Finding the Pattern You Need: The Design Pattern Intent Ontology

Holger Kampffmeyer and Steffen Zschaler

Technische Universität Dresden, Germany
Holger.Kampffmeyer@googlemail.com,
Steffen.Zschaler@tu-dresden.de

Abstract. Since the seminal book by the Gang of Four, design patterns have proven an important tool in software development. Over time, more and more patterns have been discovered and developed. The sheer amount of patterns available makes it hard to find patterns useful for solving a specific design problem. Hence, tools supporting searching and finding design patterns appropriate to a certain problem are required. To develop such tooling, design patterns must be described formally such that they can be queryed by the problem to be solved. Current approaches to formalising design patterns focus on the solution structure of the pattern rather than on the problems solved. In this paper, we present a formalisation of the intent of the 23 patterns from the Gang-of-Four book. Based on this formalisation we have developed a Design Pattern Wizard that proposes applicable design patterns based on a description of a design problem.

1 Introduction

Since Gamma, Helm, Johnson, and Vlissides (the so-called Gang of Four (GoF)) published their seminal book [14], design patterns have proven a useful tool in software development. A design pattern encapsulates a solution for a recurring design problem in template form, ready to be applied to new instances of the problem. It, thus, is a form of encoding and transferring design knowledge between projects and developers.

Because design patterns are so useful, lots of them have been discovered or developed and documented since the publication of the GoF book. Current design patterns have appeared in specific application domains (J2EE patterns [5,6,11], User-Interface patterns [20]), as language-dependent patterns (also called idioms), as patterns at different abstraction levels (analysis patterns [10], architectural patterns [11,12]), or simply as large collections of design patterns in pattern catalogues [7,25]. Even though the GoF book only contains 23 design patterns, the authors state that "it might be hard to find the one (design pattern) that addresses a particular design problem especially if the catalogue is new and unfamiliar to you." [14]. The sheer number of design patterns available today impedes effective reuse of design patterns, because it is very difficult to find the

G. Engels et al. (Eds.): MoDELS 2007, LNCS 4735, pp. 211–225, 2007.

right design pattern for a given design problem. This is especially true for inexperienced developers who do not yet know a large number of design patterns by heart. To deal with the large number of patterns effectively, software developers require tool support for finding design patterns that can solve a certain design problem. This paper is a step towards such tooling.

For this, we require a description of design patterns to be available in a machine-readable format. This description must contain a formal specification of the design patterns. It must be constructed in such a way as to allow querying based on the design problem to be solved.

Existing approaches to formalising design patterns generally cover only the formal description of the *solution structure* of design patterns. While the *structure* of a design pattern explains *how* it is applied in software design, it does not explain *when* to apply a design pattern for a given design problem. Only the *intent* section of a design pattern description explains the purpose of a design pattern. To the best of our knowledge, no work exists trying to formalise the intent of design patterns. However, software tools based on such a formalisation could enable users to query for a design pattern by giving a description of their design problem based on terminology defined in the specification. Ontologies are one way of expressing such a formalisation, because they directly support the creation and querying of such knowledge bases.

The main contribution of this paper is, therefore, a Design Pattern Intent Ontology (DPIO); that is, an extensible knowledge base of design patterns (in our case the 23 GoF patterns) classified by their *intent*.

The remainder of this paper is structured as follows: The following two sections give a short introduction to design patterns and to ontologies. Section 4, the main section of the paper, presents the DPIO. In Sect. 5 the ontology is evaluated by checking that certain competency questions (sample queries) can be formalised and answered based on the ontology. Section 6 discusses the design pattern wizard developed on top of the DPIO. The paper closes with a discussion of related work (Sect. 7) and a conclusion (Sect. 8).

2 Design Patterns

A design pattern is "*a solution to a problem in a context*" [13]. It is a way to achieve reusability in software design. Design patterns first emerged in the context of architecture and town building [4]. However, the idea of reusing design by applying patterns to recurring design problems has been ported to object-oriented software design in the GoF book *Design Patterns: Elements of Reusable Object-Oriented Software* [14].

In the GoF book, design pattern descriptions are structured into the following parts: *pattern name and classification, intent, motivation (forces), applicability, structure, participants, collaboration, consequences, implementation, sample code, known uses,* and *related patterns.* Formalisations of design patterns typically focus on the structure of the solution proposed in the pattern (for example, [17]). This does not, however, uniquely characterise a design pattern. Consider, for

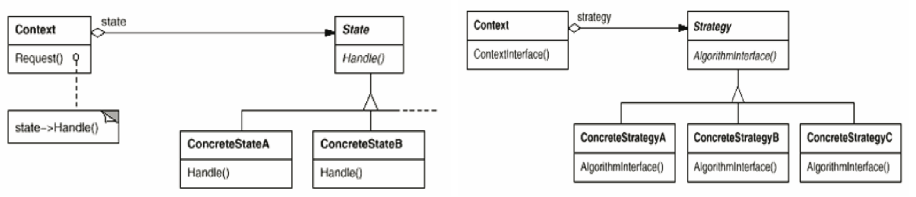


Fig. 1. The structure of the (a) *State* and (b) *Strategy* patterns. Copied from [14].

example, the patterns *State* and *Strategy* (cf. Fig. **??**). Their structure is more or less identical. However, their intent is not. The intent of *State* is given in [14] as

> "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class." [14]

In contrast, the intent of *Strategy* is

> "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." [14]

The intent of a design pattern is the first section a developer reads when trying to understand whether a design pattern is a solution to the developer's current problem. Hence, this is the section that should form the basis of a formalisation of design patterns that can help developers find the pattern they need.

3 Ontologies

"An ontology is an explicit specification of a conceptualisation" [16]. Ontologies were developed by the Artificial Intelligence community to support the sharing and common understanding of domain knowledge. Every ontology consists of a hierarchy of `classes`, `properties` (attached to the classes and used to model relationships between them), and `individuals` (instances of classes).

Ontologies are suitable means for formalising the intent of design patterns, because they allow to encode domain knowledge in a simplified abstract way and enable queries to be evaluated against a knowledge base defined by an ontology. For this reason, in this paper, we present an ontology-based formalisation of the intent of design patterns, thus defining a machine-readable, queryable catalogue of design patterns. We use OWL, the Web Ontology Language [26], as the formalisation language. We have chosen OWL, because it is a W3C recommendation (that is, an accepted standard) and because its good tool- and framework-support (see [2,3]) allows easy extension of the ontology and development of tools using the ontology as a knowledge base.

4 The Design Pattern Intent Ontology

The aim of the Design Pattern Intent Ontology (DPIO) is to support developers in choosing a design pattern for a given design problem. That is, the domain

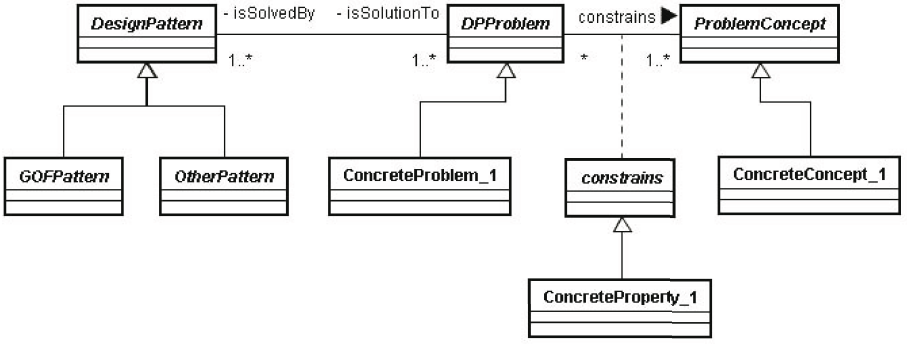


Fig. 2. The parent classes of the three hierarchies *Design Pattern*, *Design Problem* and *Problem Concept*

of the ontology is the area of software development. In this paper, we constrain the scope of the ontology to cover only the design patterns defined in the GoF book.[1] Thus, the ontology should provide the terms and concepts the GoF book uses to describe software design and design patterns.

The scope of the ontology is restricted to the intent and the application of design patterns in software design. The ontology must be elaborate enough to enable the querying for solutions to design problems. However, it is not intended to describe the *structure* of a design pattern. There is other work that is formalising these aspects of design patterns (see Sect. 7). For the scope of this work, the formalisation of the structure of design patterns does not give any additional benefits.

Competency questions are a way of determining the scope of an ontology [24]. They are the kind of questions the ontology should be able to answer. Here are some possible competency questions for the DPIO:

- Which design patterns are contained in the ontology?
- Which concepts are contained in the ontology that can be used to model a design problem?
- Which design pattern is a solution to the problem of varying an algorithm?
- Which design pattern is a solution to the problem of objectifying state?

In Sect. 5 we show how a formalised representation of the competency questions can be used to evaluate the ontology.

In designing the structure of the DPIO, we need to take into account the possible relations between design patterns and the problems they solve: One design pattern can be the solution to more than one design problem. But one design problem can also be solved by more than one design pattern. For example, both the *Prototype* pattern and the *Builder* pattern are concerned with object creation. Consequently, there exists an n:m relationship between design patterns and design problems. The solution to model this n:m relationship is

[1] This is for reasons of associated effort only. The basic structure of the DPIO has been designed to be extensible to arbitrary design patterns.

```
1  Class: StrategyDesignPattern
2
3  SubClassOf: GOFPattern
4
5     and isSolutionTo some
6         AlgorithmDecoupling
7     and isSolutionTo some
8         AlgorithmSelection
9     and isSolutionTo some
10        AlgorithmVariation
11
12 ...
```

Listing 1.1. Definition of the design pattern class `StrategyDesignPattern`

by defining a set of *design problems* and relating these design problems to design patterns. Therefore, a design pattern is a *solution* to one or more design problems. We furthermore describe a design problem by using *problem concept* terms. A design problem is *constrained* by problem concepts. Figure **??** gives a graphical overview of the core structure of the DPIO. The relations between `DesignPattern`, `DesignProblem` and `ProblemConcept` classes are depicted using UML-notations.[2] UML classes symbolise OWL classes, UML associations symbolise OWL object properties. The association between `DesignPattern` and `DesignProblem` indicates an object property `isSolutionTo` that relates `DesignPattern` classes to `DesignProblem` classes. The property `isSolvedBy` is an inverse property of `isSolutionTo`. The association class `constrains` indicates an OWL object property that can be further specialised by subproperties. Both `DesignPattern`, `DesignProblem` and `ProblemConcept` classes are the root classes of subclass hierarchies specialising the root concepts.

To discuss the structure of the ontology in more detail, we look at some example definitions. Listing 1.1 is an example of a subclass from the `DesignPattern` hierarchy and shows the definition of the class `StrategyDesignPattern` in Manchester OWL syntax [18]. It states that an individual is a `StrategyDesignPattern` if it is a subclass of `GOFPattern` (Line 3) and is a solution to certain problems as per the intent of the *Strategy* design pattern: "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. [...] Strategies provide a way to configure a class with one of many behaviors [they can be used when] we need different variants of an algorithm" [14]. The *design problems* formalising these aspects are `AlgorithmDecoupling` (Line 6), `AlgorithmSelection` (Line 8), and `AlgorithmVariation` (Line 10). Together they define a set of design problem facets with the need to decouple, select and vary algorithms.

[2] There seems to be no commonly agreed visual representation of ontologies yet. We have chosen UML class diagrams because they are easy to understand.

Consequently, the *Strategy* design pattern is a solution to these design problems and connected to them via the object property `isSolutionTo`.

```
1 Class: AlgorithmDecoupling
2
3 SubClassOf: DecouplingProblem
4
5 and decouples some
6       Algorithm
```

Listing 1.2. Definition of the design problem class `AlgorithmDecoupling`

Listing 1.2 is an example of a subclass from the `DPProblem` hierarchy and shows the definition of the class `AlgorithmDecoupling`. For an individual to be of class `AlgorithmDecoupling` it is necessary to be a member of the anonymous class of things that is linked to at least one member of class `Algorithm` via the object property `decouples` (Line 5–6). The property `decouples` is thereby an example of a subproperty of the `constrains` hierarchy. An `AlgorithmDecoupling` design problem is therefore simply concerned with the decoupling of algorithms.

The examples discussed so far (Listing 1.1 and 1.2) have been modelled using vocabulary defined in the DPIO. Tables 1 and 2 show an excerpt of the terms and concepts we have defined in the DPIO[3]. Table 1 shows the design problem hierarchy. An abstract concept `DPProblem` is defined that is the root node for more specific problems. The inheritance relationship is represented by a ⌞ symbol and an indentation. The description column gives a short comment on the intent of each problem concept. For example, a `Problem` "is the abstract base class and the root node of the problem hierarchy". A `DPProblem` is a `Problem` that can be solved by design pattern solutions. Further specialisations are `ControlProblem`, `AlgorithmSelection`, `DecouplingProblem` etc. The top level hierarchy is based on the Tichy catalogue [28], an initial classification of design-pattern intents. However, we extend and restructure Tichy's hierarchy to allow a more detailed modelling of the problem domain.

Similar to classes, OWL properties also can be specialised by sub-properties. The OWL object property `constrains`, modelled as a UML association class, is the root property of more specialised object properties. Table 2 lists an excerpt of modelled OWL object properties which are used to describe design problems.

The aforementioned framework structure of `DesignPattern`, `DesignProblem`, `constrains`-properties and `ProblemConcept` is what allows developers to formulate queries without knowing the design patterns in the knowledge base. To do so, developers use the terminology provided in the `ProblemConcept` hierarchy and the `constrains`-properties to model their design problem. Standard ontology reasoning can then be used to determine the design patterns solving such

[3] The whole ontology can be downloaded from `http://www.holger-kampffmeyer.de/DesignpatternsIntentOntology.owl`.

Table 1. The design problem taxonomy

Design Problem	Description
Problem	Abstract base class and root of the hierarchy.
⌞ DPProblem	A problem from the domain of design patterns.
⌞ ControlProblem	controls execution and method selection.
⌞ ...	...
⌞ AlgorithmSelection	controls algorithm selection.
⌞ ...	...
⌞ DecouplingProblem	divides a software system into independent parts. The parts can be built, changed, replaced.
⌞ AlgorithmDecoupling	decouples algorithms from the rest of the system.
⌞ ...	...
⌞ VariantManagementProblem	treats different objects uniformly.
⌞ ...	...
⌞ AlgorithmVariation	varies an algorithm.
⌞ ...	...

```
(retrieve (?solution )
   (and (?solution |GOFPattern|)
   (?solution ?designproblem |isSolutionTo|)
   (?designproblem ?problemConstraint |<objectproperty>|)
   (?problemConstraint |<someclass>|)
))
```

Listing 1.3. Template-structure of a nRQL query to retrieve all design patterns that are a solution to some design problem

problems. The structure of a query asking for those design patterns solving a specific problem can look like Listing 1.3, showing the structure of a query in nRQL syntax [19]. In nRQL, a query is composed of a preliminary *command*, followed by the query *head* and *body* put in parentheses, respectively. The head of the query contains variables that are bound to the result set of the query. The body contains constraints of a query, similar to those of SQL where-clauses. It consists of one or more query atoms. Query atoms can be combined to complex queries by using logical operators such as `AND`, `OR`, and `NOT`. The query in Listing 1.3 retrieves all design patterns being a solution to a specified design problem. The placeholders `<objectproperty>` and `<someclass>` need to be replaced by concrete subproperties of `constrains` and subclasses of `ProblemConcept`, respectively. For example, `<objectproperty>` could be replaced by `distributes` and `<someclass>` could be replaced by `Behavior` to retrieve all *behavioral* design patterns. The Design Pattern Wizard we introduce in Section 6 generates queries with a similar structure to the query shown in Listing 1.3.

Table 2. The OWL object properties used to model design problems

Object Property	Description
constrains	binds a problem concept to a design problem, super property for all other properties.
∟ varies	allows diversity.
∟ ...	...
∟ controls	handles something.
∟ ...	...
∟ handles	equivalent property to "controls".
∟ ...	...
∟ selects	chooses some behavior.
∟ ...	...
∟ decouples	loosens the coupling of objects.
∟ ...	...

```
1 (concept-descendants |DesignPattern|)
```

Listing 1.4. nRQL query to retrieve all design patterns defined in the ontology

5 Evaluation

There are different possibilities for evaluating an ontology such as the DPIO. The final proof of the concepts can, of course, only be found through a controlled experiment in which developers are asked to use the ontology to solve certain design problems. Such an experiment would show if the ontology achieves our underlying goal of providing a formalisation of design patterns more appropriate to the problem of finding the design pattern one needs to solve a specific problem in software design. Performing such experiments is costly and time consuming. For this reason, we have not done so yet. However, we acknowledge the importance of such work and propose to perform it in future research.

A different approach to evaluating our ontology is to check whether it can answer the kinds of questions that are likely to be asked of it. To this end, we have developed a catalogue of *competency questions*. Here, we translate the competency questions into a formal version we can use to query the DPIO. In a second step, we test the ontology by verifying that the result set of the queries correspond to the intended meaning of the ontology. We use the nRQL query language [19] for formalising the competency questions.

Which design patterns are contained in the ontology? Listing 1.4 shows the formal representation of this competency question. It retrieves all children of the concept `DesignPattern`. The result set of the query contains the 23 GoF patterns modelled in the ontology.

Which concepts are contained in the ontology that can be used to model a design problem? Listing 1.5 shows the formal representation of this competency

```
(concept-descendants |ProblemConcept|)
```

Listing 1.5. nRQL query to retrieve all problem concepts defined in the ontology

```
(retrieve (?x )
   (and (?x |GOFPattern|)
   (?x ?p |isSolutionTo|)
   (?p ?a |varies|)
   (?a |Algorithm|)
))
```

Listing 1.6. nRQL query to retrieve all design patterns that are a solution to varying an algorithm

question. Concepts that are intended to model a design problem are subclasses of the class `ProblemConcept`. Consequently, the nRQL query retrieves all children of `ProblemConcept`.

Which design pattern is a solution to the problem of varying an algorithm? Listing 1.6 shows the formal representation of this competency question. The query asks for those subclasses of `GOFPattern` that are linked to a design problem via the object property `isSolutionTo`. It furthermore asks for only those design problems that have an object property `varies` that relates the concept `Algorithm`. The query results in the Template Method and the Strategy design patterns.

Which design pattern is a solution to the problem of objectifying state? Listing 1.7 shows the formal representation of this competency question. The query asks for those subclasses of `GOFPattern` that are linked to a design problem via the object property `isSolutionTo`. It furthermore asks for only those design problems that have an object property `objectifies` that relates the concept `State`. The result set for this query contains the Memento and the State design patterns.

```
   (and (?x |GOFPattern|)
   (?x ?p |isSolutionTo|)
   (?p ?a |objectifies|)
   (?a |State|)
))
```

Listing 1.7. nRQL query to retrieve all design patterns that are a solution to objectifying state

6 The Design Pattern Wizard

As we have outlined in the introduction, an ontology can be used in tools as a knowledge base. The Design Pattern Intent Ontology contains the vocabulary for describing the intent of design patterns. In order to extract knowledge from the ontology, the user has to execute queries on it. However, the construction of these queries can be quite complicated. This reduces the usability of ontologies to domain experts only. The Design Pattern Wizard serves as a front-end for generating well-defined queries. It allows design problems to be described visually and suggests a set of matching design patterns for a given design problem. It, furthermore, provides inexperienced users with vocabulary they can use to define design problems. The Design Pattern Wizard can be obtained from the first author at email request.

The Design Pattern Wizard is a prototype and proof of concept. It shows the applicability of ontologies for tool support in the area of software design. The Design Pattern Wizard has been implemented as an Eclipse RCP (Rich Client Platform) application [1]. The RCP architecture allows the developer to configure an application to either be integrated as a plug-in into the Eclipse platform or to be deployed as a stand-alone application.

Figure 3 shows the problem description window when the Design Pattern Wizard is started. The main part of the dialogue is filled by the problem description table. It consists of a predicate constraint column and an object (concept) constraint column. Each row represents a statement constraining the design problem a design pattern should solve. The first column consists of check boxes used to select rows for editing or deletion. Below the constraint table resides a button for adding constraint rows to the table and a button for deleting a selected row. In the bottom right corner of the dialogue the button for retrieving a design pattern suggestion based on the design problem description is located.

Clicking on the *predicate constraint* in one of the rows allows to open the *Predicate Constraint Dialog* shown in Fig. 4 a). A tree representation of all *predicates* modelled in the ontology is presented. The user can choose a predicate and the

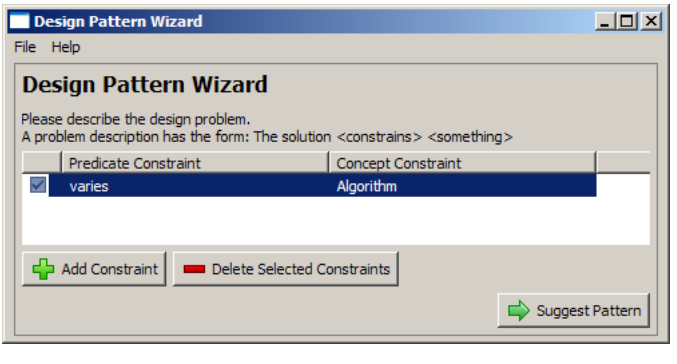
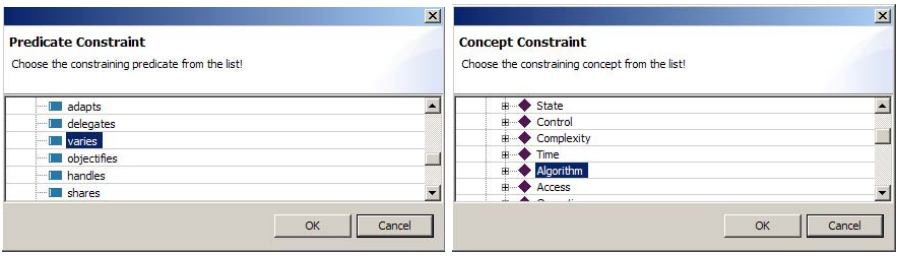


Fig. 3. Screenshot of the Design Pattern Wizard



(a) constraining problem predicates (b) constraining problem concepts

Fig. 4. Screenshot of the dialogues constraining problem predicates and problem concepts

chosen predicate value will be set in the constraint row of the wizard. A similar dialogue is opened when the user wants to constrain the *concept constraint* part of a problem constraint. The *Concept Constraint Dialog* (Fig. 4 b)) allows the user to select a concept that further constrains a predicate. The selected concept is then set in the constraint cell.

Figure 5 shows the *Result Dialog* that is opened when the user hits the *Suggest Design Pattern* button. It presents all suitable design patterns matching the modelled design problem in a simple table. The first column shows the name of the design pattern while the second shows a description of the pattern.

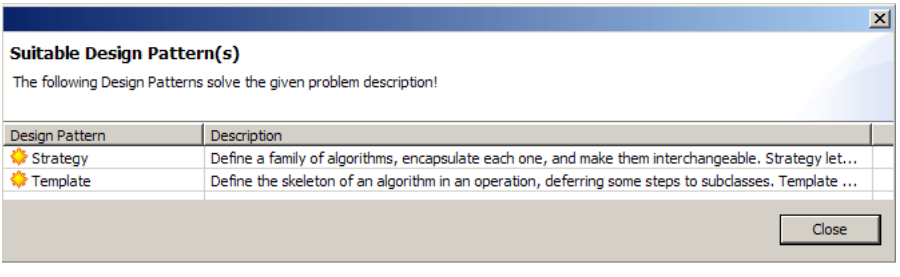


Fig. 5. Screenshot of the result window suggesting suitable design patterns

7 Related Work and Discussion

Our survey of current work in design pattern formalisation indicates that no other work before has tried to classify design patterns according to their intent by using ontologies.

The formalisation of design problems in this paper is based on the work of Tichy [28], who developed a catalogue (but no formal ontology) listing over 100 design patterns. In his classification, he concentrates on the problems patterns solve. Among Tichy's design problem categories are `decoupling` which refers to dividing a software system into independent parts such that the parts can be

built, changed, replaced, and reused independently; `variant management` whose patterns treat different but related objects uniformly by factoring out their commonalities; `state handling` whose patterns allow the generic manipulation of object state; `control` whose patterns are used to control execution and method selection and others. We have both formalised and refined Tichy's classification and, thus, made it available to mechanical treatment and to computer-aided querying by software developers.

Henninger et al. [17] are using an ontology-based metamodel to formally describe software patterns. The goal is to develop intelligent tools that provide a computational basis utilising software patterns. As a use case they mention usability patterns. They base their core metamodel on properties such as `hasProblem`, `hasSolution`, `hasContext`, `hasRationale`, `hasForces`, properties that are developed from the original structure of the pattern description of [4]. They extend the core metamodel with properties that describe the relationships between patterns, such as `uses`, `requires`, `alternative`, and `conflictsWith`. The ontology they have developed does not so much concentrate on the support of selecting a suitable pattern for a given problem, but rather on the relationships between patterns. Furthermore, Henninger et al. concentrate on the domain of usability patterns of web sites, but not on GoF patterns or design patterns in general. The main goal of their work is to define a shareable vocabulary in the domain of usability patterns. Furthermore, they do not describe how to query their ontology, nor do they describe how to model design problems in an ontology. A very similar approach to [17] is [23]. Here, ontologies are used to formalise hypermedia and web design patterns. The vocabulary is almost identical to [17] but includes additional concepts such as `PatternComponent`, `Category`, `Problem` and `Solution`. The scope of their work is the support of and an integration into hypermedia design tools.

Pereira de Medeiros et al. [8] have developed the Kuaba Ontololgy, an ontology and design vocabulary to describe the design rationale of software design. The goal is to make explicit the decisions and justifications that have led to a design. The formal description of the design rationale enables reuse at a high abstraction level. Important reasoning elements of the ontology are `Question`, `Idea` and `Argument`. The vocabulary defined in the Kuaba Ontology helps in the decision-making process of software design. Their work is not particularly focused on design patterns, but more on software design in general.

The work of Dietrich et al. [9] uses OWL to formally describe design patterns. However, only the structure of design patterns is considered, not the intent or applicability. They use their ontology to detect patterns in software artefacts, not to help in the selection process of a design pattern. Other approaches that use formal languages to describe the structure of design patterns include [21] who developed a design pattern modelling language DPML, and [22,27]. What is common to all this work is the sole concentration on the structural aspects of design patterns and the omission of the intent of design patterns.

8 Conclusion and Further Work

In this paper, we have presented a novel approach to formalising design patterns, based on their intent. We have proposed the Design Pattern Intent Ontology providing terminology for formulating intents and classifying the 23 GoF design patterns by their intent. To the best of our knowledge, this is the first formalisation of design patterns based on their intent. This ontology forms the basis for the Design Pattern Wizard, a tool supporting software developers in finding the right design pattern(s) for a given design problem.

The work presented in this paper enables software developers to efficiently find design patterns applicable for their design problems. It is, thus, a measure countering the effect of the ever increasing number of design patterns available in various design pattern catalogues. The hierarchical structure of the ontology allows developers to provide incomplete descriptions of their design problems and still receive valuable responses. A developer with only a rough picture of her design problem can simply choose predicates from the top parts of the hierarchies. Such a query results in the retrieval of all design patterns matching this rough problem description. Iteratively, the developer can describe her design problem more precisely, based on the results of the former queries. The basic structure of `DesignPattern, DesignProblem,` and `ProblemConcept` is valid for all design patterns. Therefore, the ontology can easily be extended to cover other design patterns beyond the GoF book. To this end, it may be necessary to add new vocabulary to the `DesignProblem` and `ProblemConcept` hierarchies. Testing the ontology with other catalogues of design patterns remains for future work.

So far, the Design Pattern Wizard is a stand-alone application. In the future, this should be integrated with CASE tools to allow direct integration of a pattern found into a design under development. Developers could select a design and add a design pattern to it, using the Design Pattern Wizard to select the pattern. For this to work, we need to study ways of selecting and manipulating the parts of a model where a design pattern should be added, in addition to the formalisation of design-pattern intent.

Another interesting question is how we can help developers understand the problem they are trying to solve. We believe, developers often know that there are flaws in their design, but cannot immediately understand the source of the problem. Thus, they will experiment with different design choices, which with good designers will eventually lead to a better understanding of the problem, and, thence, to a better design. An interesting question is if this problem-finding process can be supported by CASE tools observing the different experiments of a designer and using the DPIO to suggest design patterns that might be helpful.[4]

Additionally, the basic structure of the DPIO should also be applicable to patterns that are not strictly design patterns. For example, [15] have proposed an approach to graphically organise, analyse and refine non-functional requirements for the structuring, understanding, and applying of design patterns during design. We plan to study how the DPIO can be applied in this context.

[4] Thanks to Mirko Seifert for this suggestion.

References

1. Eclipse – Rich Client Platform (2006), http://www.eclipse.org/home/categories/rcp.php
2. Jena – a semantic web framework for Java (2006), http://jena.sourceforge.net/
3. Protégé ontology editor and knowledge acquisition system (2006), http://protege.stanford.edu
4. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language. Center for Environmental Structure Series, vol. 2. Oxford University Press, New York, NY (1977)
5. Alur, D., Crupi, J., Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies. Pearson Education (2001), Patterns catalog available at http://java.sun.com/blueprints/corej2eepatterns/index.html
6. Alur, D., Crupi, J., Malks, D.: Sun Java center – J2EE patterns (March 2001), http://java.sun.com/developer/technicalArticles/J2EE/patterns/
7. PatternShare Community. Patternshare community (2006), http://patternshare.org/
8. de Medeiros, A.P., Schwabe, D., Feijó, B.: A design rationale representation for model-based designs in software engineering. In: Belo, O., Eder, J., e Cunha, J.F., Pastor, O. (eds.) CAiSE Short Paper Proceedings, CEUR Workshop Proceedings. CEUR-WS.org. vol. 161 (2005), http://www.ceurws.org/Vol-161/FORUM_27.pdf
9. Dietrich, J., Elgar, C.: A formal description of design patterns using OWL. In: Australian Software Engineering Conference (ASWEC'05), pp. 243–250. IEEE Computer Society, Los Alamitos (2005), http://doi.ieeecomputersociety.org/10.1109/ASWEC.2005.6
10. Fowler, M.: Analysis Patterns: Reusable Object Models. The Addison-Wesley Object Technology Series. Addison-Wesley Professional, Reading (1996)
11. Fowler, M.: Patterns of Enterprise Application Architecture. The Addison-Wesley Signature Series. Addison Wesley, Reading (2003)
12. Fowler, M.: Patterns in enterprise software (2005), http://www.martinfowler.com/articles/enterprisePatterns.html
13. Fowler, M.: Writing software patterns (August 2006), http://martinfowler.com/articles/writingPatterns.html
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY (1995)
15. Gross, D., Yu, E.S.K.: From non-functional requirements to design through patterns. Requirements Engineering 6(1), 18–36 (2001), http://citeseer.ist.psu.edu/gross00from.html
16. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition 6(2), 199–221 (1993), http://portal.acm.org/citation.cfm?id=173747
17. Henninger, S., Padmapriya, A.: An Ontology-Based Metamodel for Software Patterns. In: Seke2006. 18th Int. Conf. on Software Engineering and Knowledge Engineering, San Francisco (July 2006) (to be presented), http://cse.unl.edu/~scotth/papers/SEKE06-TechReport.pdf
18. Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., Wang, H.H: The Manchester OWL syntax (2005), http://owl-workshop.man.ac.uk/acceptedLong/submission_9.pdf

19. Racer Systems GmbH & Co. KG. Racerpro users guide version 1.9 (2005), http://www.racer-systems.com/products/racerpro/manual.phtml
20. Mahemoff, M., Johnston, L.: Principles for a usability-oriented pattern language (1998), http://citeseer.ist.psu.edu/article/mahemoff98principles.html
21. Mapelsden, D., Hosking, J., Grundy, J.: Design pattern modelling and instantiation using DPML. In: CRPIT '02. Proceedings of the Fortieth International Conference on Tools Pacific, pp. 3–11. Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2002)
22. Mikkonen, T.: Formalizing design patterns. In: ICSE '98. Proceedings of the 20th international conference on Software engineering, pp. 115–124. IEEE Computer Society, Washington, DC, USA (1998)
23. Montero, S., Diaz, P., Aedo, I.: Formalization of web design patterns using ontologies (2005)
24. Noy, N.F., McGuinness, D.L.: Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Knowledge Systems Laboratory, Stanford University, Stanford, CA, 94305, USA (March 2001)
25. Portland Pattern Repository. Portland pattern repository (2006), http://c2.com/ppr/
26. Smith, M.K., Welty, C., McGuinness, D.L.: OWL web ontology language guide (2004), http://www.w3.org/TR/owl-guide
27. Taibi, T., Ngo, D.C.L.: Formal specification of design patterns - a balanced approach. Journal of Object Technology 2(4), 127–140 (2003), http://www.jot.fm/issues/issue_2003_07/article4.pdf
28. Tichy, W.F.: A catalogue of general-purpose software design patterns. In: TOOLS '97. Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems, IEEE Computer Society, Washington, DC, USA (1997)