

Towards a web of patterns

Jens Dietrich, Chris Elgar*

Massey University, Institute of Information Sciences and Technology, Palmerston North, New Zealand

Received 7 June 2006; accepted 10 November 2006

Available online 20 January 2007

Abstract

Design patterns have been used successfully in recent years in the software engineering community in order to share knowledge about the structural and behavioural properties of software. There is a growing body of research in the area of design pattern detection and design recovery, requiring a formal description of patterns which can be matched by tools against the software that is to be analysed.

We propose a novel approach to the formal definition of design patterns that is based on the idea that design patterns are knowledge that is shared across a community and that is by nature distributed and inconsistent. By using the web ontology language (OWL) we are able to formally define design patterns and some related concepts such as pattern participant, pattern refinement, and pattern instance. We discuss the respective ontology and give examples of how patterns can be defined using this ontology. We present the prototype of a Java client that accesses the pattern definitions and detects patterns in Java software, and analyse some scan results. This leads to the discussion on design pattern instantiation.

© 2007 Elsevier B.V. All rights reserved.

Keywords: Semantic web; Software design patterns; Collaborative software engineering

1. Introduction

Design patterns represent knowledge shared within the software engineering community—there are experts publishing design patterns such as the GangOf4 group [9] or Beck [2], and there are people consuming patterns in various ways, for instance by instantiating a pattern in order to solve a certain design problem, or by trying to recognise a pattern in order to comprehend the structure of a complex piece of software. As the popularity of design patterns has grown the body of knowledge about patterns has reached an extent that makes it difficult to find and select good design patterns. The question arises what a good design pattern is. The pragmatic answer is: one which the community thinks is good. This is determined either by the level of usage, or by whether the pattern author is a trusted expert in the field. Obviously, this is the case for the GangOf4 patterns. Other ways to establish trust might be associations with certain vendors, publishers, or online communities publishing or endorsing particular patterns.

If different communities talk about a certain pattern then there is no guarantee that they mean the same thing. For instance,

lets consider the rather simple Iterator pattern. According to the original definition in [9] an Iterator has four methods with the following (informally defined) semantics: a method to retrieve the current element, a method to test whether there are more elements, and methods to retrieve the first and the last element, respectively. In Grand's definition [10] the methods to fetch the first and the last element have been dropped, making it consistent with popular Java implementations of Iterator (the `Iterator` and the `Enumeration` interfaces in the `java.util` package). As many programmers learn about the Iterator pattern by generalising from a particular pattern implementation (e.g., the Java interface `java.util.Iterator` and the classes implementing it) they might expect another method to be part of the Iterator pattern: an optional¹ method to remove the current element from the underlying container. One might also argue that an Iterator should have a `release` or `close` method. This is useful to invoke clean up operations if the iterator has to fetch objects into memory on demand².

The bottom line is that there are multiple, slightly different versions of Iterator and other patterns used within the community and that there are good reasons for this. This may lead to

* Corresponding author.

E-mail addresses: J.B.Dietrich@massey.ac.nz (J. Dietrich),
chris.elgar@solnetsolutions.co.nz (C. Elgar).

¹ The semantic of optional is defined by throwing a runtime exception indicating that the method is not supported.

² This functionality is typical for database cursors. Note that `Cursor` is used as synonym for `Iterator` in [9].

communication problems that can be easily avoided by adding contextual information whenever we refer to a particular pattern. This problem can be addressed by using unique names for patterns, such as unique resource identifiers (URIs) composed of a scope (the URL referring to a pattern author, book, or web site) and a pattern name. This allows us to distinguish easily between the GangOf4 Iterator and Grand's Iterator. Still, there will be inconsistencies in how a pattern is used in context. For instance, one author might consider a pattern to be a creational pattern while another author might reject this. But knowledge about best practice design is community knowledge and as such it is distributed and inherently inconsistent. The software engineering community is accustomed to deal with this situation—source code management systems support the sharing of distributed, formalised knowledge (in particular source code, but also configuration files, data files, and documentation) and there is tool support to handle arising inconsistencies (such as merge and diff tools). However, there are many more artefacts which the community shares as well, but for which a similar infrastructure is still missing. This includes design patterns, but also anti patterns, refactorings, and component meta data.

Here we see a great potential for adopting semantic web technology, in particular the resource description framework (RDF) [18] and the web ontology language OWL [14]. The first version of a pattern description language using OWL, including the design pattern ontology, has been introduced in [4]. In the next section, we discuss this ontology. We extend the ontology presented in [4] by adding pattern refinement. In Section 3, we discuss the ontology-based recognition of design patterns and a respective prototype for Java. We also elaborate on how this relates to the semantics of the concepts defined in the ontology, and propose a formal definition of design pattern instance. In Section 4 we discuss several variants of the concept of pattern instance motivated by scan results obtained with our Java client. Finally, we compare our approach with related work and outline further directions of research.

2. Publishing design patterns

Existing design pattern catalogues such as the GangOf4 book [9] use a combination of narrative descriptions and formal models like OMT (object-modelling technique) or UML (unified modelling language) diagrams to define design patterns. This is clearly appropriate if the aim is to support software engineers to understand and apply patterns. However, this is not sufficient to underpin tools. The main use case that requires tools is automated design recovery, i.e. recognising patterns in a given program. There is a growing body of research in the area of formal design pattern description languages and detection tools, including [3,8,12,13].

In [4] we have proposed using the web ontology language OWL for this purpose. This allows us to meet the following design objectives:

- A formal, machine processable pattern definition language.
- Distributed knowledge representation without a single point of control but facilitating shared terminologies.

- A modular design that supports the separation of schema and instances.
- Meta data annotation that allows reasoning about pattern descriptions, in particular to facilitate the selection of quality patterns.
- A format that is compatible with standard web technologies (design pattern definitions can be deployed on standard http servers and can be detected and indexed by existing search engines).

RDF/OWL can be used to meet these objectives. Due to the formal semantics OWL has [15] it is safe to reason about OWL ontologies. Standard vocabularies such as Dublin Core [6] can be used to attach meta data to pattern definitions. Members of the software engineering community can easily publish patterns using a rather small set of basic concepts defined in a central ontology, and these descriptions can be published on standard http servers where they can be found by search engines like Google. Alternatively, maintained directories are possible which contain lists of URLs pointing to pattern descriptions.

The foundation of our framework is the object-oriented design ontology in which the concepts needed to describe design patterns are defined. These concepts include Pattern, PatternCatalogue, and Participant. Participant is the OWL class representing artefacts which are part of a design pattern. In particular, this includes classes and their members (constructors, fields, and methods). The respective OWL classes have a "Template" postfix indicating that their instances are variables for the respective type of objects. These classes are subclasses of Participant. For instance, AbstractFactory. ConcreteFactory is a participant of the AbstractFactory pattern. It is not a (Java or Smalltalk) class but a placeholder for such a class, i.e. a typed variable. We use the OWL class ClassTemplate to represent this kind of object. The ontology also defines properties of classes and their relationships. The current version of the ontology is deployed at: <http://www-ist.massey.ac.nz/wop/20050204/wop.rdf.xml>.

The following script is part of the ontology which has been created using the Protege ontology editor [17]:

```
<rdf:description rdf:about="#Pattern">
<rdf:type rdf:resource="#owl:Class"/>
</rdf:Description>
<!-- participants -->
<rdf:description rdf:about="#Participant">
<rdf:type rdf:resource="#owl:Class"/>
</rdf:Description>
<!-- classes -->
<rdf:description rdf:about="#ClassTemplate">
<rdf:type rdf:resource="#owl:Class"/>
<rdfs:subClassOf rdf:resource="#Participant"/>
</rdf:Description>
...
<!-- subpattern relationship -->
<rdf:Description rdf:about="#isSubPatternOf">
<rdfs:domain rdf:resource="#Pattern"/>
<rdfs:range rdf:resource="#Pattern"/>
<rdf:type rdf:resource="#owl:ObjectProperty">
<rdf:type rdf:resource="#owl:TransitiveProperty"/>
</rdf:Description>
...
```

```
<!-- classes participating in a pattern -->
<rdf:Description rdf:about="#participants">
<rdfs:range rdf:resource="#ClassTemplate"/>
<rdfs:domain rdf:resource="#Pattern"/>
<rdf:type rdf:resource="#owl:ObjectProperty"/>
</rdf:Description>
```

... The concepts defined in the ontology can then be used to formally define design patterns. This is done using a different physical resource (i.e., the respective OWL file is deployed using a different URL), the `owl:import` directive is used to refer to the ontology. This does not necessarily contradict the no single point of control objective as the current ontology serves as a reference of how such an ontology might look. Alternative ontologies are possible as well, and OWL itself offers a range of features to express the relationships between different ontologies (for instance, to declare synonyms).

We use the AbstractFactory pattern to discuss some important points of the pattern definition, the complete definition is available on the WOP web site [27] either as plain file or as generated OWLDoc report. The definition of the AbstractFactory has a unique URI that coincides with the URL of the OWL resource: <http://www-ist.massey.ac.nz/wop/20050204/AbstractFactory.rdf>.

In the header of the file, various name space prefixes are defined and the ontology language version used (OWL1.0) is declared. An `owl:import` directive imports the ontology:

```
<owl:imports rdf:resource="http://www-ist.massey.ac.nz/wop/20050204/
wop.rdf-xml"/>
```

Then the pattern itself and its participants are defined. This is done by instantiating the imported OWL classes and making assertions about properties of these instances and their relationships.

```
<!-- Abstract Factory Pattern Definition -->
<rdf:description rdf:about="#AbstractFactory">
<rdf:type rdf:resource="#Pattern"/>
<participants rdf:resource="#AbstractFactory.AbstractFactory"/>
<participants rdf:resource="#AbstractFactory.ConcreteFactory"/>
<participants rdf:resource="#AbstractFactory.AbstractProduct"/>
...
<dc:date rdf:datatype="xsl:string">1995</dc:date>
<dc:publisher rdf:datatype="xsl:string">
Addison-Wesley
</dc:publisher>
<dc:creator rdf:datatype="xsl:string">
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four)
</dc:creator>
...
</rdf:Description>
<!-- Concrete Product Participant Definition -->
<rdf:Description rdf:about="#AbstractFactory.ConcreteProduct">
<rdf:type rdf:resource="#ClassTemplate"/>
<isSubclassOf rdf:resource="#AbstractFactory.AbstractProduct"/>
<isAbstract>concrete</isAbstract>
</rdf:Description>
```

Note that meta data annotations are attached to the pattern definition. For this purpose the Dublin Core [6] vocabulary is used. This includes information about the creators, the publisher, and the time of publication. This information can then be used by clients which have found multiple pattern descriptions (for

instance, by using a search engine) to compare and select patterns. By using the common DC vocabulary we can ensure that the clients understand these annotations. The respective rules used can be written in a declarative manner, for instance by using a rule markup language such as RuleML [20]. We cannot necessarily expect that all meta information is explicitly attached to patterns, often these assertions have to be harvested from contextual information. For instance, information about the creator might only be considered as valid if it is consistent with a certified digital signature which has been used to sign the pattern definition, or if the pattern definition has been loaded from a trusted URL.

Subclasses of Participant like ClassTemplate, FieldTemplate, MethodTemplate, and RelationshipTemplate have properties and relationships which reflect the use of the respective concepts in object oriented programming: class templates can be abstract, have super class templates, and contain member templates (fields, constructors, and methods). The ontology does not support advanced features such as inner classes, annotations, static initializers, class extensions, etc., these concepts could be defined in an extension of the ontology that could then be used to define more programming language specific design patterns that require those features. This of course would narrow the scope of these pattern definitions by excluding programming languages not supporting these features. Other types of relationships usually not available in standard OO modelling languages include the actual return type of methods and invocation relationships.

The ontology also defines the `isSubPatternOf` relationship between patterns that represents pattern refinement [1,26]. The meaning of this relationship is defined as follows: a pattern A is a subpattern of a pattern B if all participants of B can be mapped to participants of A (but A may have additional participants) and the mapping retains all of the relationships and properties defined in B (but A may have additional relationships or properties). In other terms, refinement means that a monomorphism, and injective structure preserving mapping between two patterns, exist. Multiple inheritance is explicitly allowed, i.e., a pattern can be a direct subpattern of more than one other pattern. This requires a mapping between participants inherited from different parents. We use an example to discuss this. The Visitor pattern [9] is used to traverse data structures such as lists, trees, and multi dimensional structures. On the other hand, the Composite pattern [9] defines such a data structure: a tree built from container and component nodes. The Composite-Visitor is a visitor traversing the composite tree. Both patterns have different participants which have to be identified within the context of the common sub pattern. The visited element (from Visitor) is the component defined in the Composite pattern. That is, we have the following join conditions for the Element participant:

- (1) CompositeVisitor.Element = Visitor.Element.
- (2) CompositeVisitor.Element = Composite.Component.

The following listing shows the relevant parts of the CompositeVisitor definition (the DC annotations are omitted).

```

<!-- CompositeVisitor Pattern Definition -->
<rdf:Description rdf:about="#CompositeVisitor">
<rdf:type rdf:resource="#Pattern"/>
<isSubPatternOf rdf:resource="#Visitor"/>
<isSubPatternOf rdf:resource="#Composite"/>
<participants rdf:resource="#CompositeVisitor.Element"/>
<participants rdf:resource="#CompositeVisitor.ConcreteElement"/>
<rdf:Description rdf:about="#CompositeVisitor.Element">
<rdf:type rdf:resource="#ClassTemplate"/>
<inheritedFrom rdf:resource="#Visitor.Element"/>
<inheritedFrom rdf:resource="#Composite.Component"/>
</rdf:Description>
<rdf:Description rdf:about="#CompositeVisitor.ConcreteElement">
<rdf:type rdf:resource="#ClassTemplate"/>
<inheritedFrom rdf:resource="#Visitor.ConcreteElement"/>
<inheritedFrom rdf:resource="#Composite.Composite"/>
</rdf:Description>
...

```

Note that it is not necessary to redefine the properties and relationships between the participants with inheritance mappings—this can be inferred by applications based on the given semantics of the `isSubPatternOf` relationship. Multiple inheritance can lead to conflicts, for instance if one class template is “abstract” according to a statement inherited from parent 1 and “concrete” according to a statement inherited from parent 2. There is no formal mechanism to prevent these inconsistencies from occurring, however, tools could be used to verify that the inheritance relationships are consistent. Metadata (such as `DC:description` annotations) are not inherited, new meta data must be provided for the child pattern.

We call a design pattern **basic** if it does not contain `isSubPatternOf` relationships, and **derived** otherwise.

3. Instantiating patterns

The main motivation for using OWL was to have a network centric, machine processable definition of design patterns. The structure of a client that analyses software for pattern instances is relatively simple: it has to use an http client to connect to a pattern server, download and scan the patterns, translate them into constraints, and resolve these constraints with respect to the program to be analysed. A prototype for the Java language has been implemented [27] and released under the GNU General Public License (GPL). The patterns currently supported are `AbstractFactory`, `Adapter`, `Bridge`, `Composite`, `CompositeVisitor`, `Immutable`, `Prototype`, `Proxy`, `RemoteProxy`, `Singleton`, `Strategy`, `TemplateMethod` and `Visitor`. Part of this prototype is an Eclipse plugin that consists of three parts: a scanner that checks Eclipse Java projects for pattern instances and generates XML/HTML reports, a visual pattern browser, and a pattern publishing wizard that can be used to extract patterns from code and generates the appropriate RDF file.

Finding pattern instances is mainly a matter of instantiating the variables in the pattern description. Each pattern description can be mapped to a (first order logic) derivation rule as follows.

3.1. Basic pattern to rule transformation

(1) *Variables*. For each participant (i.e., instance of a class subclassing `Participant`) that occurs in the pattern definition, a

variable symbol is introduced. The name of the variable is the name of the participant. Examples³: `AbstractFactory.AbstractProduct`, `Composite.Component`.

(2) *Predicates*.

(a) For each participant class, a unary predicate is introduced. The predicate name consists of the prefix “is” and the name of the type defined in the ontology with the “Template” extension being omitted. Examples: `isClass`, `isMethod`.

(b) For each property a binary predicate is introduced that associates the object with the value of the property. The name of the predicate is the name of the property in the ontology. Examples: `visibility`, `isAbstract`.

(c) For each relationship between participants a new binary predicate is introduced associating the variables associated with the respective participants. The name of the predicate is the name of the relationship in the ontology. Examples: `isSubclassOf`, `contains`, `declaredReturnType`, `actualReturnType`.

(d) For the pattern itself a predicate called `pattern` predicate is introduced. The arity of this predicate equals the number of participants that occurs in the pattern definition, the name of the predicate is the name of the pattern in pattern definition. Examples: `Singleton`, `AbstractFactory`.

(3) *Pattern Rule*. Each pattern P is transformed into a derivation rule as follows:

(a) The head of the rule consists of the pattern predicate for P 2(d) and a list of terms containing a variable as defined in 1 for each participant that occurs in the pattern definition. Examples: `AbstractFactory (AbstractFactory.AbstractFactory, AbstractFactory.AbstractProduct, AbstractFactory.ConcreteFactory, etc.)`.

(b) For each participant p that has the type T a prerequisite is added to the rule body consisting of the predicate associated with T 2(a) and the variable terms associated with p (1). Example: `isMethod(AbstractFactory.AbstractFactory.Creator)`

(c) For each relationship r between participants p_1 and p_2 , a prerequisite is added to the rule body consisting of the predicate associated with r (2(c)) and the variables associated with the participants p_1 and p_2 (1). Examples: `isSubclassOf(AbstractFactory.ConcreteProduct, AbstractFactory.AbstractProduct)`.

(d) For each property q of a participant p a prerequisite is added to the rule body consisting of the predicate associated with q (2(b)), the variable term associated with p (1) and the constant term consisting of the property value. Example: `isAbstract (AbstractFactory.AbstractProduct, “abstract”)`.

Example: the `Abstract Factory` pattern has the following participants (abbreviated syntax—prefixes omitted) `Abstract-`

³ We use local names to simplify the examples. In general, the full URIs have to be used.

Factory, ConcreteFactory, Abstract-Product, ConcreteProduct, AbstractFactory.Creator, ConcreteFactory.Creator. The prerequisites of the pattern rule include:

- (1) isSubclassOf(ConcreteFactory, AbstractFactory).
- (2) isSubclassOf(ConcreteProduct, AbstractProduct).
- (3) contains (AbstractFactory, AbstractFactory.Creator).
- (4) contains (ConcreteFactory, ConcreteFactory.Creator).
- (5) declaredReturnType(AbstractFactory.Creator, Abstract-Product).
- (6) declaredReturnType (ConcreteFactory.Creator, Abstract-Product).
- (7) actualReturnType(ConcreteFactory.Creator, Concrete-Product).
- (8) overrides (ConcreteFactory.Creator, AbstractFactory.Creator).
- (9) is Abstract (AbstractFactory,” abstract“).
- (10) isAbstract(AbstractFactory.Creator,”abstract“).
- (11) isAbstract (AbstractProduct,” abstract“).
- (12) ...

3.2. Derived pattern to rule transformation

A derived pattern definition P is transformed into a rule as follows:

- (1) Transformation into an atomic pattern P' .
 - (a) Recursion: first transform all parent patterns (patterns referenced in statements using the isSubPatternOf relationship) into basic patterns.
 - (b) Create a copy P' of P .
 - (c) Replace all references to parent patterns in statements using isSub- PatternOf and inheritedFrom predicates by references to respective atomic patterns which are the result of (a).
 - (d) Copy all statements about participants from the parents into the pattern definition P' , replacing the participants in the statements by participants defined in P' according to the rules in the inheritedFrom statements in P' .
 - (e) Remove all statements for the isSubPatternOf and inheritedFrom predicates from P' .
- (2) The atomic pattern P' is transformed into a rule as defined above.

Note that in this definition we assume that there are no circles in the graph spawned by the pattern refinement relationship. This restriction cannot be expressed in the ontology itself as OWL is not expressive enough.

For a given pattern P we use $R(P)$ to denote the rule associated with this pattern, we use $A(P)$ to denote the head of $R(P)$ and $\text{Var}(P)$ the set of variables found in the pattern. For a given program Progr we use $A(\text{Progr})$ to denote the set of artefacts found in the project. This includes classes, members and relationships. We make no further assumptions on how these artefacts are represented, usually unique names will be used. The pattern rule allows us to formally define a pattern instance as follows: for a given pattern P and a given Program Progr a pattern

instance is a variable binding, i.e. a function $instance: \text{Var}(P) \rightarrow A(\text{Progr})$.

This translation into a logical representation of the pattern is similar to the pattern definitions found [12,7]. Standard logic programming technology can be used to compute variable bindings of the formula. What makes the structure of this rule particularly simple is the fact that it does neither contain negated prerequisites nor function symbols (nested terms). The real challenge is to find the fact base on which the rule for a given pattern operates which an inference engine can use in order to compute bindings for the variables occurring the pattern rule. The following methods can be used to compile the fact base:

- (1) *Reflection based analysis.* Object-oriented languages usually have reflection APIs (application programming interfaces) that allow the discovery of relationships between artefacts that are part of the program at runtime. These artefacts are represented as instances of special classes (such as `java.lang.Class` or `java.lang.reflect.Method` in Java), and analysis tools can gather information about the structure of the program by calling the methods of these classes. The advantage of this approach is that at this point the classes have been compiled and the artefacts have been linked together by the compiler. The drawback is that analysis relies on syntactically correct classes, and that reflection does not allow access to all the structural information needed.
- (2) *Source code analysis.* This is usually done by traversing an object model of the source code (the abstract syntax tree AST) with visitors. Source code analysis can reveal structural information that is usually not accessible with reflection based analysis tools. This includes the actual return type of methods (as opposed to the declared return type that is implemented/subclassed by the actual return type(s)), and call dependencies between methods.
- (3) *Naming pattern analysis.* If naming pattern have meaning then they can be used for structural analysis. A good example when this is the case is the Java Beans framework [11]. Naming patterns in (public) methods such as `add<ListenerType>(<ListenerType>)` and `remove<ListenerType>-(<ListenerType>)` indicate a one to many association between the class in which these method are defined and the `ListenerType`. Although this meaning is not enforced by the compiler, it underpins a wide range of tools such as user interface builders.
- (4) *Behavioral analysis.* This approach is based on the idea to run and observe the program. Technically this can be achieved by employing debugger APIs or aspect oriented programming (injecting code analysis statements). It relies on having suitable program entry points (such as main methods or test cases) which instantiate all classes, invoke all methods and execute all expressions found in the program. This is suitable in a couple of situations: to analyse programs when source code is not available, to analyse programs without explicit type information in source code (such as Smalltalk or (non-generic) containers in Java), or to

analyse programs using reflection, aspects, or other dynamic programming techniques. For instance, using behavioural analysis it is possible to obtain the actual return type of a Java method containing the following line of code: `return (MyType) Class.forName(aClassName).newInstance()`.

- (5) *Documentation and annotation analysis*. If the structure of the program has been documented, documentation analysis tools such as XDoclets can be used to analyse this information.

Having a tool that can scan programs for instances of a given pattern requires answering the question of whether the tool is complete and correct, i.e. whether it finds all pattern instances and whether it finds only those. Metrics such as precision and recall [16] measuring the ratio between detected and really implemented pattern instances could be used to quantify the level of completeness. But since there is no standard formal definition the concept of correctness is fuzzy and refers to what the community and experts think is correct. In order to prove the partial correctness and completeness of our approach we have taken examples available in books on design patterns [10,22] and turned them into JUnit test cases for our client. The shortcoming of this approach is obvious: while these examples are appropriate for readers of the respective books they do not exhibit the complexity of real world software. Scanning more complex packages leads to some interesting questions which we discuss in the next section.

Using an ontology language such as OWL removes a lot of ambiguity from the design pattern definitions. It does support some kind of reasoning about patterns (for instance, applications can take advantage of the declared transitivity of properties such as `isSubPatternOf` and `isSubClassOf`, and the consistency of pattern definitions can be checked). However, the fact that we are using semantic web technology does not guarantee that all concepts defined in the ontology have a well-defined meaning. Part of it still has to be hardwired into applications such as our client for Java. This includes that rules such as “when instantiating patterns, class templates should be mapped to objects representing classes in the target programming language” and “the relation `isSubClassOf` should only contain pairs of classes (A, B) so that A is a subclass of B according to the rules in the target programming language” are obeyed. While the constraints and the names used in our ontology make it extremely likely that every application will use them correctly, this is not directly enforced by the ontology based pattern definitions. Therefore, the semantics of the web of patterns consists of three parts (using Uschold’s terminology [25]):

- (1) *Explicit semantics intended for machine processing*. This includes reasoning using the rules built-into the ontology, such as class inheritance, symmetry of transitivity of properties etc, and reasoning using additional, explicit derivation rules.
- (2) *Formal semantics for human processing*. For example, the participants have to be mapped to the programming lan-

guage following certain rules, such as “ClassTemplate is to be instantiated by software artefacts representing classes in the target programming language”. Another example is the rule that multiple inheritance should not result in conflicting properties.

- (3) *Informal semantics for human processing, further guidelines*, such as comments in the ontology, helping the programmer to write applications which are consistent with the intended meaning of the concepts in the ontology. This includes guidelines what kind of analysis mechanism could be used to instantiate certain predicate like “the instantiation of `actualReturnType` requires AST analysis”. The definition of `Pattern` contains `owl:sameAs` statements associating the defined pattern concept with URLs containing informal descriptions⁴ of design patterns.

In the existing client, reflection and naming pattern analysis are used in order to accomplish a consistent and mainly automated mapping between the ontology and the Java environment.

4. Normal and aggregated pattern instances

Scanning non-trivial software packages such as JDBC + MySQL driver implementation and AWT + Swing reveals some interesting results. It turns out that some of the pattern instances found are not correct, i.e. they are not intended instances, although they meet the constraints in the formal pattern definition. This indicates that there are at least some additional, implicit constraints. We also know that the scanner is not complete as it does not build the complete fact base. In particular, this is the case if reflection is used in the analysed code. Thirdly, the number of correct instances found is still large, indicating that some additional concepts have to be introduced to aggregate those.

To illustrate the first case, consider the following example. Assume we have a (Java) class that requires two different clone methods with different semantics—deep and shallow copy. This is a rather common situation, and while our class could implement `Cloneable` it needs to declare at least one more `clone` like method. Lets assume this method is called `copy`. Let us further assume that the class A is abstract and has an implementation class `AImpl`. The code might look as follows:

```
public abstract class A {
    public abstract A copy();
}
public class AImpl extends A {
    public A copy () {
        A clone = new AImpl();
        // copy/take over some instance variables
        return clone;
    }
}
```

⁴ http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29 and <http://hillside.net/patterns/definition.html>.

Table 1
AbstractFactory instance found in clone example

Participant	Instance
AbstractFactory	A
Abstract Factory.Creator	A.copy
AbstractProduct	A
ConcreteFactory	AImpl
ConcreteFactory.Creator	AImpl.copy
ConcreteProduct	AImpl

These classes and methods instantiate the AbstractFactory pattern as shown in Table 1.

It is questionable whether this should be considered as an instance of the AbstractFactory pattern. The intention of AbstractFactory is to provide a public interface that is used to instantiate *another* class. This is clearly not the case if both classes are the same. The same is true for other patterns—the separation of responsibilities is a major objective for design patterns in general. The question arises whether we should add this additional constraint to pattern definitions, e.g. by introducing additional conditions such as Abstract Factory \neq AbstractProduct, or whether we consider this as a general feature patterns have. We opt for introducing a separate concept that captures this distinction. We call a pattern instance **normal** iff it is injective. This means that different participants are mapped to different artefacts. In this terminology the code given in the last example is a valid pattern instance, but not a normal pattern instance.

Even the number of normal instances found in medium sized software is surprisingly high. In a program consisting of the JDBC API (from JDK1.4.2) and the MySQL ConnectorJ version 3.0 implementation the tools finds 186 instances of Bridge and 21 instances of AbstractFactory. This high number renders the results as almost useless. A detailed analysis shows that many instances can be merged. For instance, consider the following two instances of AbstractFactory: the class `com.mysql.jdbc.jdbc2.optional.ConnectionWrapper` (the concrete factory) implements the interface `java.sql.Connection` (the abstract factory). Therefore, it implements several flavours of the `prepareStatement` (this method is overloaded) method (the creator) returning an instance of `com.mysql.jdbc.jdbc2.optional.PreparedStatementWrapper` (the concrete product) implementing the declared return type `java.sql.PreparedStatement` (the abstract product). These two instances (there are six similar instances altogether) seem to be variations of one instance. This instance is characterised by the variable bindings for the abstract participants, i.e. the class templates and method templates which have the abstract property set to the literal “abstract”.

In general, it seems to be useful to group pattern instances together. Such aggregated instances facilitates the understanding of program design as they are conceptually closer to the understanding software engineers have about design patterns. An aggregation is defined by an equivalence relation \sim between

pattern instances, i.e. a relation satisfying the following conditions:

- (1) $instance \sim instance$ reflexivity;
- (2) $instance1 \sim instance2 \Rightarrow instance2 \sim instance1$ symmetry;
- (3) $instance1 \sim instance2$ and $instance2 \sim instance3 \Rightarrow instance1 \sim instance3$ transitivity.

An aggregated pattern instance is then defined as a class of pattern instances modulo such a relation. An equivalence relation can be easily defined by a selection predicate for the pattern participants $Select \subseteq Var(P)$. Two pattern instances are considered to be equal iff the variable mappings for the selected participants are the same.

$instance1 \sim_{select} instance2 \Leftrightarrow \forall v \in Select :$

$$instance1(v) = instance2(v)$$

The most common selection predicates are

- (1) *All abstract participants.* This aggregation identifies pattern instances with the same specification part (bindings for abstract class and method templates) but different implementation parts (bindings for non abstract participants).
- (2) *All abstract classes.* This aggregation identifies pattern instances with the same bindings for the abstract class templates. For instance, consider two AbstractFactory instances having the same abstract factory class with two different (abstract) creator methods. Under 2 these two instances are considered to be equal, under 1 they are not considered to be equal. Another useful aggregation not defined by a selection predicate is related to overloading: two instances are considered equal iff they map abstract class templates to the same classes, and if methods are mapped to methods with the same name (but possibly different parameter types).

It is in principle possible to extend the ontology itself to define an aggregation relationship by marking certain pattern participants as aggregation points. This would mean that one distinguished aggregation is part of the pattern. Keeping the definitions of the aggregation and the pattern apart seems to be more flexible: different aggregations can be used with one pattern.

Table 2 shows the scan results obtained with the WOP client version 0.4 for the JDBC API (from JDK1.4.2) and the MySQL ConnectorJ 3.0 implementation.

The fact that adapter all instances are aggregated into only one aggregated instances is a consequence of adapter not having abstract participants at all.

Table 2
Number of pattern instances and aggregated instances found in the MySQL JDBC Driver

	Abstract-factory	Bridge	Strategy	Adapter
No aggregation	21	186	428	228
Normal instances only	21	150	302	228
Group by abstract participants	17	48	109	1
Group by abstract classes	9	2	6	1

5. Conclusion

In this paper, we have presented a novel approach to defining design patterns based on the web ontology language. We have shown that this yields a description that can be used by clients to detect design pattern instances in software. The inherent advantage of our approach is that this yields a definition of pattern that is machine processable, but also suitable for a community to share knowledge taking advantage of the decentralised infrastructure of the Internet. The system of loosely linked ontologies and design pattern descriptions is what we call the web of patterns. In order to become a reality, the following need to be available: effective, scalable clients in various programming languages and embedded in standard software engineering tools (Eclipse, Visual Studio), pattern authoring kits facilitating the creation and publication of new patterns, improved pattern documentation (for instance, SVG based UML diagrams attached to patterns which can be instantiated for pattern instances), pre-defined rule sets, and reasoners to select patterns. If successful, there could be a web of refactorings as well. The ultimate use case would be as follows: a software engineer tries to improve the design of a project, and presses the improve button in his development environment (IDE). The IDE issues a query to a search engine, and finds resources (RDF documents) describing refactorings suitable for the respective program. Based on user specific rules, a (trusted) refactoring is selected and applied. The user will be given a chance to customise, interrupt, or undo the refactoring.

The existing pattern scanner does only find exact matches. This could be generalised as follows: approximate matches could be detected by dropping certain constraints, or by relaxing the interpretation of certain constraints. For instance, the ontology distinguishes only between private, public, and protected methods. Some programming languages support further access modifiers like “default” (package) which a client has to map to the concepts used in the ontology. At this point applications have a certain level of freedom that will lead to slightly different scan results.

The interpretation of the constraints in the programming language (Java in our case) is done in Java classes. This association is currently hard coded in the client, although it could be published on the web as well: by making statements associating the properties used in the ontology with resources interpreting them in a given application context (Java version 1.4+/WOP version 0.9 client architecture). These resources would be URLs pointing to jar files containing the respective classes. Clients could integrate them using standard Java URL class loaders. This would support a true detect, plug and play architecture that would allow clients to process patterns defined using new constraints.

There is a rather large body of research in the area of design pattern description and detection. A wide range of formalisms are used to describe patterns, including UML Profiles [5] and logic programs [12]. Eden [7,8] proposes a logic based framework that allows precise definitions of patterns, pattern instances, and pattern refinement. His higher order entities are used to address the problems which we are trying to solve with

pattern aggregation. Tichelaar et al. [23] have developed a meta model for refactoring that is similar to the ontology presented here. This meta model contains two additional concepts Access and Invocation that can be used to describe relationships between methods and instance variables. Rosengard and Ursu [19] have analysed different existing pattern representations and argue that an ontological based approach facilitates the automatic organisation and retrieval of patterns.

The W3C has set up a working group to explore how semantic web technology can be used in software engineering [21].

References

- [1] E. Agerbo, A. Cornils, How to preserve the benefits of design patterns, in: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1822, Vancouver, Canada, October 1998*, pp. 134–143.
- [2] K. Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1996.
- [3] D. Beyer, A. Noack, C. Lewerentz, Simple and efficient relational querying of software structures, in: *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, IEEE Computer Society, 2003, pp. 216–225.
- [4] J. Dietrich, C. Elgar, A formal description of design patterns using OWL, in: *Proceedings of the ASWEC 2005*, IEEE Computer Society, 2005.
- [5] J. Dong, S. Yang, Visualizing design patterns with a UML profile, in: *Proceedings of the IEEE Symposium on Visual/Multimedia Languages (VL)*, Auckland, New Zealand, 2003, pp. 123–125.
- [6] The Dublin Core Metadata Initiative. <http://dublincore.org/>.
- [7] A. Eden, Y. Hirshfeld, A. Yehudai, LePUS—a declarative pattern specification language, Technical report 326/98, Department of Computer Science, Tel Aviv University, 1998.
- [8] A. Eden, LePUS, a visual formalism for object-oriented architectures, in: *Proceeding of the Sixth World Conference on Integrated Design and Process Technology*, Pasadena, California, June 22–28, 2002.
- [9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1994.
- [10] M. Grand, *Patterns in Java A Catalog of Reusable Design Patterns Illustrated with UML*, Wiley, 1998.
- [11] The Java Beans Specification. <http://java.sun.com/products/javabeans/docs/spec.html>.
- [12] C. Krämer, L. Prechelt, Design recovery by automated search for structural design patterns in object-oriented software, in: *Proceedings of the Third Working Conference on Reverse Engineering (WCRE '96)*, 1996.
- [13] D. Mapelsden, J. Hosking, J. Grundy, Design pattern modelling and instantiation using DPML, in: *Proceedings of the 40th International Conference on Tools Pacific*, Australian Computer Society, 2002.
- [14] Web Ontology Language (OWL), W3C Recommendation, February 10, 2004. <http://www.w3.org/TR/OWL/>.
- [15] P.F. Patel-Schneider, P. Hayes, I. Horrocks (Eds.), *OWL Web Ontology Language Semantics and Abstract Syntax*, W3C Recommendation, February 10, 2004. <http://www.w3.org/TR/owl-semantics/>.
- [16] I. Philippow, D. Streitferdt, M. Riebisch, S. Naumann, *An Approach for Reverse Engineering of Design Pattern*, Software and Systems Modeling, Springer, 2004.
- [17] The Protege Ontology Editor and Knowledge Acquisition System. <http://protege.stanford.edu/>.
- [18] Resource Description Framework (RDF), W3C Recommendation, February 10, 2004. <http://www.w3c.org/RDF/>.
- [19] J. Rosengard, M. Ursu, Ontological representations of design patterns, in: *Proceedings of the KES 2004, LNCS*, vol. 3215, Springer, 2004, pp. 28–31.
- [20] The Rule Markup Language project. <http://www.ruleml.org>.
- [21] Semantic Web Best Practices and Deployment Working Group Software Engineering Task Force (SETF). <http://www.w3.org/2001/sw/BestPractices/SE/>.

- [22] S. Stelting, O. Maassen, *Applied Java Patterns*, Prentice Hall, PTR, 2001.
- [23] S. Tichelaar, S. Ducasse, S. Demeyer, O. Nierstrasz, A meta-model for language-independent refactoring, in: *Proceedings of the International Symposium on Principles of Software Evolution*, 2000, pp. 157–169.
- [25] M. Uschold, Where are the semantics in the semantic web? *AI Mag.* 24 (3) (2003) 25–36.
- [26] J. Vlissides, *Multicast*, C++ Report, vol. 9, no. 8, SIGS Publications, New York, NY, 1997.
- [27] The WOP Project. <http://www-ist.massey.ac.nz/wop/>.