An Ontological View of Design in the Software Context

Murillo V. H. B. Castro, Monalessa P. Barcellos, Ricardo de A. Falbo

Ontology & Conceptual Modeling Research Group (NEMO)
Computer Science Department, Federal University of Espírito Santo
Vitória, ES - Brazil

murillo.castro@aluno.ufes.br, monalessa@inf.ufes.br

Abstract. *Design plays a central role in software development. Design as a verb refers to the act of designing something, while as a noun refers to a specification resulting from the act of designing. When talking about design of objects in general, mental and physical elements should be considered. However, due to the software abstract nature, in the software development context, some ontological aspects related to design as a noun are still not clear, such as: what is the nature of design specifications, design objects and their components and how are they connected to each other? In this paper, we present the Software Design Reference Ontology, which aims to provide a well-founded conceptualization of design as a noun in the software context.*

1. Introduction

Design is a key factor for software quality. It has been object of research for many years, but challenges continue to arise [Taylor and Van der Hoek 2007]. One of the topics still under discussion concerns the nature of software design and the relations between design and other aspects, such as requirements and code [Osterweil 2007]. Since the activity of designing shares many common characteristics across different fields, understanding the meaning of design in general helps to learn about software design [McPhee 1996]. On the other hand, software has an intangible and abstract nature, which makes software design exceptionally different from fields that deal with physical objects [Osterweil 2007]. Hence, to understand design in the software context, we must consider both, general aspects of design and specific aspects of software.

In general, the term "design" can have different meanings [McPhee 1996]. As a noun, it represents a specification or a plan for creating something (the design object). It can even be manifested by the object itself [Ralph and Wand 2009]. As a verb, it represents the activity or the process of creating a design [Ralph and Wand 2009]. Many definitions of software design, such as "a process of defining the architecture, components, interfaces, and other characteristics of a system or component" [ISO 2010], or "a process that is concerned with describing how a requirement is to be met by the design product" [Budgen 2003], focus on characterizing design as a verb, mentioning design as a noun merely as a result of the process. Consequently, some aspects of software design as a noun are not clear yet. For example, what is the essential nature and characteristics of a software design specification? How does it relate to software requirements and the implemented software? How can a software design specification address software objects or their parts, even when they do not exist yet? Questions like these call for a shared understanding of design (as a noun) in the software context.

Ontologies are useful in this matter, since they can be used for establishing a formal and common conceptualization of a domain of interest [Studer et al. 1998]. Previous works have already given some direction towards a shared understanding of design in general by using ontologies. For example, the works [Ralph and Wand 2009] and [Guarino 2014] have a similar view of the essential characterization of design by the existence of a specification detailing how intrinsic properties of the envisioned object and its parts should be obtained. The specification can be in the designer's mind, or be presented as a physical representation (e.g., a prototype), or even as the final object itself [Ralph and Wand 2009]. For example, if we consider a situation in which a software design team works on a new feature of an existing software, there are specifications in those three forms simultaneously. It is not clear if they are different instances of specification or instances of different types of specifications for the same object. Neither it is clear what are the relations between these specifications. Guarino and Melone (2015) extended the discussion about the ontological status of design objects and their components by defining what the authors call conventional system components (i.e., what designers have in mind to be expected to exist in a particular place of the design object, playing a specific role). Is this definition also suitable for software, considering its intangible and abstract nature? If so, how can we characterize conventional system components in this context?

Considering the need for better understanding software design conceptualization, in this paper we propose the *Software Design Reference Ontology* (SDRO). A reference ontology is a special kind of conceptual model; a solution-independent specification with the aim of making a clear and precise description of domain entities for the purposes of communication, learning and problem-solving [Guizzardi 2007]. SDRO is grounded in the Unified Foundational Ontology (UFO) [Guizzardi 2005] and is integrated to the Software Engineering Ontology Network (SEON) [Ruy et al. 2016]. By reusing SEON concepts, connections between requirements, design, coding and testing aspects can be represented, providing a more comprehensive view of software design in the software development context.

The remainder of this paper is organized as follows. Section 2 presents the background for the paper. Section 3 presents the Software Design Reference Ontology. Section 4 discusses how SDRO helps explain some design aspects in the software development context. Section 5 discusses related works and presents our final considerations.

2. Background

2.1. Design and Software Design

"Design" can be either a verb (e.g., *"to make or draw plans for something"* [Design 2020a]; *"to conceive and plan out in the mind"* [Design 2020b]) or a noun (e.g., *"a drawing or set of drawings showing how a product is to be made and how it will work and look"* [Design 2020a]; *"a mental project or scheme in which means to an end are laid down"* [Design 2020b]). One meaning does not exclude the other, rather, they are complementary and suggest different viewpoints of the design phenomenon.

In the Software Engineering literature, we can also find definitions referring to "software design" as a verb and as a noun. For example, it can be defined as the process of describing architecture, components, modules, interfaces, and data for a software system, to specify how requirements are to be met by the implemented software, being ap-

plied regardless of the software process model [Budgen 2003; ISO/IEC/IEEE 2017; Pressman and Maxim 2020]. The result of the design process is a description that acts as a blueprint for building the software, which is also referred to as the software design.

A more formal conceptualization of "design" as a verb and as a noun was proposed by Ralph and Wand (2009) and was based on a literature review of design definitions across different fields, summarizing what they have in common and trying to resolve disagreements. Design as a noun is defined by these authors as *"a specification of an object (the design object), manifested by an agent, intended to accomplish goals, in a particular environment, using a set of primitive components, satisfying a set of requirements, subject to constraints".* As a verb, design is defined as the process of creating a design [Ralph and Wand 2009]. Hence, a designer is the agent who manifests a specification. A specification, in turn, is a detailed description of a design object's structural properties and, it may be purely mental, presented as a physical or symbolic representation or as the object itself. This is in accordance with the aforementioned definitions (from the dictionary) and with the discussion provided by Guarino (2014). According to him, the design object is the thing being designed, which in the context of this work is software. It has essential characteristics that result from the design choices encoded in the design specification [Guarino 2014]. These choices involve the selection and manipulation of components (or primitives) that will compose the designed object. Guarino and Melone (2015) made a discussion about components referred by designers when designing and highlighted that they have a different ontological status from the physical components that constitute the realized design object. Those authors named these components as conventional system components, which represent components that designers have in mind, existing in a particular place of the object and playing a specific role. Goals, requirements, constraints and the design object environment are considered inputs to the design process [Ralph and Wand 2009] and are all encompassed by the term "requirements" in the software development context.

2.2. Ontologies

An ontology is a formal, explicit specification of a shared conceptualization [Studer et al. 1998]. According to Scherp et al. (2011), ontologies can be organized in a three-layered architecture. *Foundational ontologies* aim at modeling the very basic and general concepts and relations that make up the world (e.g., objects, events and parthood). *Core ontologies* provide a refinement to foundational ontologies by adding detailed concepts and relations in a specific area (such as service, organizational structure) that spans across various domains. Finally, *domain ontologies* make the best possible description of knowledge that is specific for a particular domain (such as a domain-specific medical ontology describing the anatomy of the human). Guizzardi (2007) points out another important distinction that differentiates ontologies as conceptual models representing a model of consensus within a community, regardless of its computational properties, called *reference ontologies*, from ontologies as computational artifacts (machine-readable ontologies), called *operational ontologies*.

The Unified Foundational Ontology (UFO). SDRO is grounded in UFO. Figure 1 shows the UFO fragment relevant to this paper. Concepts that are directly used in this work are highlighted in purple in the figure and are described in the following, based on [Guizzardi 2005] and [Guizzardi et al. 2008]. In the model description, UFO concepts are written in *italics*.

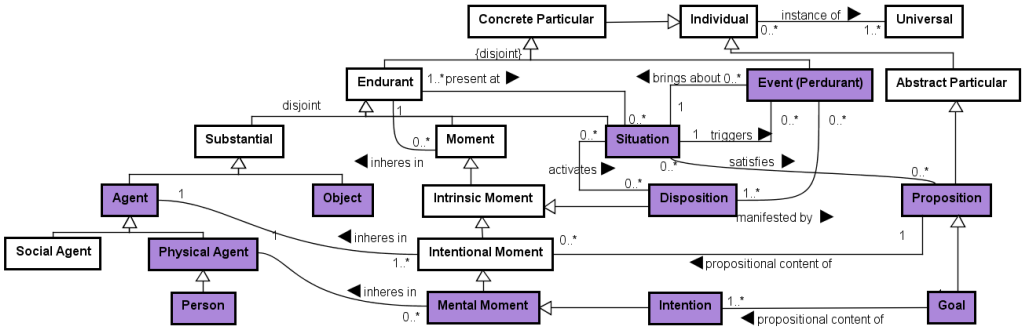

Figure 1 – UFO fragment relevant to this work.

The first fundamental distinction in UFO is between *Individuals* (particulars) and *Universals* (types). *Individuals* are entities that exist in reality, possess a unique identity (e.g., a person, a car) and instantiate *Universals*, which are patterns of features that can be realized in a number of different individuals (e.g., Person, Car). In this work, our focus is on *Individuals*, which can be *Abstract* or *Concrete*. *Abstract Individuals* include numbers, sets, propositions, among others. *Concrete Individuals* are partitioned into *Endurants* and *Perdurants (Events)*. *Endurants* are said to be wholly present whenever they are present (e.g., a person), while *Events (Perdurants)* are individuals composed of temporal parts (e.g., a soccer match). Among the types of *Endurants*, two are detached: *Substantials* and *Moments*. *Substantials* are existentially independent individuals (e.g., an apple), while *Moments*, in contrast, denote properties of individuals. *Situations* are special types of *Endurants* (i.e., complex entities constituted by possibly many *Endurants*, including other *Situations*) that represent a portion of reality that can be comprehended as a whole, also known as a state of affairs (e.g., John has the flu and a fever). A *Situation* may trigger an *Event*, which brings about a new Situation. *Intrinsic Moments* are moments dependent on one single individual (e.g., an apple's color) and *Dispositions* are types of *Intrinsic Moments* that are only manifested in particular *Situations* on the occurrence of certain triggering *Events* (e.g., a magnet attracting property triggered after approaching to a metal object).

A basic distinction of *Substantials* is related to *Agents* and (non-agentive) *Objects*. *Agents* are agentive substantial individuals that can be *Physical Agents* (e.g., a person) or *Social Agents* (e.g., an organization, a society). *Objects* are non-agentive substantial individuals that can also be physical (e.g., a book, a table) or social (e.g., money, language). *Agents* can bear special types of *Intrinsic Moments* named *Intentional Moments*. In this case, intentionality refers to the capacity of some properties of certain individuals to refer to possible situations in reality. Thus, *Intentional Moments* have a propositional content (*Proposition*), which is an abstract representation of a class of situations referred to by that *Intentional Moment*. A proposition can be satisfied by a *Situation* when the *Situation* actually occurs in the real world. *Mental Moments* are specialization of *Intentional Moments* referring to mental components of a *Physical Agent*. A specific type of *Mental Moment* is an *Intention*, which is the proper representation of "intending something" and has a *Goal* as its propositional content.

Software Engineering Ontology Network (SEON). For large and complex domains, ontologies can be organized in an ontology network (ON), which consists of a set of ontologies connected to each other through relationships in such a way to provide a comprehensive and consistent conceptualization [Suárez-Figueroa et al. 2012]. SDRO is

integrated to the Software Engineering Ontology Network (SEON) [Ruy et al. 2016], an ontology network that describes various subdomains of the Software Engineering domain (e.g., software requirements, coding, testing, software measurement, etc.). SDRO reuses concepts from three SEON ontologies, namely: *Software Process Ontology* (SPO) [Bringuente et al. 2011], *System and Software Ontology* (SysSwO) and *Software Requirements Ontology* (RSRO) [Duarte et al. 2018]. Figure 2 shows the integrated view of the concepts from these ontologies that are relevant for this work. In the figure, SPO concepts are presented in gray, SysSwO in green, RSRO in red and UFO in white. Blue relationships represent the grounding of concepts in UFO. In the description, UFO concepts are written in *italics* while **bold** is used in SEON concepts.

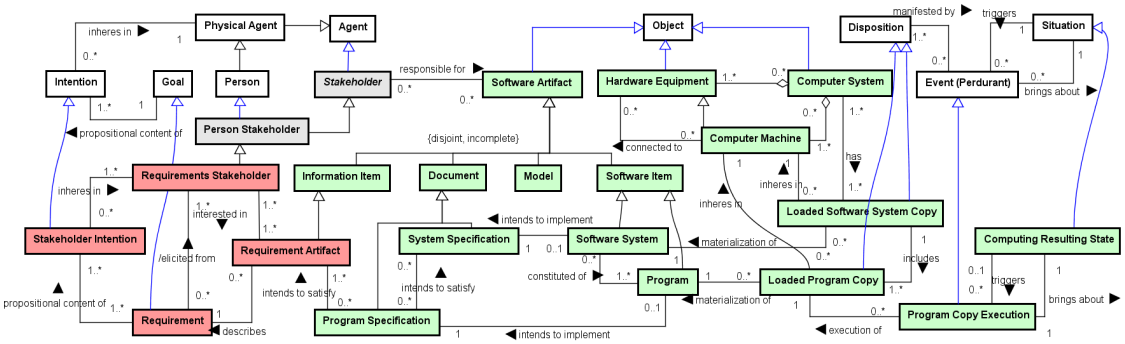

Figure 2 – SEON fragment relevant to this work.

A **Stakeholder** is an *Agent* interested in a particular software project. It can be a person, an organization, or a team. In the first case, it is called **Person Stakeholder**. A **Stakeholder** may be responsible for **Software Artifacts**, which are *Objects* intentionally produced to serve a given purpose in the context of a software project or organization. **Software Artifacts** can be classified according to their nature. A **Software Item** is a piece of software, produced during the software process, not considered a complete product, but an intermediary result (e.g., a component). A **Document**, in turn, is any written or pictorial, uniquely identified information related to the software development, usually presented in a predefined format (e.g., a requirements document). An **Information Item** is a piece of relevant information for human use, produced or used by an activity (e.g., a component description). A **Model** is a representation (abstraction) of a process or system from a particular perspective (e.g., a use case model, a class model).

A **Software System** (e.g., a system to buy airline tickets) is a **Software Item** constituted of **Programs** aiming to implement a **System Specification** (a subtype of **Document**). A **Program**, in turn, is also defined as a **Software Item**, a piece of software, produced during the software process, not considered a complete **Software System** (e.g., the system component to select available flights on a certain date). A **Program** aims at producing a certain result through execution on a computer, in a particular way, given by the **Program Specification**, which is a **Document** describing structural and functional information about the **Program** with enough detail that would allow implementation and maintenance. A **Hardware Equipment** is a *Physical Object* used to process, transform, store, display or transmit information or data. A **Hardware Equipment** that can run programs, process, transform and store data and information is a **Computer Machine**. A **Computer System** is a system containing one or more **Computer Machines** and other **Hardware Equipment** connected to them. A **Loaded Software System** is the materialization of a **Software System** (e.g., the system to buy air-

line tickets loaded in Mary's computer machine) as a complex *Disposition* inhering in a **Computer System**, including one or more **Loaded Program Copies**. A **Loaded Program Copy**, in turn, is the materialization of a **Program** (e.g., the component to select available flights in a certain date loaded in Mary's computer machine) as a *Disposition* manifested by a **Program Copy Execution** (*Event*). A **Program Copy Execution** (e.g., the execution of the program copy to show flights available on a certain date) brings about a **Computer Resulting State** (e.g., a set of flights), a *Situation* involving properties of the **Computer Machine** in which the **Loaded Program Copy** inheres, as well as of entities residing in that **Computer Machine** (including the **Loaded Program Copy** itself). A **Computer Resulting State** can trigger other **Program Copy Executions**.

A **Requirement** is a *Goal* in the sense of UFO, i.e., the propositional content of an *Intention* that inheres in a **Requirements Stakeholder**. When a **Requirement** is recorded in some kind of document, there is a **Requirement Artifact** describing that **Requirement**. A **Requirement Artifact** is an **Information Item** responsible for keeping relevant information for human use. **Requirements** are connected to implemented software through the following relation: a **Program Specification** intends to satisfy some **Requirement Artifacts**. Thus, a **Program** that intends to implement a **Program Specification** also intends to satisfy these **Requirement Artifacts**.

3. Software Design Reference Ontology (SDRO)

The Software Design Reference Ontology (SDRO) aims at providing a well-founded consensual conceptualization of software design (as a noun), describing the mental and physical elements involved in the design of software systems and the relations between them. Here, the term "physical" is borrowed from other works [Baker and Hoek 2006; Guarino 2014; Ralph and Wand 2009] referring to the perception of something through the senses. However, considering that software is abstract, there are differences in the way it is perceived when compared to physical objects like a chair or a car.

SDRO was developed by following SABiO [Falbo 2014], using the works by Ralph and Wand (2009) and Guarino (2014) as a reference to describe the core design notions and reusing concepts from SEON [Ruy et al. 2016] to address software particularities. SABiO was chosen because it has been successfully used to develop domain ontologies, in particular Software Engineering reference domain ontologies, including the ones already integrated to SEON and reused in this work. Moreover, by combining general notions of design and specific aspects of Software Engineering, we can provide a comprehensive conceptualization of software design integrated to other related aspects, such as requirements, coding and testing. SDRO allows the instantiation of software design situations regardless of the design paradigm, process, or method used in software development. It is focused on design as a noun, i.e., it is not concerned with describing activities involved in a general software design process, which is addressed in SEON by the Design Process Ontology [Ruy et al. 2016].

We defined the ontology scope by means of competency questions, i.e., questions the ontology must be able to answer and are used as a basis to develop the ontology conceptual model. Considering the ontology purpose, we raised the following set of Competency Questions (CQ): (CQ1) *How does a software designer reason about the object being designed?* (CQ2) *What is a software design specification and* (CQ3) *which are its components?* (CQ4) *What is a software design object and* (CQ5) *which are its*

components? (CQ6) *What is described in a software design specification?* (CQ7) *What is the motivation for a software design choice?* (CQ8) *How can a software design object be implemented from a software design specification?* (CQ9) *How can a software design object be evaluated against a software design specification?*

Considering the distinction between mental and physical aspects of software design, we divided SDRO into two sub-ontologies, namely the *Mental Aspects* sub-ontology and the *Physical Aspects* sub-ontology. Figure 3 shows the conceptual model of the *Mental Aspects* sub-ontology. In the figure, red double-dashed horizontal lines separate the layers of SEON architecture according to the classification proposed in [Scherp et al. 2011], while black single-dashed lines separate concepts from different ontologies at the same layer. Different colors are used to indicate concepts from different ontologies. For SEON and UFO concepts, we kept the same colors from Figure 2. Concepts from the *Mental Aspects* sub-ontology are presented in light violet. In the description, SEON concepts are written in **bold** and SDRO concepts in ***bold italics***.

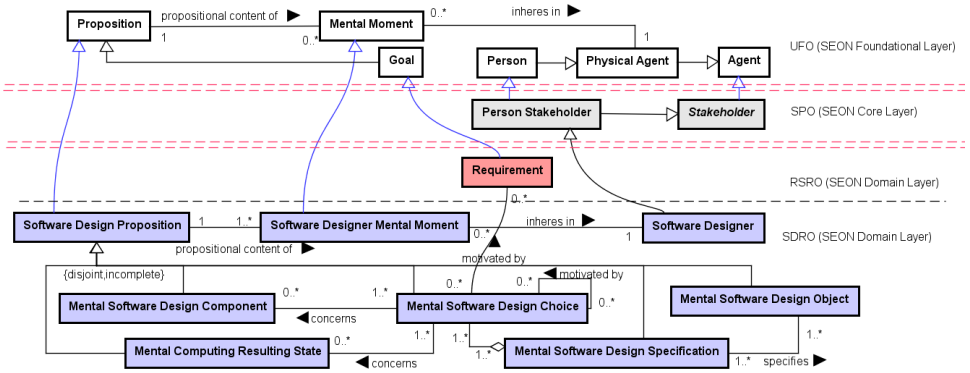

Figure 3 – SDRO Mental Aspects sub-ontology conceptual model.

Mental aspects are treated in SDRO as ***Software Designer Mental Moments*** that inhere in a ***Software Designer***, which is a **Person Stakeholder** that uses his/her skills to directly contribute to the outcome of the design effort. The propositional content of a ***Software Designer Mental Moment*** is a ***Software Design Proposition***, i.e., a sentence describing an idea in the designer's mind about software design aspects of a certain object. Thus, ***Software Designer Mental Moments*** represent mental properties of a ***Software Designer*** that enable him/her to imagine possible solutions for software design problems, while ***Software Design Propositions*** are the expression of such solutions in a conversation or in the designer internal dialogue. The realization of design solutions is given by situations in reality where the propositional content of ***Software Designer Mental Moments*** (i.e., ***Software Design Propositions***) are true. The number of design solutions that can be actually realized varies according to the intersection between desirable situations that satisfy the requirements and feasible situations that respect the constraints (e.g., time, cost, computational resources) [Baker and Hoek 2006].

Five different types of ***Software Design Propositions*** were identified, based on what their subject refers to. A ***Mental Software Design Object*** concerns the software object being designed, which may not exist yet (i.e., it represents what designers visualize as the final design object when they refer to "the system"). A ***Mental Software Design Object*** is specified by a ***Mental Software Design Specification***, a detailed description of the object structure (i.e., how the object should be decomposed and organized

into smaller elements and how these elements interact with each other). At the beginning of the design process, this description usually represents the object structure in a higher level of abstraction (e.g., architecture aspects) and, as design iterations occur, it is refined into more detailed representations (e.g., components and modules) at lower levels of abstraction [Pressman and Maxim 2020]. The ***Mental Software Design Specification*** consists of one or more ***Mental Software Design Choices*** made by the ***Software Designer***. Each ***Mental Software Design Choice*** can be motivated by **Requirements** or by other ***Mental Software Design Choices*** and contains details about a decision made by the ***Software Designer*** concerning structural or behavioral properties of the designed object or about its components and their connections. Thus, a ***Mental Software Design Choice*** may concern ***Mental Software Design Components***, which represent what the ***Software Designer*** expects to exist as a part of the designed object in a particular place, playing a specific role and having its own properties (e.g., modules, partitions and layers in which the system's architecture is organized), also referred as "conventional system components" by Guarino and Melone (2015). A ***Mental Software Design Choice*** may also concern a ***Mental Computing Resulting State***, which represents an expected result of a ***Mental Software Design Choice*** that can only be assessed in runtime (e.g., obtaining a certain return after the execution of a system's module).

The *Physical Aspects* sub-ontology is presented in Figure 4. In the figure, concepts from this sub-ontology are presented in dark violet. Physical aspects are treated in SDRO as sub-types of **Software Artifact**. For example, a ***Software Design Object*** is a **Software System** that implements one or more ***Mental Software Design Objects***. This does not imply that every ***Mental Software Design Object*** results in the development of a new **Software System**. An existing **Software System** is also considered a ***Software Design Object*** when it implements at least one ***Mental Software Design Object*** specified by a ***Mental Software Design Specification***. ***Software Design Objects*** are composed of ***Software Design Components***, which are **Programs** that play specific roles in the designed **Software System**, implementing ***Mental Software Design Components***. ***Software Design Components*** can be composed of sub-components, allowing for the representation of more complex architectures. A (sub)component can also be part of more than one component (e.g., in situations where the code is properly modularized and reused).

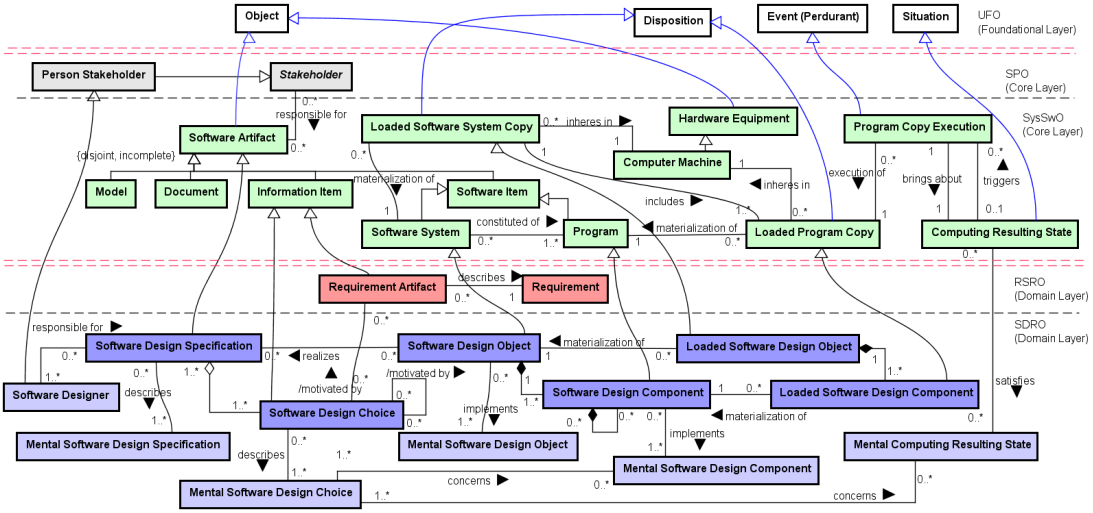

Figure 4 – SDRO Physical Aspects sub-ontology conceptual model.

In order to be executed and used, a ***Software Design Object*** and its ***Software Design Components*** must be materialized respectively as a ***Loaded Software Design Object*** (**Loaded Software System Copy**) and ***Loaded Software Design Components*** (**Loaded Program Copies**), which are respectively the resulting designed software and its constituent programs inhering in a **Computer Machine** (i.e., the software must be loaded in the computer's main memory). A ***Loaded Software Design Component*** can be executed as a **Program Copy Execution** that brings about a **Computing Resulting State**. If the program was implemented and executed correctly, this **Computing Resulting State** may satisfy a ***Mental Computing Resulting State*** associated in a ***Mental Software Design Choice*** with the corresponding ***Mental Software Design Component*** materialized by the executed ***Loaded Software Design Component***. This relationship allows us to verify if the implemented software meets the design specification.

A ***Software Design Specification*** is a **Software Artifact** created by one or more ***Software Designers***. It can be either a **Model** (e.g., a class diagram), a **Document** (e.g., a detailed textual description), or a **Software Item** (e.g., a functional prototype), providing an explicit representation that describes ***Mental Software Design Specifications*** (also referred to as Software Design Descriptions in IEEE 1016 [IEEE 2009]). A ***Software Design Specification*** is an aggregation of one or more ***Software Design Choices***, which are **Information Items** describing ***Mental Software Design Choices***. Therefore, a ***Software Design Choice*** is a piece of information that physically represents choices made by a ***Software Designer*** and can be used for communication and evaluation purposes (e.g., a sentence like *"The system will be implemented in Java"* or details added in a class diagram that indicates how entities and relations should be represented in the database). As a derived relation, a ***Software Design Choice*** is motivated by a **Requirement Artifact** when the ***Mental Software Design Choice*** it describes is motivated by the **Requirement** described by the **Requirement Artifact**. This connection establishes a traceability relation between design and requirements artifacts.

To evaluate SDRO, we performed Ontology Verification & Validation (V&V) activities by using two approaches: assessment by human approach and data-driven approach [Brank et al. 2005]. In the first, we performed a verification activity by means of expert judgment, in which we checked if the concepts and relations defined in SDRO were able to answer the competency questions. In the second, to validate SDRO, we instantiated its concepts and relations using data extracted from a real-world scenario. Table 1 presents part of the results of SDRO verification, which showed that the ontology answers all the CQs and, thus, is able to cover the scope established to it. The complete results of the verification can be found in the SDRO full specification available at https://bit.ly/SDRO-specification.

Table 1. SDRO verification against some of its CQs.

CQs	Description, **Concepts** and *Relations*
CQ2	What is a software design specification? **Software Design Specification** is a **Software Artifact** *created by* **Software Designers** that *describes* **Mental Software Design Specifications**.
CQ3	Which are the components of a software design specification? **Software Design Specification** *is composed of* **Software Design Choices**, which are Information Items that *describe* **Mental Software Design Choices**.
CQ6	What is described in a software design specification? **Software Design Specification** *describes* a **Mental Software Design Specification** which *specifies* a **Mental Software Design Object** and *is composed of* **Mental Software Design Choices**, which may *concern* to **Mental Software Design Components** or **Mental Computing Resulting States**.

For a brief **validation**, we took as an example of design object the car rental software system (here referred to as *CRS*) specified in [Falbo 2018] and used SDRO to instantiate and analyze a scenario considering the CRS design.

The *CRS* system aims to support rent-a-car companies in managing fleets of cars and rentals, as well as allowing customers to make car rentals via the internet. Based on this context, the following **Requirements** of *CRS* had been elicited: (*i*) a rent-a-car company wants to manage fleets of cars, customers and car rentals; and (*ii*) customers want to make car rentals via the internet. These **Requirements** were described and recorded as **Requirement Artifacts** in a requirements document. Two students, John and Mary, were responsible for designing *CRS* (i.e., playing the role of ***Software Designers***) and discussed how they would address these requirements in the system's implementation. In their discussions, they referred to the software system being designed (i.e., the ***Mental Software Design Object***) as "the system" or "*CRS*", corresponding to what they had in mind (i.e., ***Software Designer Mental Moments***) as a solution to satisfy the **Requirements**. John made some ***Mental Software Design Choices*** to meet the requirement (*ii*) and communicated them to Mary, proposing to implement the system in *Java* using libraries that can run in different browsers, and implement the user interface (a ***Mental Software Design Component*** in this context) as an independent component from the rest of the application. These ***Mental Software Design Choices*** were also related to ***Mental Computing Resulting States*** corresponding to John's visualization of the system's user interface being executed in different browsers. Mary agreed with John's suggestions and complemented that they could organize the system in an architecture based on a combination of partitions and layers, i.e., ***Mental Software Design Components*** related to each other in the Mary's ***Mental Software Design Choices***. She considered that the combination of the ***Mental Software Design Choices*** made by her with the ones made by John (i.e., Mary's ***Mental Software Design Specification***) was sufficient to start implementing the system. However, John had trouble in understanding the choices proposed by Mary (i.e., his ***Mental Software Design Specification*** was not equivalent to hers), so he asked her to produce a ***Software Design Specification*** describing what she had in mind. Mary presented to him a diagram encoding ***Software Design Choices*** (e.g., the representation of the partitions and layers as UML elements) describing the ***Mental Software Design Choices*** she had in mind. After seeing the diagram in the ***Software Design Specification*** produced by Mary, John understood what she proposed (i.e., their ***Mental Software Design Specifications*** became equivalent) and then they decided to implement the system. They presented the ***Software Design Specification*** to a developer and explained what they had in mind to him. The interpretation of the developer produced a ***Mental Software Design Specification*** in his mind, as well as other ***Software Design Propositions*** associated with it. After that, the developer produced a **Software System** written in *Java*, which satisfied the specification he had in mind (i.e., a ***Software Design Object*** implementing the ***Mental Software Design Object*** specified by his ***Mental Software Design Specification***). Then, he asked John and Mary to assess if the implemented software corresponded to what they had designed. John and Mary first inspected the code and observed that the partitions and layers had been correctly implemented as ***Software Design Components*** (i.e., the ***Software Design Object*** realized the ***Software Design Specification***). In the sequence, they loaded a copy of the system in the computer's main memory (a ***Loaded Software Design Object*** composed of ***Loaded Software Design Components***) and accessed the system's user interface through different browsers. Each of those accesses produced a **Program Copy Execu-**

tion of the ***Loaded Software Design Component*** that materialized the implementation of a particular layer of a specific partition (i.e., a ***Mental Software Design Component***), which, in turn, might have triggered the execution of the other components. When the execution was completed, it brought about **a Computing Resulting State** (e.g., an HTTP response code 200) that satisfied the ***Mental Computing Resulting State*** imagined by John. Based on that, John and Mary concluded that the ***Software Design Object*** had been correctly implemented according to their ***Mental Software Design Objects***.

4. Discussion

Since SDRO is integrated into SEON, it can also be used as a conceptual framework to discuss design aspects in a wider software development context, exploring questions such as: when a design succeeds or fails, the role of design documentation and the relation between software design and human-computer interaction.

A design effort is considered complete in SDRO when the design specification is realized, which can be satisfied by the following condition: if there are Software Design Components implementing all Mental Software Design Components concerned in all Mental Software Design Choices that are described in Software Design Choices of a Software Design Specification, then we can say that this Software Design Specification is realized by the Software Design Object composed of those Software Design Components. Hence, an incomplete design occurs when at least one Mental Software Design Component is not implemented. However, a complete design does not mean necessarily that the software is correctly implemented, since Software Design Choices that prescribe behaviors of the system can only be assessed when the system is running. Therefore, the correct implementation occurs when all Mental Computing Resulting States concerned by these choices are satisfied by at least a Computing Resulting State. An incomplete or incorrect implementation may happen when a programmer does not follow what was described in the Software Design Specification. In this case, the programmer's interpretation of the Software Design Choices probably was not the same as the designer's. Another reason could be that the Software Design Specification does not properly describe the Mental Software Design Specification created by the designer, maybe because the tools and the language used to create the specification were not adequate [Baker and Hoek 2006].

In SDRO, it is possible to represent a Software System (Design Object) developed without the existence of any physical Software Design Specification. This is the case where Ralph and Wand (2009) describe that the specification is presented as the Design Object itself. Although doing so could be considered a bad practice in software development, it addresses simpler situations where the designer creates the specification only in his mind and develops the system by himself or communicates the design verbally to the developer. Since software development often involves teams and more complex systems, the use of artifacts to represent design specifications is essential to allow for evaluation in earlier stages and communication of design ideas between designers and other stakeholders. Moreover, it also provides a form of reflexive conversation where the designer can have insights of improvements as she/he looks at the specification [Schön 1983; Simon 1996].

SDRO conceptualization also helps highlight what makes software and software design unique compared to other fields involving design: there is a large gap between

what is produced as the Software Design Object and what is perceived by the user in his/her interaction experience. The Software Design Object, a Software System constituted by code [Duarte et al. 2018], does not interact directly with the user as does a car or a house, for example. It must be loaded in a computer system (i.e., Loaded Software Design Object and Loaded Software Design Components) and then executed, so the user can interact with the result of this execution (i.e., Computing Resulting States). Therefore, software design methods, tools and languages should consider not only the internal structural aspects of software but also its external characteristics exhibited to the user. Traditionally, Software Engineering research is more focused on the former, while the latter is relegated to Human-Computer Interaction (HCI) studies [Taylor and Van der Hoek 2007]. Thus, to reduce the gap between software design and user experience, it is essential to have a holistic view of design and look at software as a whole.

5. Final Considerations

This paper presented SDRO, a reference ontology about software design grounded in UFO and integrated into a Software Engineering ontology network. From SDRO conceptualization we showed that software design has mental and physical natures since it involves elements from both perspectives. We discussed the composition of mental and physical design elements and the relations between them, and how design relates to requirements and code in a more comprehensive view of software development.

Concerning related works, [Ralph and Wand 2009] and [Guarino 2014] discussed ontological aspects of design in general, but did not explain the ontological distinction between design specifications that are purely mental and the ones encoded in artifacts, for example. In addition, specific aspects related to the abstract nature of the design object in the software development context (e.g., how developed software can be evaluated against its design specification) were also not addressed by these works as we did in SDRO. Some works, such as [Baker and Hoek 2006], [Ralph 2015] and [Gero 1990; Gero and Kannengiesser 2014], also addressed design in general and did not discuss further details about design specifications and the composition of design objects. On the other hand, other works proposed ontologies addressing software design in specific contexts, namely domain-driven design [Saiyd et al. 2009], model-based design [De Medeiros et al. 2005] and design intent [Solanki 2015], and therefore did not provide a comprehensive conceptualization of software design as SDRO does. It is worth noting that, since we reused concepts from SEON ontologies, we have committed to the conceptualization provided by them. Similar concepts (e.g., document, software) are also addressed by other works, such as the Information Artifact Ontology [IAO 2020], the Ontology4 Language Ontology [Ontology4 2021] and the Functional Requirements for Bibliographic Records [IFLA 2009].

As for SDRO applications, we have specialized it and created a more specific ontology to address HCI design, which was integrated into an HCI ontology network, allowing us to connect both Software Engineering and HCI domains. That ontology was used in the development of a tool to support knowledge sharing in HCI design [Castro et al. 2021]. As future work, we intend to improve SDRO validation by using formal validation techniques (e.g., using Alloy) and instantiating other scenarios from real-world situations. By taking advantage of the existing connections between SDRO and other ontologies of SEON, we also plan to use SDRO in semantic documentation of design artifacts aiming to improve the traceability between requirements, code and tests.

References

Baker, A. and Hoek, A. Van der (2006). Examining Software Design from a General Design Perspective.

Brank, J., Grobelnik, M. and Mladenić, D. (2005). A survey of ontology evaluation techniques. In *In In Proceedings of the Conference on Data Mining and Data Warehouses (SiKDD 2005)*.

Bringuente, A. C. de O., Falbo, R. de A. and Guizzardi, G. (2011). Using a Foundational Ontology for Reengineering a Software Process Ontology. *Journal of Information and Data Management*, v. 2, n. 3, p. 511–526.

Budgen, D. (2003). *Software design*. 2nd ed ed. Harlow, England ; New York: Addison-Wesley.

Castro, M. V. H. B., Barcellos, M. P., Falbo, R. de A. and Costa, S. D. (2021). Using Ontologies to aid Knowledge Sharing in HCI Design. In *XX Brazilian Symposium on Human Factors in Computing Systems (IHC'21)*. . ACM, New York, NY, USA.

De Medeiros, A. P., Schwabe, D. and Feijó, B. (2005). Kuaba Ontology: Design Rationale Representation and Reuse in Model-Based Designs. [L. Delcambre, C. Kop, H. C. Mayr, J. Mylopoulos, & O. Pastor, Eds.]In *Conceptual Modeling – ER 2005*. , Lecture Notes in Computer Science. Springer.

Design (2020a). Design meaning in the Cambridge English Dictionary.

Design (2020b). Definition of Design in Merriam-Webster Dictionary.

Duarte, B. B., De Castro Leal, A. L., Falbo, R. D. A., et al. (2018). Ontological foundations for software requirements with a focus on requirements at runtime. *Applied Ontology*, v. 13, n. 2, p. 73–105.

Falbo, R. de A. (2014). SABiO: Systematic Approach for Building Ontologies. In *CEUR Workshop Proceedings*. , CEUR Workshop Proceedings. CEUR-WS.org.

Falbo, R. de A. (2018). Car Rental Software Design Specification v1.0. http://www.inf.ufes.br/~jssalamon/wp-content/uploads/disciplinas/projsistsoft/trabalho/Documento_Projeto_v1.0.pdf, [accessed on Jul 26].

Gero, J. S. (1990). Design Prototypes: A Knowledge Representation Schema for Design. *AI Magazine*, v. 11, n. 4, p. 26.

Gero, J. S. and Kannengiesser, U. (2014). The Function-Behaviour-Structure Ontology of Design. In: Chakrabarti, A.; Blessing, L. T. M.[Eds.]. . *An Anthology of Theories and Models of Design: Philosophy, Approaches and Empirical Explorations*. London: Springer. p. 263–283.

Guarino, N. (2014). Artefactual Systems, Missing Components and Replaceability. In: Franssen, M.; Kroes, P.; Reydon, T. A. C.; Vermaas, P. E.[Eds.]. . *Artefact Kinds: Ontology and the Human-Made World*. Cham: Springer International Publishing. p. 191–206.

Guarino, N. and Melone, M. R. S. (2015). On the Ontological Status of Design Objects. [F. A. Lisi & S. Borgo, Eds.]In *AIDE@ AI* IA*. , CEUR Workshop Proceedings. CEUR-WS.org.

Guizzardi, G. (2005). Ontological foundations for structural conceptual models. Telematica Instituut / CTIT.

Guizzardi, G. (2007). On Ontology, ontologies, Conceptualizations, Modeling Languages, and (Meta)Models. In *Proceedings of the 2007 conference on Databases and Information Systems IV: Selected Papers from the Seventh International Baltic Conference DB\&IS'2006*. . IOS Press.

Guizzardi, G., Falbo, R. de A. and Guizzardi, R. (2008). Grounding Software Domain Ontologies in the Unified Foundational Ontology (UFO): The case of the ODE Software Process Ontology. In *CIbSE*.

IAO (2020). Information Artifact Ontology. http://purl.obolibrary.org/obo/iao.owl, [accessed on Oct 15].

IEEE (2009). *IEEE Std 1016-2009 (Revision of IEEE Std 1016-1998), IEEE Standard for Information Technology—Systems Design—Software Design Descriptions*. v. 2009

IFLA (2009). *Functional Requirements for Bibliographic Records: Final Report*. IFLA.

ISO/IEC/IEEE (2017). ISO/IEC/IEEE 24765 - Int. Standard - Systems and software engineering Vocabulary.

ISO (2010). *24765-2010 - ISO/IEC/IEEE International Standard - Systems and software engineering -- Vocabulary*. Place of publication not identified: IEEE.

McPhee, K. (1996). Design Theory and Software Design.

Ontology4 (2021). Language Ontology. https://ontology4.us/english/Ontologies/Language%2520Ontology/Documents/index.html, [accessed on Oct 15].

Osterweil, L. J. (2007). A Future for Software Engineering? In *Future of Software Engineering (FOSE '07)*.

Pressman, R. S. and Maxim, B. R. (2020). *Software engineering: a practitioners approach*. 9th. ed. McGraw Hill.

Ralph, P. (2015). The Sensemaking-Coevolution-Implementation Theory of software design. *Science of Computer Programming*, Towards general theories of software engineering. v. 101, p. 21–41.

Ralph, P. and Wand, Y. (2009). A Proposal for a Formal Definition of the Design Concept. In *Design Requirements Eng.: A Ten-Year Perspective*. . Springer.

Ruy, F. B., Falbo, R. de A., Barcellos, M. P., Costa, S. D. and Guizzardi, G. (2016). SEON: A Software Engineering Ontology Network. In *Knowledge Eng. and Knowledge Management*. . Springer.

Saiyd, N. Al, Said, I. Al and Neaimi, A. Al (2009). Towards an ontological concepts for domain-driven software design. In *2009 First International Conference on Networked Digital Technologies*.

Scherp, A., Saathoff, C., Franz, T. and Staab, S. (2011). Designing core ontologies. *Applied Ontology*, v. 6, n. 3, p. 177–221.

Schön, D. A. (1983). *The reflective practitioner: how professionals think in action*. New York: Basic Books.

Simon, H. A. (1996). *The sciences of the artificial*. 3rd ed ed. Cambridge, Mass: MIT Press.

Solanki, M. (2015). DIO: A Pattern for Capturing the Intents Underlying Designs. In *Proceedings of the 6th Workshop on Ontology and Semantic Web Patterns (WOP 2015)*. , CEUR Workshop Proceedings. CEUR-WS.org.

Studer, R., Benjamins, V. R. and Fensel, D. (1998). Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*, v. 25, n. 1, p. 161–197.

Suárez-Figueroa, M. C., Gómez-Pérez, A., Motta, E. and Gangemi, A. (2012). *Ontology Engineering in a Networked World*. Springer.

Taylor, R. N. and Van der Hoek, A. (2007). Software Design and Architecture: The once and future focus of software engineering. In *Future of Software Engineering (FOSE '07)*. . IEEE.