software development*. [S.l.]: Springer Science & Business Media, 2012. Cited on page 188.

PAPATHEOCHAROUS, E.; ANDREOU, A. S. Empirical evidence and state of practice of software agile teams. *J. Softw. Evol. Process*, John Wiley & Sons, Inc., USA, v. 26, n. 9, p. 855–866, sep 2014. ISSN 2047-7473. Disponível em: <https://doi.org/10.1002/smr.1664>. Cited on page 34.

PARDO, C.; OROZCO, C.; GUERRERO, J. Devops ontology-an ontology to support the understanding of devops in the academy and the software industry. *Periodicals of Engineering and Natural Sciences*, v. 11, n. 2, p. 207–220, 2023. Cited on page 103.

PARSONS, D. Agile software development methodology, an ontological analysis. In: *Proceedings of 9th International Conference on Applications and Principles of Information Science*. [S.l.: s.n.], 2010. p. 5–9. Cited on page 102.

PEFFERS, K. et al. A design science research methodology for information systems research. *J. Manage. Inf. Syst.*, M. E. Sharpe, Inc., USA, v. 24, n. 3, p. 45–77, dec 2007. ISSN 0742-1222. Disponível em: <https://doi.org/10.2753/MIS0742-1222240302>. Cited on page 24.

POKRAEV, S. Model-driven semantic integration of service-oriented applications. In: . [S.l.: s.n.], 2009. Cited on page 21.

PUPPET. *State of DevOps Report*. 2015. Disponível em: <https://puppetlabs.com/2015-devops-repor/>. Cited on page 41.

PUTTA, A.; PAASIVAARA, M.; LASSENIUS, C. Adopting scaled agile framework (safe): A multivocal literature review. In: *Proceedings of the 19th International Conference on Agile Software Development: Companion*. New York, NY, USA: Association for Computing Machinery, 2018. (XP '18). ISBN 9781450364225. Disponível em: <https://doi.org/10.1145/3234152.3234164>. Cited on page 34.

RAHMAN, M. T. et al. Feature toggles: Practitioner practices and a case study. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2016. (MSR '16), p. 201–211. ISBN 9781450341868. Disponível em: <https://doi.org/10.1145/2901739.2901745>. Cited on page 43.

RENAULT, L. D.; BARCELLOS, M. P.; FALBO, R. D. A. Using an Ontology-based Approach for Integrating Applications to Support Software Processes. In: *Anais do XVII Simpósio Brasileiro de Qualidade de Software*. [S.l.: s.n.], 2018. Cited on page 154.

RISING, L.; JANOFF, N. S. The scrum software development process for small teams. *IEEE software*, IEEE, v. 17, n. 4, p. 26–32, 2000. Cited on page 35.

RODRÍGUEZ, P. et al. Survey on agile and lean usage in finnish software industry. In: *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: Association for Computing Machinery, 2012. (ESEM '12), p.

139–148. ISBN 9781450310567. Disponível em: <https://doi.org/10.1145/2372251.2372275>. Cited on page 34.

RUBIN, K. S. *Essential Scrum: A practical guide to the most popular Agile process*. [S.l.]: Addison-Wesley, 2012. Cited 3 times on the pages 63, 64, and 67.

RUNESON, P. et al. *Case study research in software engineering: Guidelines and examples*. [S.l.]: John Wiley & Sons, 2012. Cited 3 times on the pages 123, 129, and 176.

RUY, F. et al. SEON: A Software Engineering Ontology Network. In: *Proceedings of the 20th International Conference on Knowledge Engineering and Knowledge Management*. [S.l.: s.n.], 2016. Cited 20 times on the pages 10, 21, 22, 25, 28, 31, 44, 47, 48, 61, 103, 104, 116, 120, 132, 134, 155, 180, 181, and 183.

RUY, F. B. *Software Engineering Standards Harmonization: An Ontology-based Approach*. Tese (Doutorado) — Universidade Federal do Espírito Santo, 2017. Cited 3 times on the pages 48, 104, and 137.

SALAMON, J. S. *Uma Abordagem Orientada a Objetivos para Desenvolvimento de Ontologias baseado em Integração*. Dissertação (Mestrado) — Universidade Federal do Espírito Santo, 2018. Cited 2 times on the pages 137 and 156.

SANTOS, P. S.; ALMEIDA, J. a. P. A.; BARCELLOS, M. Towards federated ontology-driven data integration in continuous software engineering. In: *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2023. (SBES ’23), p. 31–36. ISBN 9798400707872. Disponível em: <https://doi.org/10.1145/3613372.3613380>. Cited 3 times on the pages 158, 159, and 182.

SANTOS, P. S.; BARCELLOS, M. P.; ALMEIDA, J. P. A. An ontology-based approach to enable data-driven decision-making in agile software organizations. In: *V Doctoral and Masters Consortium on Ontologies (WTDO 2021)*. [S.l.: s.n.], 2021. (Ontobras 2021), p. 279–284. Cited on page 28.

SANTOS, P. S.; BARCELLOS, M. P.; CALHAU, R. F. Am i going to heaven? first step climbing the stairway to heaven model results from a case study in industry. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (SBES ’20), p. 309–318. ISBN 9781450387538. Disponível em: <https://doi.org/10.1145/3422392.3422406>. Cited 15 times on the pages 11, 26, 27, 28, 116, 120, 121, 122, 131, 132, 159, 160, 177, 181, and 182.

SANTOS, P. S. et al. From a scrum reference ontology to the integration of applications for data-driven software development. *Information and Software Technology*, Elsevier BV, v. 136, p. 106570, aug 2021. Disponível em: <https://doi.org/10.1016%2Fj.infsof.2021.106570>. Cited 15 times on the pages 11, 13, 25, 28, 109, 110, 111, 131, 132, 136, 137, 158, 181, 182, and 183.

SANTOS, P. S. et al. Flying over brazilian organizations with zeppelin: A preliminary panoramic picture of continuous software engineering. In: *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022. (SBES ’22), p. 279–288. ISBN 9781450397353. Disponível em: <https://doi.org/10.1145/3555228.3555234>. Cited 6 times on the pages 20, 28, 131, 158, 160, and 180.

SANTOS, P. S. d.; BARCELLOS, M. P.; CALHAU, R. F. First step climbing the stairway to heaven model - results from a case study in industry. *Journal of Software Engineering Research and Development*, v. 10, p. 5:1 – 5:18, Mar. 2022. Disponível em: <https://sol.sbc.org.br/journals/index.php/jserd/article/view/1992>. Cited 11 times on the pages 26, 28, 116, 120, 121, 131, 158, 166, 181, 182, and 188.

SANTOS, P. S. d.; BARCELLOS, M. P.; RUY, F. B. Tell me: Am i going to heaven? a diagnosis instrument of continuous software engineering practices adoption. In: *Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. (EASE 2021), p. 30–39. ISBN 9781450390538. Disponível em: <https://doi.org/10.1145/3463274.3463324>. Cited 11 times on the pages 11, 27, 28, 125, 126, 127, 129, 130, 131, 181, and 183.

SATPATHY, T. et al. A guide to the scrum body of knowledge (sbok™ guide). *Scrumstudy™, a brand of VMEdu, Inc*, 2016. Cited 3 times on the pages 63, 64, and 66.

SCHERP, A. et al. Designing core ontologies. *Applied Ontology*, v. 6, 08 2011. Cited 2 times on the pages 43 and 48.

SCHWABER, J.; KEN, S. The scrum guide-the definitive guide to scrum: The rules of the game. In: . [s.n.], 2013. Disponível em: <http://scrum.org>. Cited 3 times on the pages 19, 34, and 71.

SCHWABER, K.; BEEDLE, M. *Agile software development with Scrum*. [S.l.]: Prentice Hall Upper Saddle River, 2002. v. 1. Cited 4 times on the pages 63, 67, 69, and 71.

SCHWABER, K.; SUTHERLAND, J. The scrum guide. *Scrum Alliance*, v. 21, n. 1, 2011. Cited 3 times on the pages 35, 63, and 67.

SHAHIN, M.; BABAR, M. A.; ZHU, L. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, v. 5, p. 3909–3943, 2017. ISSN 2169-3536. Cited 12 times on the pages 10, 36, 41, 42, 43, 79, 82, 83, 87, 95, 97, and 98.

SHI, J. et al. Study on log-based change data capture and handling mechanism in real-time data warehouse. In: *2008 International Conference on Computer Science and Software Engineering*. [S.l.: s.n.], 2008. v. 4, p. 478–481. Cited on page 146.

SKELTON, M.; O'DELL, C. *Continuous delivery with windows and. NET*. [S.l.]: O'Reilly Media, 2016. Cited on page 42.

SOUZA, E. F. D.; FALBO, R. D. A.; VIJAYKUMAR, N. L. ROoST: Reference Ontology on Software Testing. *APPLIED ONTOLOGY (ONLINE)*, v. 12, p. 59–90, 2017. ISSN 18758533. Cited 2 times on the pages 48 and 104.

SOUZA, L. M. S. d. *Ramble ON: Utilizando Medição de software e Ontologias em Rede para prover Dados de apoio à Tomada de Decisão*. [S.l.], 2023. Cited on page 151.

STÅHL, D.; BOSCH, J. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, Elsevier, v. 87, p. 48–59, 2014. Cited on page 79.

STARON, M. et al. Measurement and impact factors of speed of reviews and integration in continuous software engineering. *Foundations of Computing and Decision Sciences*, v. 43, n. 4, p. 281–303, 2018. Cited on page 79.

STERMAN, J. *Business dynamics.* [S.l.]: Irwin/McGraw-Hill c2000.., 2010. Cited on page 115.

STUDER, R.; BENJAMINS, V. R.; FENSEL, D. Knowledge engineering: principles and methods. *Data & knowledge engineering*, Elsevier, v. 25, n. 1-2, p. 161–197, 1998. Cited on page 43.

SUÁREZ-FIGUEROA, M. C.; GÓMEZ-PÉREZ, A.; FERNÁNDEZ-LÓPEZ, M. The neon methodology for ontology engineering. In: *Ontology engineering in a networked world.* [S.l.]: Springer, 2012. p. 9–34. Cited on page 21.

SUÁREZ-FIGUEROA, M. C. et al. *Ontology Engineering in a Networked World.* [S.l.: s.n.], 2012. ISBN 978-3-642-24793-4. Cited on page 44.

SVENSSON, R. B.; FELDT, R.; TORKAR, R. The unfulfilled potential of data-driven decision making in agile software development. In: KRUCHTEN, P.; FRASER, S.; COALLIER, F. (Ed.). *Agile Processes in Software Engineering and Extreme Programming.* Cham: Springer International Publishing, 2019. p. 69–85. ISBN 978-3-030-19034-7. Cited 2 times on the pages 20 and 24.

TAUTZ, C.; WANGENHEIM, C. Gresse von. *REFSENO: A representation formalism for software engineering ontologies.* 1998. Cited on page 103.

THEMISTOCLEOUS, M.; IRANI, Z. Evaluating the integration of supply chain information systems: A case study. *European Journal of Operational Research*, v. 159, p. 393–405, 12 2004. Cited on page 58.

TRINKENREICH, B. et al. Combining GQM+Strategies and OKR - Preliminary Results from a Participative Case Study in Industry. In: *Proceedings of the 20th International Conference on Product-Focused Software Process Improvement.* [S.l.: s.n.], 2019. Cited on page 151.

VERNADAT, F. Interoperable enterprise systems: Principles, concepts, and methods. *Annual Reviews in Control*, v. 31, p. 137–145, 12 2007. Cited 2 times on the pages 57 and 58.

WACHE, H. et al. Ontology-based information integration: A survey. *Bremen, The BUSTER Project, Intelligent Systems Group*, 2001. Cited 3 times on the pages 21, 154, and 181.

WEBER, I.; NEPAL, S.; ZHU, L. Developing dependable and secure cloud applications. *IEEE Internet Computing*, v. 20, n. 3, p. 74–79, 2016. Cited on page 41.

WEGNER, P. Interoperability. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 28, n. 1, p. 285–287, mar 1996. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/234313.234424>. Cited on page 57.

WIERINGA, R. J. *Design science methodology for information systems and software engineering.* [S.l.]: Springer, 2014. Cited on page 106.

WIKIPEDIA. *Compiler — Wikipedia, The Free Encyclopedia.* 2023. <http://en.wikipedia.org/w/index.php?title=Compiler&oldid=1136440759>. [Online; accessed 07-February-2023]. Cited on page 85.

WIKIPEDIA. *Data Lake — Wikipedia, The Free Encyclopedia.* 2023. <https://en.wikipedia.org/wiki/Datalake>. [Online; accessed 07-February-2023]. Cited on page 148.

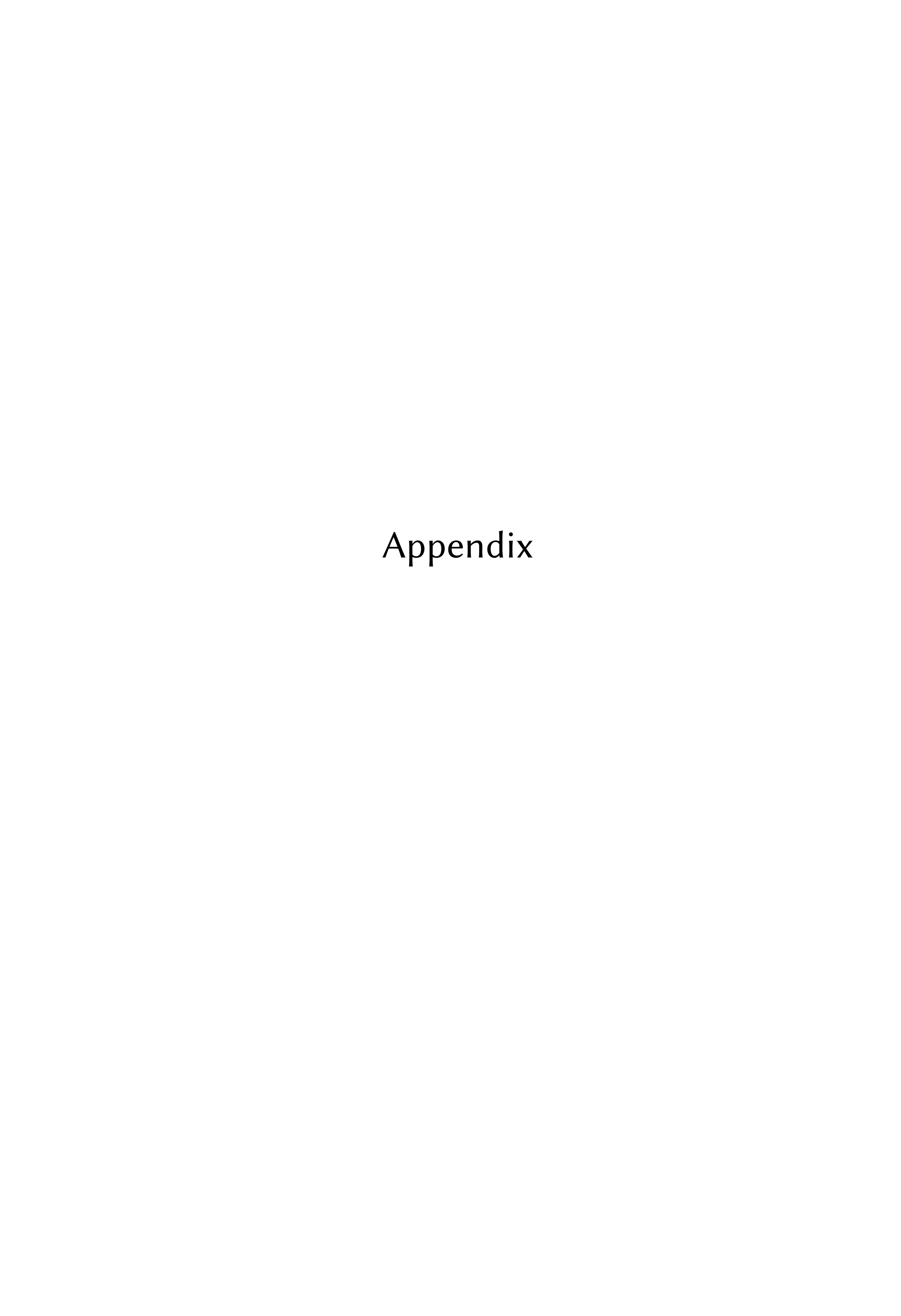
WIKIPEDIA. *Interpreter (computing) — Wikipedia, The Free Encyclopedia.* 2023. <http://en.wikipedia.org/w/index.php?title=Interpreter%20(computing)&oldid=1134005862>. [Online; accessed 07-February-2023]. Cited on page 85.

WIKIPEDIA. *Lint (software) — Wikipedia, The Free Encyclopedia*. 2023. <http://en.wikipedia.org/w/index.php?title=Lint%20(software)&oldid=1137767297>. [Online; accessed 07-February-2023]. Cited on page 89.

WIKIPEDIA. *No-Code Development Platform — Wikipedia, The Free Encyclopedia*. 2023. <https://en.wikipedia.org/wiki/No-codedevelopmentplatform>. [Online; accessed 07-February-2023]. Cited on page 148.

WIKIPEDIA. *Source-to-source compiler — Wikipedia, The Free Encyclopedia*. 2023. <http://en.wikipedia.org/w/index.php?title=Source-to-source%20compiler&oldid=1126015947>. [Online; accessed 07-February-2023]. Cited on page 85.

WILLIAMS, L.; COCKBURN, A. Agile software development: it's about feedback and change. *Computer*, v. 36, n. 6, p. 39–43, 2003. Cited on page 115.



Appendix

APPENDIX A – Zeppelin

Yes, there are two paths you can go by, but in the long run, there's still time to change the road you're on

Led Zeppelin, Stairway to Heaven

This Appendix introduces details about *Zeppelin*. Section A.1 presents the statements of the *Zeppelin*'s Diagnostic Questionnaire and their related stage.

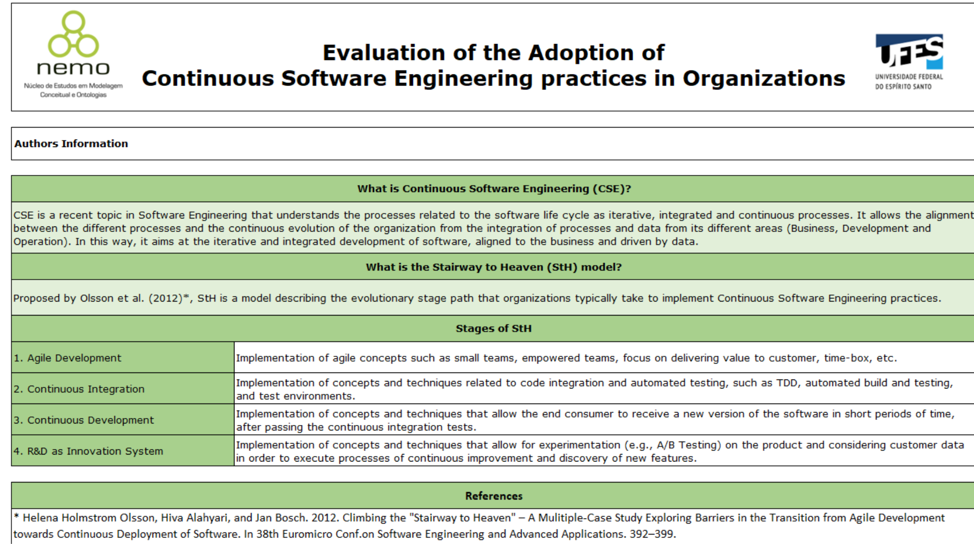
A.1 Diagnostic Questionnaire

The *Zeppelin's Diagnostic Questionnaire* contains the following components: (i) Context, (ii) Instructions, (iii) Organization Profile, (iv) Participant Profile, and (v) 76 statements expressing CSE practices organized in four stages of the StH model: Agile Organization (AO) (26 practices), Continuous Integration (CI) (15 practices), Continuous Deployment (CD) (17 practices) and R&D as Innovation System (RD) (13 practices). Each StH stage is a form in *Zeppelin's Diagnostic Questionnaire*. Therefore, *Zeppelin's Diagnostic Questionnaire* comprises 8 forms.

Context presents content about Continuous Software Engineering (CSE), the Stairway to Heaven model (StH), and its stages (*Agile Organization*, *Continuous Integration*, *Continuous Deployment*, and *R&D as Innovation System*) to provide basic infomation for answering the questionnaire. Instruction explains t how to answer the statements in each table using the Adoption levels (Not Adopted, Abandoned, Project/Product, Process, and Institutionalized). Figures 78, and 79 present Context, and *Instruction* forms.

Organization Profile aims to characterize the organization by capturing the following data: (i) Organization Type (Startup, Software House, and Organization with It Department), (ii) Organization Size (Micro, Small, Medium-sized, and Large Organization), and (iii) Organization Time of Existence. In addition, it is possible to provide data about the development teams: (i) Size of the Development team and (ii) Quantity of members of the development team work with agile or other practices present in StH, as can be seen in Figure 80.

Participant Profile is used to understand general data about the person answering the questionnaire (name, e-mail, Academic Background, Academic Degree, and Conclusion of the Academic Degree) and its *level of knowledge* and *practice* in CSE practices in each StH' stage. For each stage of the StH, the respondent can declare the following levels of knowledge: (i) None (You have never heard about the topic, or have no knowledge about it.), (ii) Low (You have not taken any courses, but obtained some knowledge from books or other materials.),



nemo Núcleo de Estudos em Modelagem Conceitual e Ontologias

Evaluation of the Adoption of Continuous Software Engineering practices in Organizations

UFES UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Authors Information

What is Continuous Software Engineering (CSE)?

CSE is a recent topic in Software Engineering that understands the processes related to the software life cycle as iterative, integrated and continuous processes. It allows the alignment between the different processes and the continuous evolution of the organization from the integration of processes and data from its different areas (Business, Development and Operation). In this way, it aims at the iterative and integrated development of software, aligned to the business and driven by data.

What is the Stairway to Heaven (StH) model?

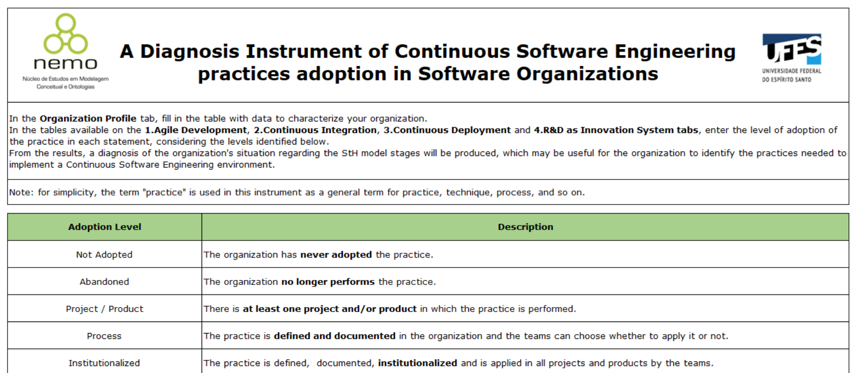
Proposed by Olsson et al. (2012)*, StH is a model describing the evolutionary stage path that organizations typically take to implement Continuous Software Engineering practices.

Stages of StH	
1. Agile Development	Implementation of agile concepts such as small teams, empowered teams, focus on delivering value to customer, time-box, etc.
2. Continuous Integration	Implementation of concepts and techniques related to code integration and automated testing, such as TDD, automated build and testing, and test environments.
3. Continuous Development	Implementation of concepts and techniques that allow the end consumer to receive a new version of the software in short periods of time, after passing the continuous integration tests.
4. R&D as Innovation System	Implementation of concepts and techniques that allow for experimentation (e.g., A/B Testing) on the product and considering customer data in order to execute processes of continuous improvement and discovery of new features.

References

* Helena Holmstrom Olsson, Hiva Alahyari, and Jan Bosch. 2012. Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In 38th Euromicro Conf.on Software Engineering and Advanced Applications. 392–399.

Figure 78 – Diagnostic Questionnaire - Context Form.



nemo Núcleo de Estudos em Modelagem Conceitual e Ontologias

A Diagnosis Instrument of Continuous Software Engineering practices adoption in Software Organizations

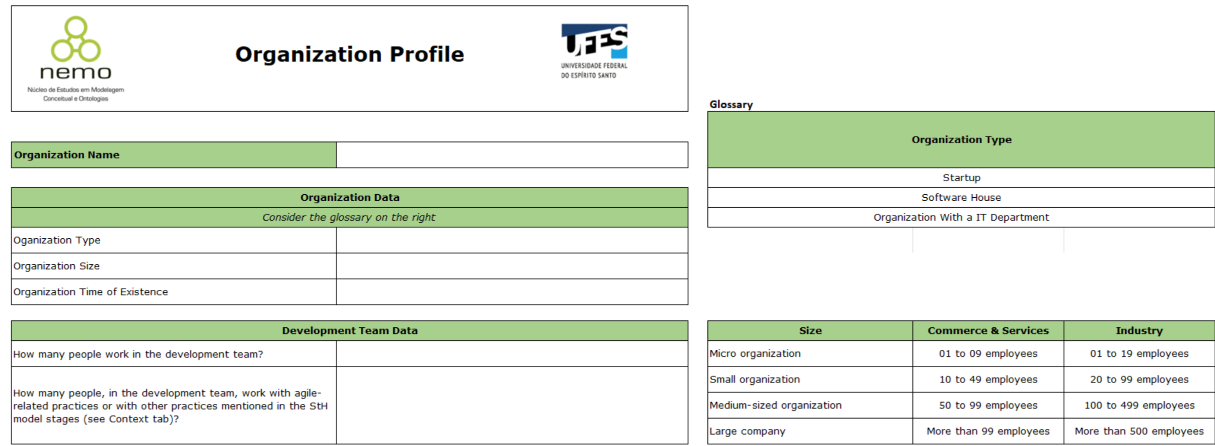
UFES UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

In the **Organization Profile** tab, fill in the table with data to characterize your organization.
In the tables available on the **1.Agile Development**, **2.Continuous Integration**, **3.Continuous Deployment** and **4.R&D as Innovation System tabs**, enter the level of adoption of the practice in each statement, considering the levels identified below.
From the results, a diagnosis of the organization's situation regarding the StH model stages will be produced, which may be useful for the organization to identify the practices needed to implement a Continuous Software Engineering environment.

Note: for simplicity, the term "practice" is used in this instrument as a general term for practice, technique, process, and so on.

Adoption Level	Description
Not Adopted	The organization has **never adopted** the practice.
Abandoned	The organization **no longer performs** the practice.
Project / Product	There is **at least one project and/or product** in which the practice is performed.
Process	The practice is **defined and documented** in the organization and the teams can choose whether to apply it or not.
Institutionalized	The practice is defined, documented, **institutionalized** and is applied in all projects and products by the teams.

Figure 79 – Diagnostic Questionnaire - Instruction Form.



nemo Núcleo de Estudos em Modelagem Conceitual e Ontologias

Organization Profile

UFES UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Organization Name	

Organization Data	
Consider the glossary on the right	
Oganization Type	
Organization Size	
Organization Time of Existence	

Development Team Data	
How many people work in the development team?	
How many people, in the development team, work with agile-related practices or with other practices mentioned in the StH model stages (see Context tab)?	

Glossary

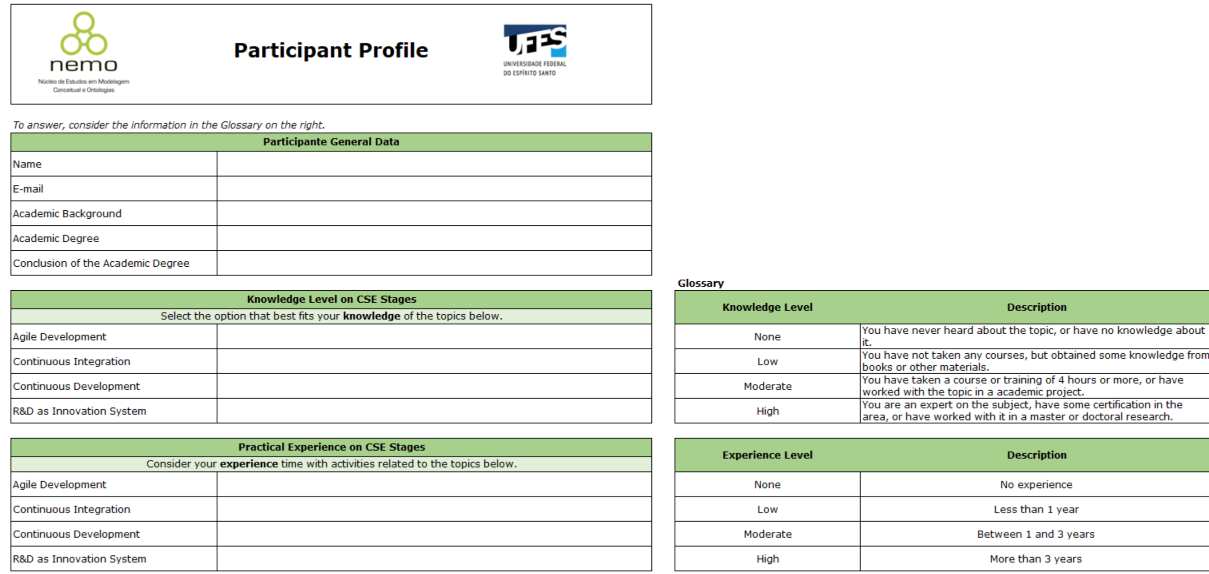
Organization Type
Startup
Software House
Organization With a IT Department

Size	Commerce & Services	Industry
Micro organization	01 to 09 employees	01 to 19 employees
Small organization	10 to 49 employees	20 to 99 employees
Medium-sized organization	50 to 99 employees	100 to 499 employees
Large company	More than 99 employees	More than 500 employees

Figure 80 – Diagnostic Questionnaire - Organization Profile Form.

(ii) Moderate (You have taken a course or training of 4 hours or more, or have worked with the topic in an academic project.), and (iv) High (You are an expert on the subject, have some certification in the area, or have worked with it in a master or doctoral research.). In the same way as the level of knowledge, the respondent can also inform their level of practice on CSE

practice in each StH‘ stage: None (No experience), Low (Less than 1 year), Moderate (Between 1 and 3 years), and High (More than 3 years). The *Participant Profile* can be seen in Figure 81.



Participant Profile

nemo – Núcleo de Estudos em Modelagem Conceitual e Ontologias

UFES – Universidade Federal do Espírito Santo

To answer, consider the information in the Glossary on the right.

Participante General Data	
Name	
E-mail	
Academic Background	
Academic Degree	
Conclusion of the Academic Degree	

Knowledge Level on CSE Stages Select the option that best fits your **knowledge** of the topics below.	
Agile Development	
Continuous Integration	
Continuous Development	
R&D as Innovation System	

Practical Experience on CSE Stages Consider your **experience** time with activities related to the topics below.	
Agile Development	
Continuous Integration	
Continuous Development	
R&D as Innovation System	

Glossary

Knowledge Level	Description
None	You have never heard about the topic, or have no knowledge about it.
Low	You have not taken any courses, but obtained some knowledge from books or other materials.
Moderate	You have taken a course or training of 4 hours or more, or have worked with the topic in a academic project.
High	You are an expert on the subject, have some certification in the area, or have worked with it in a master or doctoral research.

Experience Level	Description
None	No experience
Low	Less than 1 year
Moderate	Between 1 and 3 years
High	More than 3 years

Figure 81 – Diagnostic Questionnaire - Participant Profile Form.

The *Agile Organization* form presents the statements related to the implementation of the agile concepts such as small teams, empowered teams, focus on delivering value to customers, time-box, etc. Figure 82 shows this form.

Agile Organization

nemo – Núcleo de Estudos em Modelagem Conceitual e Ontologias

UFES – Universidade Federal do Espírito Santo

Implementation of agile concepts such as small teams, empowered teams, focus on delivering value to customer, time-box, etc.

#	Statement	Adoption Level	Comments *(provide details on how the practices are performed or, if don't, what the organization does)*
AO.01	Roles involved in the agile development process (e.g., Scrum Master, Product Owner, Developer, and Tester) exist in the organization.		List the roles in the organization:
AO.02	Project teams include a Product Owner, who is responsible for representing the Customer and actively participates in the projects.		
AO.03	The scope of the project is defined gradually, using the Product Backlog (or equivalent artifact).		
AO.04	Effort estimation is performed by (or together with) the development team considering short activities to implement a set of selected requirements (and not the project as a whole).		
AO.05	Cost estimation is established based on the effort estimation and considers the effort needed to implement a selected set of requirements (and not the project as a whole).		
AO.06	To deliver value to the customer, requirements are defined and prioritized according to customer needs, are periodically reviewed, and changes are absorbed into iterations of the development process.		
AO.07	The development process is performed iteratively, in short cycles (e.g., 2 weeks), in which selected project requirements recorded in a Sprint Backlog (or equivalent artifact) are developed.		Inform the average time of the time-box:
AO.08	The customer receives new versions of the product frequently (after one or more short development cycles), including new functionality defined according to customer needs.		
AO.09	The organization has clear acceptance criteria for software requirements and they are used to evaluate the artifacts (e.g., functionality) produced and define if they are concluded.		
AO.10	The organization has at least one person (e.g. Software Architect, Tech Lead, or Quality Analyst) responsible for the quality of the produced artifacts, including the final product.		
AO.11	The team frequently (e.g., daily, every two or three days) reflects on the development progress within the scope of what has been defined for the current time-box and adjusts tasks if necessary (e.g., stand-up meetings)		
AO.12	The team frequently meets to discuss improvements to the product, process, or tools during the projects (e.g., in retrospective meetings).		
AO.13	The team frequently meets to discuss improvements in team members' skills during the projects (e.g., in retrospective meetings).		
AO.14	Teams are small (usually between 4 to 6 developers), self-organized and multidisciplinary.		
AO.15	The project team has autonomy to make technical decisions on the project.		
AO.16	The project stakeholders (including the client) are encouraged to think about their role and responsibilities in the project.		
AO.17	Good programming practices are adopted (e.g., collective coding, standardized coding, pair programming, code review, etc.).		Inform the adopted practices:
AO.18	Good testing practices are adopted (automated testing, test-driven development, etc.)		Inform the adopted practices:
AO.19	Data is collected for metrics to evaluate quality aspects of the produced artifacts and the product (e.g., cyclomatic complexity, number of code smells).		Inform some used metrics:
AO.20	Data is collected for metrics to evaluate performance aspects of the agile development process (e.g., work in progress, velocity).		Inform some used metrics:
AO.21	Data produced (task performer, task completion date, story points, etc.) throughout the development of the projects is stored in one (or more) data repository.		Indicate whether there is one or more data repository and what data is stored:
AO.22	Decisions in projects are made based on data from one (or more) data repository.		
AO.23	The agile software development process is continuously evaluated and improved.		
AO.24	Data stored in the repository is used to improve the product and the agile software development process.		
AO.25	The organization has practices to share relevant knowledge to (agile) software development (e.g., internal lectures, tutorials, knowledge repositories, guild implementations).		Inform the adopted practices:
AO.26	The (agile) development process is aligned to the organization's business and this is perceived by the value delivery to the customer and her/his satisfaction with the delivered product.		Comment on how the alignment between the software development and the business areas is done:

Figure 82 – Diagnostic Questionnaire - Agile Organization Form.

The *Continuous Integration* form regards the statements related to the implementation of the concepts and techniques related to code integration and automated testing, such as TDD, automated build and testing, and test environments. Figure83 shows this form.

Continuous Deployment form presents the statements related to the implementation of the concepts and techniques that allow the end consumer to receive a new version of the

Continuous Integration

Implementation of concepts and techniques related to code integration and automated testing, such as TDD, automated build and testing, and test environments.

#	Statement	Adoption Level	Comments *(provide details on how the practices are performed or, if don't, what the organization does)*
CI.01	The software architecture is modular in order to allow automated testing.		
CI.02	The software architecture is modular in order to allow automated builds.		
CI.03	Code is constantly and automatically integrated.		
CI.04	Tests are automatically executed periodically (e.g., whenever new code is integrated), in a`testing environment, to verify code quality (e.g., coverage, correctness).		
CI.05	Automated tests are used to assess whether the implemented software meets established requirements.		
CI.06	Builds occur frequently and automatically.		
CI.07	Builds are canceled if one or more tests fail		
CI.08	There is version control of software artifacts (e.g., code, tests, scripts, etc.) in a repository.		
CI.09	Good check-in practices are applied in the development trunk (e.g., use of tools like GitFlow).		
CI.10	There are practices that allow organizations or individuals external to the project to contribute to the product implementation (i.e., produce and integrate code into the product being developed)		
CI.11	Data is collected for metrics that allow evaluation of the continuous integration process (e.g.,number of canceled builds, number of code integrations performed).		
CI.12	Data produced in continuous integration environments (e.g., build dates, number of tests executed, and percentage of coverage) are stored in one (or more) data repository.		
CI.13	The continuous integration process (including automated testing) is continuously evaluated and improved.		Inform the adopted practices:
CI.14	Data stored in the data repository(ies) are used to improve the product and the continuous integration process.		Inform some used metrics:
CI.15	The organization adopts practices for sharing knowledge related to continuous integration (e.g., internal lectures, tutorials, knowledge repositories, guild implementations).		

Figure 83 – Diagnostic Questionnaire - Continuous Integration Form.

software in short periods, after passing the continuous integration tests. Figure 84 shows this form.

Continuous Deployment

Implementation of concepts and techniques that allow the end consumer to receive a new version of the software in short periods of time, after passing the continuous integration tests.

#	Statement	Adoption Level	Comments *(provide details on how the practices are performed or, if don't, what the organization does)*
CD.01	The main customers/consumers are identified and participate in the development process, influencing the functionalities that will be produced and delivered.		
CD.02	There is a clear flow of information between Development and Operation, allowing that new functionality developed to go live automatically.		
CD.03	The delivery of new functionalities is performed automatically and through releases.		
CD.04	There is a clear flow of information between Operation and Business, allowing new customer/consumer needs and business opportunities to be identified from the delivery of new functionality.		
CD.05	The software architecture allows the delivery (deployment) of functionalities independently.		
CD.06	Consumers receive new functionalities frequently, including in shorter cycles than the time-box typically established in the development process.		
CD.07	Clients can test the product as soon as new functionalities are deployed.		
CD.08	The organization's business model is constantly evaluated and revised (when necessary) based on customer/consumer information		
CD.09	Marketing strategies are constantly evaluated and revised (when necessary) based on lead customers' information (customers/consumers most relevant to the organization).		
CD.10	Sales strategies are constantly evaluated and revised (when necessary) based on lead customers' information (customers/consumers most relevant to the organization).		
CD.11	Alignment between product development and the organization's business is maintained through continuous checks, in short cycles.		
CD.12	Alignment between product development and the organization's business is maintained through continuous checks in short cycles, based on data.		
CD.13	Data is collected for metrics that evaluate the continuous delivery process (e.g., number of releases, defect density in releases).		Inform some used metrics:
CD.14	Data produced in continuous deployment environments (e.g., release dates and software version delivered) are stored in one (or more) data repositories.		
CD.15	The continuous deployment process is continuously evaluated and improved.		
CD.16	Data stored in the data repository is used to improve the product and the continuous deployment process.		
CD.17	The organization has practices to share knowledge related to continuous deployment (e.g., internal lectures, tutorials, knowledge repositories, guild implementation).		Inform the adopted practices:

Figure 84 – Diagnostic Questionnaire - Continuous Deployment Form.

Finally, *R&D as Innovation System* form presents the statements related to the implementation of the concepts and techniques that allow for experimentation (e.g., A/B Testing) on the product and considering customer data to execute processes of continuous improvement and discovery of new features. Figure 85 shows this form.

R&D as Innovation System

Implementation of concepts and techniques that allow for experimentation (e.g., A/B Testing) on the product and considering customer data in order to execute processes of continuous improvement and discovery of new features.

#	Statement	Adoption Level	Comments *(provide details on how the practices are performed or, if don't, what the organization does)*
IS.01	Feedbacks (data and opinions) from customers/consumers are captured and stored in a customer/consumer data repository.		Inform how data is captured and stored:
IS.02	Feedbacks (data and opinions) from customers/consumers are continuously and automatically captured.		Inform some of the captured data:
IS.03	Feedbacks (data and opinions) from customers/consumers are used to improve products (enhance existing features and identify new ones).		
IS.04	The organization identifies new business opportunities based on automatically captured customer/consumer feedbacks.		
IS.05	Feedbacks (data and opinions) from customers/consumers are used for experimentation and innovation.		
IS.06	Experiments (e.g., A/B tests) are conducted with customers/consumers to improve products.		
IS.07	Technologies (e.g., cloud technologies) are adopted to enhance experimentation.		
IS.08	The organization continually experiments new technologies and methodologies.		
IS.09	The organization has a clear information flow between the strategic level and the development area, allowing customer/consumer data to be used in an aligned way in making technical and business decisions.		
IS.10	Data from the customer/consumer data repository is used in decision making by the software development area.		
IS.11	Data from the customer/consumer data repository is used in decision making by the business area.		
IS.12	Alignment between product development and the organization's business is maintained through continuous checks, in short cycles and based on customer/consumer data.		
IS.13	Knowledge management practices are adopted based on data from the customer/consumer data repository.		

Figure 85 – Diagnostic Questionnaire - R&D as Innovation System Form.

For better visualization, tables 29, 30, 31, and 32 list the statements of the Diagnostic Questionnaire and their related stage.

Table 29 – Agile Organization Stage's Statements.

#	**Statement**
AO.01	Roles involved in the agile development process (e.g., Scrum Master, Product Owner, Developer, and Tester) exist in the organization.
AO.02	Project teams include a Product Owner, who is responsible for representing the Customer and actively participates in the projects.
AO.03	The scope of the project is defined gradually, using the Product Backlog (or equivalent artifact).
AO.04	Effort estimation is performed by (or together with) the development team considering short activities to implement a set of selected requirements (and not the project as a whole).
AO.05	Cost estimation is established based on the effort estimation and considers the effort needed to implement a selected set of requirements (and not the project as a whole).
AO.06	To deliver value to the customer, requirements are defined and prioritized according to customer needs, are periodically reviewed, and changes are absorbed into iterations of the development process.
AO.07	The development process is performed iteratively, in short cycles (e.g., 2 weeks), in which selected project requirements recorded in a Sprint Backlog (or equivalent) are developed.
AO.08	The customer receives new versions of the product frequently (after one or more short development cycles), including new functionality defined according to customer needs.
AO.09	The organization has clear acceptance criteria for software requirements and they are used to evaluate the artifacts (e.g., functionality) produced and define if they are concluded.
AO.10	The organization has at least one person (e.g. Software Architect, Tech Lead, or Quality Analyst) responsible for the quality of the produced artifacts, including the final product.
AO.11	The team frequently (e.g., daily, every two, or three days) reflects on the development progress within the scope of what has been defined for the current time-box and adjusts tasks if necessary (e.g., stand-up meetings).
AO.12	The team frequently meets to discuss improvements to the product, process, or tools during the projects (e.g., retrospective meetings).
AO.13	The team frequently meets to discuss improvements in team members' skills during the projects (e.g., in retrospective meetings).
AO.14	Teams are small (usually between 4 to 6 developers), self-organized and multidisciplinary.
AO.15	The project team has autonomy to make technical decisions on the project.
AO.16	The project stakeholders (including the client) are encouraged to think about their role and responsibilities in the project.
AO.17	Good programming practices are adopted (e.g., collective coding, standardized coding, pair programming, code review, etc.).
AO.18	Good testing practices are adopted (automated testing, test-driven development, etc.).
AO.19	Data is collected for metrics to evaluate quality aspects of the produced artifacts and the product (e.g., cyclomatic complexity, number of code smells).
AO.20	Data is collected for metrics to evaluate performance aspects of the agile development process (e.g., work in progress, velocity).
AO.21	Data produced (task owner, task completion date, story points, etc) throughout the development of the projects is stored in one (or more) data repository.
AO.22	Decisions in projects are made based on data from one (or more) data repository.

Table 29 – *Continued from previous page*

#	Statement
AO.23	The agile software development process is continuously evaluated and improved.
AO.24	Data stored in the repository is used to improve the product and the agile software development process.
AO.25	The organization has practices to share relevant knowledge to (agile) software development (e.g., internal lectures, tutorials, knowledge repositories, guild implementations).
AO.26	The (agile) development process is aligned to the organization's business and this is perceived by the value delivery to the customer and her/his satisfaction with the delivered product.

Table 30 – Continuous Integration Stage's Statements.

#	Statement
CI.01	The software architecture is modular in order to allow automated testing.
CI.02	The software architecture is modular in order to allow automated builds.
CI.03	Code is constantly and automatically integrated.
CI.04	Tests are automatically executed periodically (e.g., whenever new code is integrated), in a testing environment, to verify code quality (e.g., coverage, correctness).
CI.05	Automated tests are used to assess whether the implemented software meets established requirements.
CI.06	Builds occur frequently and automatically.
CI.07	Builds are canceled if one or more tests fail.
CI.08	There is version control of software artifacts (e.g., code, tests, scripts, etc.) in a repository.
CI.09	Good check-in practices are applied in the development trunk (e.g., use of tools like GitFlow).
CI.10	There are practices that allow organizations or individuals external to the project to contribute to the product implementation (i.e., produce and integrate code into the product being developed).
CI.11	Data is collected for metrics that allow evaluation of the continuous integration process (e.g., number of canceled builds, number of code integrations performed).
CI.12	Data produced in continuous integration environments (e.g., build dates, number of tests executed, and percentage of coverage) are stored in one (or more) data repository.
CI.13	The continuous integration process (including automated testing) is continuously evaluated and improved.
CI.14	Data stored in the data repository(ies) are used to improve the product and the continuous integration process.
CI.15	The organization adopts practices for sharing knowledge related to continuous integration (e.g., internal lectures, tutorials, knowledge repositories, guild implementations).

Table 31 – Continuous Deployment Stage's Statements.

#	Statement
CD.01	The main customers/consumers are identified and participate in the development process, influencing the functionalities that will be produced and delivered.
CD.02	There is a clear flow of information between Development and Operation, allowing that new functionality developed to go live automatically.
CD.03	The delivery of new functionalities is performed automatically and through releases.

Table 31 – *Continued from previous page*

#	Statement
CD.04	There is a clear flow of information between Operation and Business, allowing new customer/consumer needs and business opportunities to be identified from the delivery of new functionality.
CD.05	The software architecture allows the delivery (deployment) of functionalities independently.
CD.06	Consumers receive new functionalities frequently, including in shorter cycles than the time-box typically established in the development process.
CD.07	Clients can test the product as soon as new functionalities are deployed.
CD.08	The organization's business model is constantly evaluated and revised (when necessary) based on customer/consumer information.
CD.09	Marketing strategies are constantly evaluated and revised (when necessary) based on lead customers' information (customers/consumers most relevant to the organization).
CD.10	Sales strategies are constantly evaluated and revised (when necessary) based on lead customers' information (customers/consumers most relevant to the organization).
CD.11	Alignment between product development and the organization's business is maintained through continuous checks, in short cycles.
CD.12	Alignment between product development and the organization's business is maintained through continuous checks in short cycles, based on data.
CD.13	Data is collected for metrics that evaluate the continuous delivery process (e.g., number of releases, defect density in releases).
CD.14	Data produced in continuous deployment environments (e.g., release dates and software version delivered) are stored in one (or more) data repositories.
CD.15	The continuous deployment process is continuously evaluated and improved.
CD.16	Data stored in the data repository is used to improve the product and the continuous deployment process.
CD.17	The organization has practices to share knowledge related to continuous deployment (e.g., internal lectures, tutorials, knowledge repositories, guild implementation).

Table 32 – R&D as Innovation System Stage's Statements.

#	Statement
RD.01	Feedbacks (data and opinions) from customers/consumers are captured and stored in a customer/consumer data repository.
RD.02	Feedbacks (data and opinions) from customers/consumers are continuously and automatically captured.
RD.03	Feedbacks (data and opinions) from customers/consumers are used to improve products (enhance existing features and identify new ones).
RD.04	The organization identifies new business opportunities based on automatically captured customer/consumer feedbacks.
RD.05	Feedbacks (data and opinions) from customers/consumers are used for experimentation and innovation.
RD.06	Experiments (e.g., A/B tests) are conducted with customers/consumers to improve products.
RD.07	Technologies (e.g., cloud technologies) are adopted to enhance experimentation.
RD.08	The organization continually experiments new technologies and methodologies.

Table 32 – *Continued from previous page*

#	Statement
RD.09	The organization has a clear information flow between the strategic level and the development area, allowing customer/consumer data to be used in an aligned way in making technical and business decisions.
RD.10	Data from the customer/consumer data repository is used in decision making by the software development area.
RD.11	Data from the customer/consumer data repository is used in decision making by the business area.
RD.12	Alignment between product development and the organization's business is maintained through continuous checks, in short cycles and based on customer/consumer data.
RD.13	Knowledge management practices are adopted based on data from the customer/consumer data repository.