Ontology Patterns: Clarifying Concepts and Terminology

Ricardo A. Falbo[1], Giancarlo Guizzardi[1,2], Aldo Gangemi[2], Valentina Presutti[2]

[1]Federal University of Espírito Santo, Vitória, Brazil
{falbo, gguizzardi}@inf.ufes.br
[2]ISTC, National Research Council, Italy
{aldo.gangemi,valentina.presutti}@cnr.it

Abstract. Ontology patterns have been pointed out as a promising approach for ontology engineering. The goal of this paper is to clarify concepts and the terminology used in Ontology Engineering to talk about the notion of ontology patterns taking into account already well-established notions of patterns in Software Engineering.

Keywords: ontology pattern, ontology design pattern, ontology engineering

1 Introduction

Although nowadays ontology engineers are supported by a wide range of ontology engineering methods and tools, building ontologies is still a difficult task even for experts [1]. In this context, reuse is pointed out as a promising approach for ontology engineering. Ontology reuse allows speeding up the ontology development process, saving time and money, and promoting the application of good practices [2]. However, ontology reuse, in general, is a hard research issue, and one of the most challenging and neglected areas of ontology engineering [3]. The problems of selecting the right ontologies for reuse, extending them, and composing various ontology fragments have not been properly addressed yet [4].

Ontology patterns (OPs) are an emerging approach that favors the reuse of encoded experiences and good practices. OPs are modeling solutions to solve recurrent ontology development problems [5]. Experiments, such as the ones presented in [4], show that ontology engineers perceive OPs as useful, and that the quality and usability of the resulting ontologies are improved. However, compared with Software Engineering where patterns have been used for a significant time [6, 7, 8], patterns in Ontology Engineering are still in infancy. The first works are from the beginning of the 2000s (e.g. [9, 10]), and only recently this approach has gained more attention, especially by the communities of Ontology Engineering [2, 3, 4, 5] and Semantic Web [1, 11].

In this paper, we discuss the notion of ontology pattern by means of an analogy to the notion of pattern in Software Engineering. A premise underlying the discussion made in this paper is that, for developing a domain ontology, an ontology engineer should follow an ontology development process that is quite similar to the software development process in Software Engineering. i.e., in our view, a domain ontology should be developed following an ontology development process comprising activities such as ontology requirements elicitation, ontology conceptual modeling, ontology design, ontology implementation, and ontology testing.

This paper is organized as follows. In Section 2, we present some important pattern-related concepts as used in Software Engineering, and also the current view of patterns in Ontology Engineering. In Section 3, we revisit some notions related to ontology patterns by means of an analogy to patterns in Software Engineering. In Section 4, we show, by means of examples, how some types of ontology patterns can be used during the ontology development process. Finally, in Section 5, we present our final considerations.

2 Software Engineering Patterns and Ontology Patterns

Patterns, in general, are vehicles for encapsulating knowledge. They are considered one of the most effective means for naming, organizing, and reasoning about design knowledge. “Design knowledge” in this sentence is applied in a general sense, meaning design in several different areas, such as Architecture and Software Engineering. According to [12], “a pattern describes a particular recurring design problem that arises in specific design contexts and presents a well-proven solution for the problem. The solution is specified by describing the roles of its constituent participants, their responsibilities and relationships, and the ways in which they collaborate”.

In Software Engineering, patterns help to alleviate software complexity in several phases of the software development process [13]. During the software development process, regardless of the method or process model adopted, there are some activities that should be performed, namely: requirements elicitation, conceptual modeling, architectural design, detailed design, and implementation. There are different types of patterns covering different abstraction levels related to these phases. Analysis patterns are to be used during conceptual modeling. They describe how to model (in the conceptual level) a particular kind of problem in an application domain. They comprise of conceptual model fragments that represent knowledge of the problem domain, and their goal is to aid developers in understanding the problem rather than showing how to design a solution. According to [14], there are two main types of analysis patterns: domain-specific and domain-independent analysis patterns. Domain-specific analysis patterns model problems that only appear in specific domains. They capture the core knowledge related to a domain-specific problem and, therefore, they can be reused to model applications that share this core knowledge. On the other hand, domain-independent analysis patterns capture the core knowledge of atomic notions that are not tied to specific application domains and, hence, can be reused to model the same notions whenever they appear in any domain. Patterns such as the ones proposed by Fowler [7] are examples of analysis patterns. Architectural patterns describe selected types of components and connectors (the generalized constituent elements of all software architectures) together with a control structure that governs system execution [15]. They can be seen as templates for concrete software architectures [8], and thus are to be used during the architectural design phase. Several of the patterns proposed in the Pattern Oriented Software Architecture (POSA) approach [8], and most of the patterns presented in [16] are architectural patterns. Design Patterns provide a scheme for refining subsystems or components of a software system, or the relationships between them. They describe commonly-recurring structures of communicating components that solves general design problems within a particular context [6]. Design patterns are medium-scale patterns. They are smaller in scale than architectural patterns, but they are independent of a particular programming language (implementation-independent patterns) [8]. Moreover, they are used in the detailed design phase. The Gang of Four (GoF) patterns [6] are the most known examples of a design patterns catalog. Finally, idioms (or programming patterns) represent the lowest-level patterns. They are specific to a programming language (patterns at a source code level). An idiom describes how to implement particular aspects of components or the relationships between them, using the features of a given language [8]. Idioms are used in the implementation phase. Coplien's C++ patterns [17] are examples of idioms.

The Ontology Engineering community has also tackled the notion of patterns, especially for aiding developing domain ontologies. Domain ontologies aim at describing the conceptualization related to a given domain, such as electrocardiogram in medicine [18].

According to [3], an Ontology Design Pattern (ODP) is a modeling solution to solve a recurrent ontology design problem. Gangemi and Presutti [3] have identified several types of ODPs, and have grouped them into six families: Structural, Reasoning, Correspondence, Presentation, Lexico-Syntactic, and Content ODPs.

Structural ODPs include Logical and Architectural ODPs. Logical ODPs provide solutions for solving problems of expressivity, while architectural ODPs affect the overall shape of the ontology either internally or externally. Reasoning ODPs inform about the state of an ontology, and let a system decide what reasoning has to be performed on the ontology in order to carry out queries and evaluation, among others. Correspondence ODPs include Reengineering ODPs and Mapping ODPs. Reengineering ODPs provide solutions to the problem of transforming a conceptual model (which can even be a non-ontological resource) into a new ontology. Mapping ODPs are patterns for creating semantic associations between two existing ontologies. Presentation ODPs deal with usability and readability of ontologies from a user perspective. They are meant as good practices that support the reuse of ontologies by facilitating their evaluation and selection (e.g. naming conventions). Lexico-syntactic ODPs are linguistic structures or schemas that consist of certain types of words following a specific order, and that permit to generalize and extract some conclusions about the meaning they express. They are useful for associating simple Logical and Content ODPs with natural language sentences, e.g., for didactic purposes. Finally, Content ODPs are small fragments of ontology conceptual models that address a specific modeling issue, and can be directly reused by importing them in the ontology under development. They provide solutions to domain modeling problems [3].

As pointed by Gangemi [19], Content ODP can extract a fragment of either a foundational or a core ontology, which constitutes its background. Based on this fact, Falbo et al. [20] consider two types of Content ODPs: Foundational ontology patterns, which are extracted from foundational ontologies, and tend to be more generally applied, and Domain-related ontology patterns, which are domain-specific patterns, and thus are applicable to solve problems in specific domains.

The Manchester’s Ontology Design Patterns Catalog [21] is a public catalog of ODPs focused on the biological knowledge domain. In this catalog, there types of ODPs are considered, namely [21, 22]: Extensional ODPs, which provide ways of extending the limits of the chosen knowledge representation language; Good practice ODPs, which are used to produce more modular, efficient and maintainable ontologies; and Domain Modeling ODPs, which are used to model a concrete part of the knowledge domain.

In contrast with the case of patterns in Software Engineering, patterns in Ontology Engineering are not properly linked to the development phase in which they can be applied. Moreover, it is important to highlight that the term “design” in “ontology design patterns” does not have the same meaning of “design” in “design patterns” of Software Engineering. In “ontology design patterns”, the term “design” is applied in a general sense, meaning the creation (building) of the ontology. In Software Engineering, in the other hand, “design” refers to the software development phase in which developers cross the border from the problem space to the solution space. While requirements elicitation and analysis deal with the problem domain, aiming at understanding the problem to be solved and its domain, in the design phase, the focus is on providing a solution, what requires taking technological aspects into account. Conceptual models, built during requirements analysis, are only concerned with modeling a view of the domain for a given application, and thus are independent of the technology to be applied in the solution. Design models, on the other hand, are committed to translating the conceptual view to the most suitable implementation according to the underlying implementation environment and also considering a number of non-functional (technological) requirements (such as efficiency, usability, reliability, portability, etc.). Thus, designers should know a priori features of the implementation environment to properly address the non-functional requirements in a given solution. In fact, the same conceptual model can lead to several design solutions, and the design phase involves choosing the most adequate solution for the problem.

Guizzardi [23] defends an analogous process for Ontology Engineering. In an ontology conceptual modeling phase, a *reference domain ontology* should be produced, whose aim is to make a clear and precise description of the domain elements for the purposes of communication, learning and problem solving. Reference ontologies are to be used in an off-line manner to assist humans in tasks such as meaning negotiation and consensus establishment. In the design phase, this conceptual specification should be transformed into a design specification by taking into account a number of issues ranging from architectural issues and non-functional requirements, to target a particular implementation environment. The same reference ontology can potentially be used to produce a number of (even radically) different designs. Finally, in the implementation phase, an ontology design is coded in a target language to be then deployed in a computational environment. This implementation version is frequently termed an *operational ontology* [20]. Unlike reference ontologies, operational ontologies are not focused on representation adequacy, but are designed with the focus on guaranteeing desirable computational properties. A design phase, thus, is necessary to bridge the gap between the conceptual modeling of reference ontologies and the coding of them in terms of a specific operational ontology language (such as, for instance, OWL and RDFS, but other DL-based languages [24], Datalog-based languages [25], relational databases [26], etc.). Issues that should be addressed in the design phase are, for instance: determining how to deal with the differences in expressivity of the languages that are used in each of these phases; or how to produce lightweight specifications that maximize specific non-functional requirements, such as reasoning performance. Figure 1 illustrates this Ontology Engineering view.

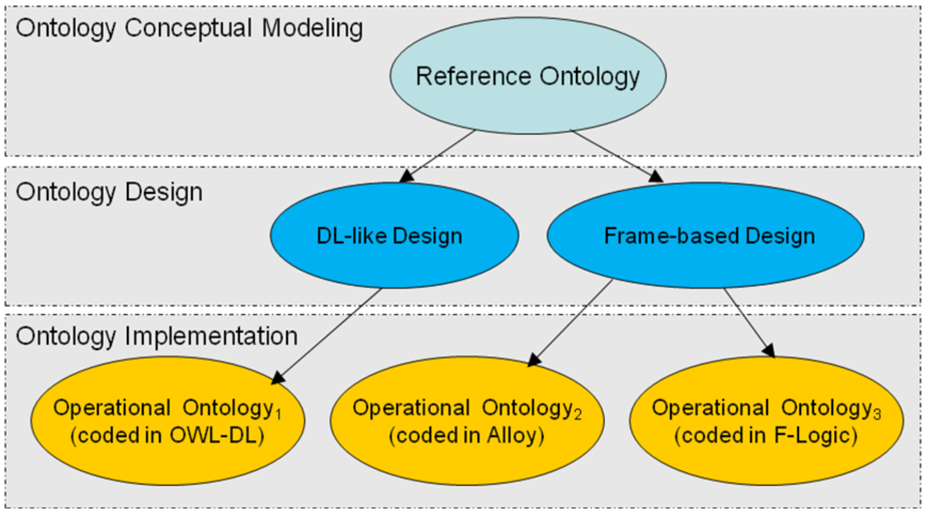
Ontology Conceptual Modeling
Reference Ontology
Ontology Design
DL-like Design
Frame-based Design
Ontology Implementation
Operational Ontology$_1$ (coded in OWL-DL)
Operational Ontology$_2$ (coded in Alloy)
Operational Ontology$_3$ (coded in F-Logic)

Fig. 1. Ontology Engineering as a Software Development Process

Based on this view of Ontology Engineering, in the next section, we revisit the notion of ontology patterns in order to clarify some concepts and the terminology used. Moreover, we compare the types of ontology patterns identified by Gangemi and Presutti [3] with the types of patterns related to the development phases in Software Engineering.

3 Ontology Patterns: Aligning Concepts and Terminology to Software Engineering Patterns

Once we have discussed a view of ontology development that is analogous to the well established view of software development in Software Engineering, we can revisit the notion of Ontology Pattern. Gangemi and Presutti [3] define Ontology Design Pattern as a modeling solution to solve a recurrent ontology design problem. As discussed in the previous section, "design" in this definition is used in a broad sense. In order to avoid confusion with the term "design" referring to the "design phase" of ontology development, we prefer to name patterns applied to Ontology Engineering as Ontology Patterns (instead of Ontology Design Patterns).

We can now further elaborate the definition of ontology pattern: *an Ontology Pattern (OP) describes a particular recurring modeling problem that arises in specific ontology development contexts and presents a well-proven solution for the problem*.

Taking this definition into account, we can now look at the types of ontology patterns identified by Gangemi and Presutti [3], and try to introduce them in another classification of ontology patterns that considers the ontology development phase depicted in Figure 1.

First, there are some types of patterns in [3] that, according to the definition presented above, do not qualify as OPs. They capture best practices, but do not conform to this definition. This is the case, for instance, of the lexico-syntatic and presentation patterns, which do not refer to a modeling problem. Mapping OPs also do not fit well to the definition above, since they do not address a modeling problem in a specific ontology development context. Mapping OPs are useful for integrating ontologies, and not for developing a new one. Although presentation OPs cannot be properly said to be ontology pattern, they have a counterpart in Software Engineering: naming conventions. In Software Engineering, there are name conventions that apply to different development phases. For instance, language-specific naming conventions, such as Java naming conventions, are to be applied during the implementation phase; naming conventions for classes, attributes and operations in general can be applied during conceptual modeling and design phases. Thus, analogously, ontology name conventions that are language-independent are to be applied during ontology conceptual modeling and design phases; language-specific ontology name conventions (such as OWL name conventions) are to be used during ontology implementation.

The Reengineering OP type is a case apart. Reengineering OPs are defined as transformation rules applied in order to create a new ontology (target model) starting from elements of a source model [3]. Based on this definition, we can say that they can be applied in several ontology development phases. However, most of the existing Reengineering patterns are language-dependent, such as patterns to transform non-OWL models to OWL DL operational ontologies, or refactoring patterns for refactoring an existing OWL DL source ontology into a new OWL DL target ontology. Such alleged patterns are, in fact, not proper patterns but Idioms.

In the other hand, content, architectural, logical and reasoning OPs can be related to ontology development phases. Content OPs are analogous to analysis patterns in Software Engineering; Architectural OPs to architectural patterns; and Logical and Reasoning OPs to design patterns, although some of them are, again, in fact, idioms.

Regarding the Manchester's Catalog, according to [21], "this catalog is generated from OWL files". Thus, its patterns better classify as Idioms.

Figure 2 shows a taxonomy of OPs, reorganizing some of the ontology pattern types identified in [3], and introducing subtypes of Content OPs (renamed as Conceptual OPs), as defined in [20].

Ontology Conceptual Patterns are fragments of either foundational ontologies (Foundational OPs) or domain reference ontologies (Domain-related OPs). They are to be used during the ontology conceptual modeling phase, and focus only on conceptual aspects, without any concern with the technology or language to be used for deriving an operational ontology. Ontology Conceptual Patterns are analogous to Analysis Patterns in Software Engineering. Foundational OPs are analogous to Domain-independent Analysis Patterns, while Domain-related OPs are analogous to Domain-specific Analysis Patterns.

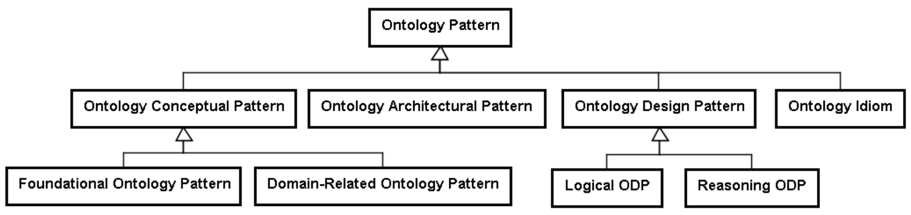
Ontology Pattern
Ontology Conceptual Pattern
Ontology Architectural Pattern
Ontology Design Pattern
Ontology Idiom
Foundational Ontology Pattern
Domain-Related Ontology Pattern
Logical ODP
Reasoning ODP

Fig. 2. Ontology Pattern Types

Foundational OPs (FOPs) are reusable fragments of foundational ontologies. Since foundational ontologies span across many fields and model the very basic and general concepts and relations that make up the world [18], FOPs can be applied in any domain. They are reused by analogy between the pattern and the problem in hand. An example of a FOP is the pattern for the problem of specifying roles with multiple disjoint allowed types, which were extracted from the ontology of substantial universals of the Unified Foundational Ontology (UFO) [27]. Another example is the appointment pattern, which were extracted from the ontology of social entities of UFO [28]. These two FOPs and how they can be reused are discussed in Section 4.

Domain-related OPs (DROPs) are reusable fragments extracted from reference domain ontologies. DROPs should capture the core knowledge related to a domain, and thus they can be seen as fragments of a core ontology of that domain. In contrast with FOPs, DROPs are reused by extension, i.e. concepts and relations of the pattern are specialized when the pattern is reused. In Section 4, we present a DROP for the domain of software processes and discuss its reuse by extension in the development of a software testing ontology.

Ontology Architectural Patterns are patterns that describe how to arrange an ontology (generally a large one) in terms of sub-ontologies or ontology modules, as well as patterns that deal with the modular architecture of an ontology network, where the involved ontologies play the role of modules [3]. These patterns can be used both during the conceptual modeling phase, and at the beginning of the ontology design phase. Since modularity is recognized as an important quality characteristic of good ontologies, we advocate for their use since the first stages of ontology development, for splitting the ontology into smaller parts, allowing tackling the problems one at a time. When applied at the beginning of the design phase, the purpose is to reorganize the ontology modules for addressing technological aspects, in special by taking non-functional requirements into account.

Ontology Design Patterns (ODPs) are patterns that address problems that occur during the ontology design phase. Based on the Gangemi and Presutti's taxonomy of types of ontology patterns [3], we identified two main types of ODPs: logical and reasoning ODPs. Reasoning ODPs, as the name suggests, aims at addressing specific design problems related to improving reasoning with ontologies (and qualities related to reasoning, such as computational tractability, decidability and reasoning performance). Logical ODPs, in turn, regards problems related to the expressivity of the formalism to be used in ontology implementation. They help to solve design problems that appear when the primitives of the implementation language do not directly support certain logical constructs. Logical ODPs are extremely important for ontology design, since most languages for coding operational ontologies are not focused on representation adequacy, but are designed with the focus on guaranteeing desirable computational properties [23]. We should highlight, however, that many patterns that address reasoning and logical problems are, in fact, Ontology Idioms (or Ontology Coding Patterns), since they describe how to solve problems related to reasoning or to expressivity of a specific logical formalism (e.g. OWL). According to the notion defended here, ODPs are more widely applied than ontology idioms, since they address problems related to various languages that share the same characteristics. For instance, a pattern addressing the problem of how to express n-ary relation semantics by only using class and binary relation primitives is a Logical ODP. It applies to all logical formalisms that do not have a construct for representing n-ary relationships, but have constructs for representing classes and binary relations. A pattern addressing the problem of how to express n-ary relation semantics in OWL is an Ontology Idiom.

Figure 3 shows the relationships between the types of ontology patterns shown in Figure 2 and the ontology development activities, shown in Figure 1. In the next section, we illustrate some Ontology Conceptual Patterns, and discuss how they can be reused.

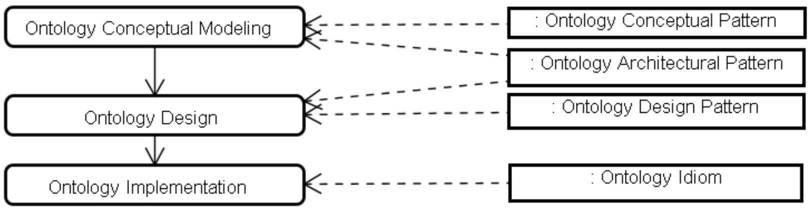
Ontology Conceptual Modeling
Ontology Design
Ontology Implementation
: Ontology Conceptual Pattern
: Ontology Architectural Pattern
: Ontology Design Pattern
: Ontology Idiom

Fig. 3. Applicability of Ontology Patterns with respect to Ontology Development Phases

4 Reusing Ontology Conceptual Patterns

There are two main ways of reusing ontology patterns: by analogy and by extension. Moreover, several patterns can be reused when developing a domain ontology. Thus, pattern composition is also an important mechanism for combining patterns.

In reuse by analogy [29], with an ontology modeling problem at hands, we look for OPs that describe knowledge related to the type of situation we are facing. Once selected the pattern, we have to identify which concepts in our domain correspond to the concepts in the pattern, and we reproduce the structure of the pattern in the domain ontology being developed. Most of the OPs are reused by analogy (including most Ontology Architectural and Design Patterns, Idioms, but also Foundational Ontology Patterns (FOPs)). Domain-related OPs (DROPs), in turn, are typically reused by extension. In reuse by extension, the OP is incorporated in the domain ontology being developed, and it can be extended by means of specialization of its concepts and relations, and also by including new properties and relationships with the extended concepts.

Figure 4 shows the conceptual model of the FOP for the problem of specifying roles with multiple disjoint allowed types [27]. In this picture, the abstract class A is the *role mixin* that covers different role types. Classes B and C are the disjoint subclasses of A that can have direct instances, representing the *roles* (i.e., sortal, anti-rigid and relationally dependent types) that carry the principles of identity that govern the individuals that fall in their extension. Classes D and E are the ultimate *kinds* that supply the principles of identity carried by B and C, respectively. The association R represents the common specialization condition of B and C, which is represented in A. Finally, class F represents a type that class A is *relationally dependent* of. *Kind*, *Role* and *Role mixin* are concepts from the Unified Foundational Ontology (UFO), part A, an ontology of endurants. This pattern is also embedded as a higher-granularity modeling primitive in the ontology-driven conceptual modeling language OntoUML. For details, see [27].

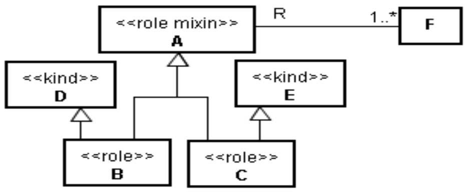

Fig. 4. Foundational OP for roles with multiple and disjunctive kinds [27].

Figure 5 shows two FOPs extracted from UFO-C, an ontology of social entities [28], namely: atomic/complex commitment types and appointments. The Atomic/Complex Commitment Types pattern, as the name suggests, models the distinction between atomic and complex commitments. According to UFO-C, *Commitments*, as intentional moments, inhere in *Agents*. Commitments can be atomic (*Atomic Commitment*) or complex (*Complex Commitment*). Complex commitments are composed of other commitments. The Appointment Pattern models a special type of commitment, named *Appointment* in UFO-C. An appointment is a commitment whose propositional content explicitly refers to a time interval.

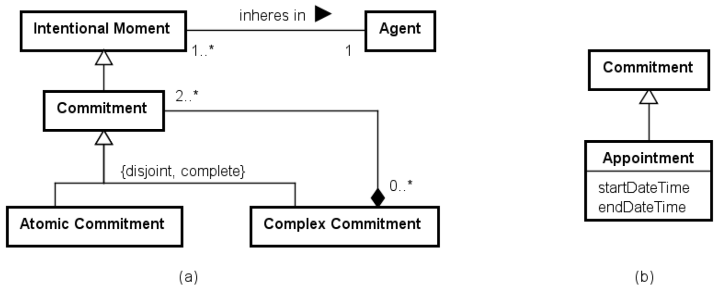

Fig. 5. (a) Atomic/Complex Commitment Types FOP; (b) Appointment FOP

In Figure 6, we apply the three aforementioned patterns when developing a domain ontology on the domain of Agendas. The Agenda Ontology was developed to support the semantic integration of the Google Calendar API and the Google Contacts API to a Software Engineering Environment. In an agenda we are mainly interested in registering contacts and scheduled appointments. Contacts can be either organizations or people, which have different principles of identity. For addressing the modeling problem regarding contacts, we reuse the FOP presented in Figure 4; for addressing the modeling problem regarding appointments, we reuse the FOPs presented in Figure 5. The Agenda Ontology fragment produced applying these three FOPs is shown in Figure 6. In this model, the fragment that was solved by reusing the FOP for roles with multiple and disjunctive kinds (Figure 4) is shown detached in grey. As one can observe, the structure of the FOP is exactly applied to solve the problem in the Agenda Ontology.

Regarding the reuse of the Atomic/Complex Commitment Types FOP and the Appointment FOP, the structure of the models, although not exactly the same, is analogous. An *Appointment* in the Agenda Ontology is an *Appointment* in the sense of UFO-C. Like *Commitments*, *Appointments* can be classified as *Atomic* and *Complex Appointments* (see Atomic/Complex Commitment Types FOP). In the case of the Agenda Ontology, all complex appointments (said *Multiple Appointment*) are composed of atomic appointments (said *Single Appointment*). *Multiple Appointments* are further categorized according to their frequency of occurrence into: *Diary Appointment*, *Weekly Appointment* and *Monthly Appointment*. Since in the agenda domain we are not interested in commitments that are not appointments, we did not represent the concept of commitment. However, since appointment is a special type of commitment, appointments inhere in an agent (*Agenda Owner*).

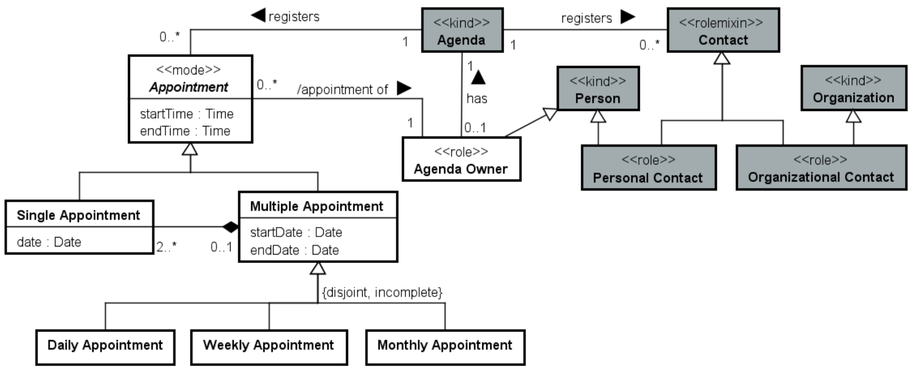

Fig. 6. A fragment of an Agenda Ontology.

Concerning the reuse of DROPs, which occur by extension, Figure 7 shows a fragment of the Reference Ontology on Software Testing (ROoST) [30], which were developed reusing DROPs organized as an ontology pattern language [20]. This picture shows a small (and simplified) fragment of ROoST dealing with artifacts produced and used by three software testing activities. This fragment of ROoST was built by composing and extending two DROPs: Work Product Participation (WPPA) pattern and Work Product Types (WPT) pattern. In Figure 7 the concepts from these patterns are shown detached in grey. As shown in this model fragment, concepts and relations from the DROPs (concepts: *Activity Occurrence*, *Document* and *Code*; relations: *uses* and *produces*) are extended for the testing domain. *Test Execution*, *Test Coding* and *Test Case Design* are subtypes of *Activity Occurrence*. *Test Case* and *Test Result* are subtypes of *Document* (sub-kinds in UFO). *Test Code* is a sub-kind of *Code*, while *Code To Be Tested* is a role played by a *Code* when used in a *Test Execution*. Regarding relation specializations, *Test Case Design produces Test Case*. *Test Coding uses Test Case* and *produces Test Code*. Finally, *Test Execution uses Test Code* and *Code To Be Tested*, and *produces Test Results*.

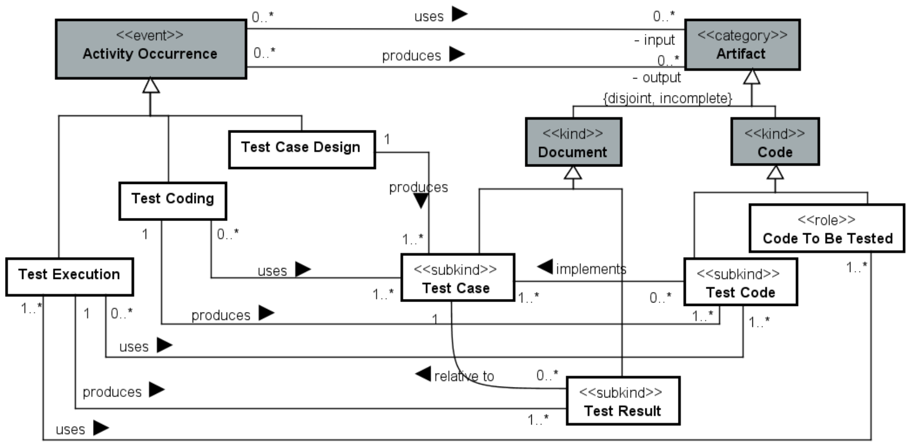

Fig. 7. A Fragment of the Reference Ontology on Software Testing

It is worthwhile to point out that new concepts and relations can also be added to the domain ontology being developed. In the case of ROoST model fragment shown in Figure 7, two relations were added: *Test Code implements Test Case*, and *Test Result is relative to Test Case*.

5 Final Considerations

Ontology patterns (OPs) are currently recognized as a beneficial approach for ontology development [3, 4]. In this paper we made an analogy between patterns in Software Engineering and patterns in Ontology Engineering, in order to clarify and harmonize the terminology used in both areas. Since patterns in Software Engineering have already been studied and used longer than in Ontology Engineering, we revisited some notions in the latter. In particular, by revisiting the classification proposed by Gangemi and Presutti in [3], we propose another way of classifying OPs, which is strongly related to the ontology development phase in which they are to be used. Moreover, we discuss ways of reusing OPs, namely by analogy and by extension. Domain-related OPs are typically reused by extension; while the other types of OPs are typically reused by analogy. It is worthwhile to point out that a third way is complementary to both: patterns composition. By means of examples, we illustrated how different ontology conceptual patterns can be reused for developing domain reference ontologies.

Acknowledgments. This research is funded by the Brazilian Research Funding Agencies FAPES (PRONEX # 52272362/11) and CNPq (#311578/2011-0).

References

1. Noppens, O., Liebig, T., Ontology Patterns and Beyond - Towards a Universal Pattern Language. In: Proceedings of the Workshop on Ontology Patterns (WOP 2009), Washington D.C., USA, 2009.
2. Poveda-Villalón, M., Suárez-Figueroa, M.C., Gómez-Pérez, A., Reusing Ontology Design Patterns in a Context Ontology Network. In: Second Workshop on Ontology Patterns (WOP 2010), Shangai, China, 2010.
3. Gangemi, A., Presutti, V., Ontology Design Patterns. In: Handbook on Ontologies, Second edition, Staab, S., Studer, R. (Eds.), Springer, 2009, pp. 221 - 243.
4. Blomqvist, E., Gangemi, A., Presutti, V. *Experiments on Pattern-based Ontology Design*. In Proceedings of K-CAP 2009, pp. 41-48. 2009.
5. Presutti, V., Daga, E., Gangemi, A., Blomqvist, E. *eXtreme Design with Content Ontology Design Patterns*. In: Proceedings of the Workshop on Ontology Patterns (WOP 2009), Washington D.C., USA, 2009.
6. Gamma, E., Helm, R., Johnson, R.E., Vlissides,J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
7. Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional Computing Series, 1997.
8. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns,* John Wiley & Sons, 1996.
9. Clark, P., Thompson, J., Porter, B. Knowledge patterns. In: Proceedings of the 7th International Conference on *Principles of Knowledge Representation and Reasoning (KR 2000)*, pp. 591–600, San Francisco, 2000.
10. Reich, J.R. Ontological design patterns: Metadata of molecular biological ontologies, information and knowledge. Proceedings of the 11th International Conference on Database and Expert Systems Applications, DEXA 2000, pp. 698–709, London, 2000.
11. Svatek, V., Design Patterns for Semantic Web Ontologies: Motivation and Discussion. In: Proceedings of the 7th Conference on Business Information Systems, Poznan, Poland, 2004.
12. Buschmann, F., Henney, K., Schmidt, D.C., *Pattern-Oriented Software Architecture: On Patterns and Pattern Language*s, John Wiley & Sons Ltd, 2007.
13. Schmidt, D. C., Fayad, M., Jonhson, R.E., Software Patterns, Communications of the ACM, vol. 39, no. 10, October 1996.
14. Hamza, H., Mahdy, A., Fayad, M.E., Cline, M. Extracting domain-specific and domain-independent patterns. In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '03), New York, USA, pp. 310-311, 2003.
15. Devedzic, V. Software Patterns, In: Handbook of Software Engineering and Knowledge Engineering Vol.2 -Emerging Technologies, S.K. Chang (Ed.), World Scientific Pub Co Inc., 2002.

16. Fowler, M., *Patterns of Enterprise Application Architecture*, Addison Wesley, 2003.
17. Coplien, J.O., Advanced C++ Programming Styles and Idioms, Addison-Wesley, 1992.
18. Guarino, N.: Formal Ontology and Information Systems. In: Guarino, N. (ed.) Formal Ontology and Information Systems, pp. 3–15, IOS Press, Amsterdam, 2008.
19. Gangemi, A. Ontology Design Patterns for Semantic Web Content, In: Proc. of the 4th International Semantic Web Conference – ISWC'2005, p. 262 – 272, Galway, Ireland, 2005.
20. Falbo, R. A., Barcellos, M.P., Nardi, J.C., Guizzardi, G. Organizing Ontology Design Patterns as Ontology Pattern Languages, 10th Extended Semantic Web Conference, Montpellier, France, 2013.
21. Ontology Design Patterns (ODPs) Public Catalog. http://www.gong.manchester.ac.uk/odp/html/. Last access in 2nd September 2013.
22. Aranguren, M.E., Antezana, E., Kuiper, M., Stevens, R. Ontology Design Patterns for bio-ontologies: a case study on the Cell Cycle Ontology. BMC bioinformatics, 9(Suppl 5):S1, 2008.
23. Guizzardi, G., On Ontology, ontologies, Conceptualizations, Modeling Languages and (Meta)Models, In O. Vasilecas, J. Edler, A. Caplinskas (Org.). *Frontiers in Artificial Intelligence and Applications, Databases and Information Systems IV*. IOS Press, Amsterdam, 2007.
24. Guizzardi, G., Zamborlini, V., A Common Foundational Theory for Bridging two levels in Ontology-Driven Conceptual Modeling. In: 5th International Conference of Software Language Engineering (SLE 2012), Dresden. Germany, 2012.
25. Angele, J., Kifer, M., Lausen, G., Ontologies in F-Logic, In: Handbook on Ontologies, Second edition, Staab, S., Studer, R. (Eds.), Springer, 2009, pp. 45 – 70.
26. Silbernagl, D., Reasoning with Ontologies in Databases: including optimization strategies, evaluation and example SQL code, VDM Verlag, 2011.
27. Guizzardi, G. *Ontological Foundations for Structural Conceptual Models*, Netherlands: Universal Press, 2005.
28. Guizzardi, G., Falbo, R.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, Recife, Brazil, pp. 244-251, 2008.
29. Rittgen, P., Translating Metaphors into Design Patterns, In: Advances in Information Systems Development, Springer, pp. 425 – 436, 2006.
30. Souza, E.F., Falbo, R.A., Vijaykumar, N.L., Using Ontology Patterns for Building a Reference Sofware Testing Ontology, The 8th International Workshop on Vocabularies, Ontologies and Rules for the Enterprise (VORTE 2013), Vancouver, Canada, 2013.