Improving Design Patterns by Description Logics: A Use Case with Abstract Factory and Strategy

Fernando Silva Parreiras, Steffen Staab
ISWeb — Information Systems and Semantic Web
Institute for Computer Science
University of Koblenz-Landau
Universitaetsstrasse 1
56070 Koblenz, Germany
(parreiras|staab)@uni-koblenz.de

Andreas Winter
Institute for Computer Science
Johannes-Gutenberg-University
Mainz
Staudingerweg 9
55128 Mainz, Germany
winter@uni-mainz.de

Abstract: This paper deals with problems in common design patterns and proposes description-logics-based modeling to remedy these issues. We exploit the TwoUse approach, which integrates OWL-DL, a W3C standard for description logics on the web, and UML-based modeling, to overcome drawbacks of the Strategy Pattern, that are also extensible to the Abstract Factory Pattern in a Model Driven Approach. The result is an OWL-based pattern to be used with design patterns: the Selector Pattern.

1 Introduction

Design patterns [GHJV95] provide elaborated, best practice solutions for commonly occurring problems in software development. During the last years, design patterns were established as general means to ensure quality of software systems by applying reference templates containing software models and their appropriate implementation to describe and realize software systems.

In addition to their advantages, [GHJV95] already characterized software design patterns by their consequences including side effects and disadvantages caused by their use. In this paper, we address the drawbacks associated with patterns based solutions for variant management [Tic97]. Design patterns rely on basic principles of reusable object design like manipulation of objects through the interface defined by abstract classes, and by favoring delegation and object composition over direct class inheritance in order to deal with variation in the problem domain.

However, the decision of what to choose from a variation typically needs to be specified at a client class. For example, solutions based on patterns like Strategy embed the treatment of variants into the clients code at various locations, leading to an unnecessary tight coupling of classes. This issue has already been identified by [GHJV95] as a drawback

*Partially supported by CAPES Brazil BEX1153054 and EU STReP-216691 MOST.
of pattern-based solutions e.g. when discussing the Strategy Pattern and its combination with the Abstract Factory Pattern. Hence, the question arises of how the selection of specific classes could be determined using only their descriptions rather than by weaving the descriptions into client classes.

Here, description logics come into play. Description logics, in general, and OWL-DL as a specific expressive yet pragmatically usable W3C recommendation [MvH04] allow for specifying classes by rich, precise logical definitions [BCM+03]. Based on these definitions, OWL-DL reasoner may dynamically infer class subsumption and object classification.

The basic idea of this paper lies in decoupling class selection from the definition of client classes by exploiting OWL-DL modeling and reasoning. We explore a slight modification of the Strategy Pattern and the Abstract Factory Pattern that includes OWL-DL modeling and that leads us to a minor, but powerful variation of existing practices: the Selector Pattern.

To realize the Selector Pattern, we apply a hybrid modeling approach in order to allow for joint UML and OWL-DL modeling, i.e. our TwoUse approach (Transforming and Weaving Ontologies and UML in Software Engineering, cf. [SPSW07]).

This paper is organized as follows. We present an example demonstrating the application of the Strategy and Abstract Factory patterns to solve a typical implementation problem in Section 2. The example illustrates the known drawbacks of the state-of-the-art straightforward adoption of these patterns. Then, we present a solution extending the existing patterns by OWL-DL based modeling in Section 3. We explain how our revision modifies the prior example and how it addresses the issues raised in the example. We describe an abstraction of the modified example, i.e. the Selector Pattern, in Section 4. We present its structure, guidelines for adoption, some consequences and related works. A short discussion of open issues concludes this paper in Section 5.

2 A Pattern Solution

This section presents a typical use case of design patterns involving the Strategy and Abstract Factory Pattern. To illustrate an application of such patterns, we take a well-known example of an order-processing system for an international e-commerce company in the United States [ST02]. This system must be able to process sales orders in many different countries, like the US and Germany, and handle different tax calculations.

Design patterns rely on principles of reusable object-oriented design [GHJV95]. In order to isolate variations, we identify the concepts (commonalities) and concrete implementations (variants) present in the problem domain. The concept generalizes common aspects of variants by means of an abstract class. When several variations are required, we subsume the variations to the contextual class, which delegates behavior to the appropriate variants. These variants are used by clients.
2.1 Applying the Strategy Pattern

Considering the principles above, we identify the class SalesOrder as context, Tax as concept, and the classes USTax and GermanTax as variants of tax calculation. Since tax calculation varies according to the country, the Strategy Pattern allows for encapsulating the tax calculation, and letting them vary independently of the context. The resulting class diagram is depicted in Fig. 1.

```plaintext
context TaskController::getRulesForCountry():Tax
body:
if so.customer.country.name = 'USA' then
  USTax.new()
else
  if so.customer.country.name = 'GERMANY' then
    GermanTax.new()
  endif
endif
endif
```

![Class Diagram](image)

Figure 1: Application of the Strategy Pattern in the problem domain.

To specify operations, we use a platform independent language, the Object Constraint Language (OCL) [OMG05] and the UML Action Semantics [OMG07b]. Owing to the fact that the UML Action Semantics does not have an standardized surface language, we use an OCL-like version, basically the operation new(). The TaskController requires the operation getRulesForCountry, which returns the concrete strategy to be used. The specification must include criteria to select from the strategies. In our example, the criterion is the country where the customer of sales order lives in.

The drawback of this solution is that, at runtime, the client TaskController must decide on the variant of the concept Tax to be used, achieved by the operation getRulesForCountry. Nevertheless, it requires the client to understand the differences between the variants, which increases the coupling between these classes.
Indeed, the decision whether a given object of SalesOrder will use the class GermanTax to calculate the tax depends on whether the corresponding Customer lives in Germany. Although this condition refers to the class GermanTax, it is specified in the class TaskController. Any change in this condition will require a change in the specification of the class TaskController, which is not intuitive and which implies an undesirably tight coupling between the classes GermanTax, Country, and TaskController.

2.2 Extending to the Abstract Factory

When the company additionally needs to calculate the freight, new requirements must be handled. Therefore, we apply again the Strategy Pattern for freight calculation. As for the tax calculation, the context SalesOrder aggregates the variation of freight calculation, USFreight and GermanFreight generalized by the concept Freight (cf. Fig. 2).

```java
context Configuration::getRulesForCountry():AbstractFactory
  body:
  if so.customer.country.name = 'USA' then
    USAbsFact.new()
  else
    if so.country.name = 'GERMANY' then
      GermanAbsFact.new()
    endif
  endif
```

![Figure 2: Strategy and Abstract Factory Patterns with configuration object.](image-url)
As we now have families of objects related to USA and Germany, we apply the Abstract Factory Pattern to handle these families. The Abstract Factory Pattern provides an interface for creating groups of related variants [GHJV95].

As one possible adaptation of the design patterns (not depicted here), the client (TaskController) may remain responsible for selecting the variants of the concept AbstractFactory to be used, i.e., the family of strategies, and may pass the concrete factory as parameter to the class SalesOrder. The class SalesOrder is associated with the class AbstractFactory, which interfaces the creation of the strategies Tax and Freight. The concrete factories USAbsFact and GermanAbsFact implement the operations to create concrete strategies USFreight, GermanFreight, GermanTax and USTax.

The adaptation of the design patterns we use as example introduces a configuration object [ST02] to shift the responsibility for selecting variants from one or several clients to a Configuration class, as depicted in Fig. 2. The class Configuration decides which variant to use. The class SalesOrder invokes the operation getRulesForCountry in the class Configuration to get the variant. These interactions are also depicted in a sequence chart in Fig. 3.

![UML Sequence diagram of Strategy and Abstract Factory Patterns with configuration object.](image-url)
2.3 Drawbacks

In general, the Strategy Pattern solves the problem of dealing with variations. However, as already documented by [GHJV95], the Strategy Pattern has a drawback. The clients must be aware of variations and of the criteria to select between them at runtime, as already described at the end of Sect.2.1.

When combining the Strategy and the Abstract Factory Pattern, the problem of choosing among the variants of the AbstractFactory remains almost the same. Indeed, the Abstract Factory Pattern just groups the families of strategies. Hence, the client must still be aware of variations.

The solution using the class Configuration does not solve this problem either. As the Configuration must understand how the variants differ, the selection is transferred from the client TaskController to the class Configuration. The coupling just migrates.

Furthermore, each occurrence of the Strategy and the Abstract Factory patterns increases the number of operations that the class Configuration must be able to handle. It makes the specification of such a class rather complex, decreasing class cohesion.

Thus, a solution that reuses the understanding of the variations without increasing the complexity is desirable. Furthermore, such a solution should allow to decide on the appropriate variants as late as possible. Separating the base of decision from the decision itself will provide an evolvable and more modular software design. In the next section we describe how an OWL-based approach can provide such a mechanism.

3 Using Patterns and Description Logics: A Use Case

A solution for the drawbacks presented at the end of Sect. 2 is to dynamically classify the context, and verify if it satisfies the set of requirements of a given variant. To do so, one requires a logical class definition language that is more expressive than UML, e.g. a description logics language like the Web Ontology Language OWL-DL [MvH04].

The strength of modeling with description logics lies in disentangling conceptual hierarchies with an abundance of relationships of multiple generalization of classes (cf. [RDH+04]). For this purpose, description logics allows for deriving concept hierarchies from logically precisely defined class axioms stating necessary and sufficient conditions of class membership. The logics of class definitions may be validated by using corresponding automated reasoning technology.

Note that reasoning could be achieved by means of OCL, since OCL constraints are essentially full first-order logic (FOL) formulas, i.e., they are more expressive than the complex class and property restriction expressions of OWL-DL, which is a decidable fragment of FOL. However, no guarantee on the completeness of reasoning with OCL is given whereas OWL-DL is equipped with automated, sound and complete reasoning services.
To benefit from the expressiveness of OWL-DL and UML modeling it is necessary to weave both paradigms into an integrated model-based approach, e.g. by using the TwoUse modeling approach (cf. [SPSW07]).

3.1 OWL for Conceptual Modeling

OWL provides various means for expressing classes, which may also be nested into each other. One may denote a class by a class identifier, an exhaustive enumeration of individuals, a property restriction, an intersection of class descriptions, a union of class descriptions, or the complement of a class description.

For sake of illustration, an incomplete specification of the problem domain using a Description Logics syntax is exposed. The identifier Customer is used to declare the corresponding class (1) as a specialization of Thing (T), since all classes in OWL are specializations of the reserved class Thing. The class Country contains the individuals USA and GERMANY (2). The class USCustomer is defined by a restriction on the property hasCountry, the value range must include the country USA (3). The description of the class GermanCustomer is analogous (5). USSalesOrder is defined as subclass of a SalesOrder with at least one USCustomer(4). The intersection of both classes is empty (⊥), i.e., they are disjoint (7). The class SalesOrder is equal to the union of GermanSalesOrder and USSalesOrder, i.e., it is a complete generalization of both classes (8).

\[
\begin{align*}
\text{Customer} & \sqsubseteq T \quad (1) \\
\{\text{USA, GERMANY}\} & \sqsubseteq \text{Country} \quad (2) \\
\text{USCustomer} & \sqsubseteq \text{Customer} \sqcap \exists \text{hasCountry}\{\text{USA}\} \quad (3) \\
\text{USSalesOrder} & \sqsubseteq \text{SalesOrder} \sqcap \exists \text{hasCustomer}.\text{USCustomer} \quad (4) \\
\text{GermanCustomer} & \sqsubseteq \text{Customer} \sqcap \exists \text{hasCountry}\{\text{GERMANY}\} \quad (5) \\
\text{GermanSalesOrder} & \sqsubseteq \text{SalesOrder} \sqcap \exists \text{hasCustomer}.\text{GermanCustomer} \quad (6) \\
\text{GermanSalesOrder} \sqcap \text{USSalesOrder} & \sqsubseteq \bot \quad (7) \\
\text{SalesOrder} & \equiv \text{GermanSalesOrder} \sqcup \text{USSalesOrder} \quad (8)
\end{align*}
\]

Different notations for OWL-DL modeling have been developed, resulting in lexical notations (cf. [HDG+06],[BPST03]) and in UML as visual notation (cf. [BHHS06],[DGDD04],[OMG07a]). When modeling the problem domain of our running example using a UML profile for OWL-DL [OMG07a], the diagram may look as depicted in Fig. 4. The number relates the list of DL statements above to the corresponding visual notation.
3.2 TwoUse-based Solution

To integrate the UML class diagram with patterns (cf. Fig. 2) and the OWL profiled class diagram (cf. Fig. 4), we rely on the TwoUse approach. The TwoUse approach uses UML profiles as concrete syntax, and allows for specifying UML entities and OWL entities using just one hybrid diagram. These entities are connected using the TwoUse profile and OCL-like expressions. This hybrid diagram, i.e., a UML class diagram with profiles for OWL and TwoUse is mapped later onto the TwoUse abstract syntax, which is a metamodel that imports the UML, OCL and OWL metamodels (cf. [SPSW07]).

The approach enables the modeler to use OCL-like expressions to describe the query operations of classes that have both semantics of an OWL class and a UML class in the same diagram. Moreover, this operation can query the OWL model, i.e., invoke a reasoning service at runtime that uses the same OWL-DL model. Hence, we can achieve dynamic classification writing OCL-like query operations in the context to classify the variation in the OWL-DL model in runtime. The result is returned as a common object-oriented class.

The OWL-DL model can be directly generated from the model, whereas the object oriented classes and OCL expressions are translated into a specific platform and later into programming code including the API for ontology and reasoning invocation.

---

1The semantics of UML and OWL-DL coincides at the M1 level, but at the M0 level the modeler has to decide whether to adopt the open world OWL-DL semantics or the closed world OCL semantics.
3.2.1 Structure

The hybrid diagram is depicted in Fig. 5. The classes Customer and Country are OWL classes and UML classes, i.e., they are hybrid TwoUse classes. They are used in the OWL-DL part of the model to describe the variations of the context SalesOrder. The TwoUse profile provides a mapping between the names in OWL and in UML in such a way that class names in both OWL and UML are preserved.

The concrete factories, i.e. the variants to be instantiated by the client TaskController are TwoUse classes as well. The concrete factories are described based on the restrictions on the class SalesOrder which must also exist in both paradigms. In the OWL-DL part of the model, the concrete factories specialize the SalesOrder, but in UML they specialize the class AbstractFactory. Hence, they do not inherit the methods of the class SalesOrder, because the associations between the variants and the context happen only in OWL-DL part of the model.

3.2.2 Participants and Collaborations

The TwoUse approach preserves the signature and behavior of existing pattern implementations, as just the body of the operation getRulesForCountry is affected. The class Configuration is no longer needed, as the selection is moved to querying the OWL-DL part of the model (cf. the query in Fig. 5).

As depicted in Fig. 6, the class TaskController invokes the operation process in the class SalesOrder (2), which invokes the operation getRulesForCountry (3). This operation calls OCL operations and operations of the OCL-DL library (4), part of the generic TwoUse implementation. The operations of the OCL-DL library queries the reasoner to classify dynamically the object SalesOrder to the appropriate subclass. The resulting OWL class, i.e., USSSalesOrder or GermanSalesOrder, is mapped onto a UML class and is returned. The remaining sequence (5-12) remains unchanged.

For instance, let \( s_1 \) be a SalesOrder with the property customer being \( cl \) with the property country being \( de \). The call \( s_1.getRulesForCountry() \) would return an object of type GermanSalesOrder.

3.2.3 Implementation

The novelty of our proposed solution is how variants are selected and instantiated. It requires behavior specification from UML and class descriptions from OWL-DL.

After the design phase, the UML class diagrams profiled with OWL and TwoUse are translated to TwoUse models, that conform to the TwoUse metamodel, using the ATL model transformation language [JK05]. The TwoUse metamodel imports the OWL, UML and OCL metamodels and extends the OCL language with operations that use the reasoner. Based on the idea that in OCL some operations are available for all UML classes, we have proposed operations available for all classes that are UML and OWL classes at the same time, i.e. TwoUse classes.
TwoUse models are translated again to the platform specific UML models and, finally, to code and to the normative OWL exchange syntax RDF/XML (please refer to [SPSW07] for more details of the implementation of TwoUse).

The OCL-DL operations reason on the OWL-DL part of the model exploiting inference services like consistency checking, concept classification and instance classification. We describe here only the two operations needed to understand the running example:

- `owlMostSpecNamedClass()`: OclType. Given an OWL-DL individual, the operation returns intersection of all OWL-DL named classes that classify this individual.
- `owl2uml(typespec: OclType): OclType`. This operation maps the identifier `typespec` in OWL onto the corresponding UML type, where the object is of the type identified by `typespec` in OWL.
In our example (cf. Fig. 5 and 6), evaluating the expression `sol.owlMostSpecNamedClass()` issues a call to the reasoner that classifies the object based on the descriptions of the classes in OWL and the properties of the object `sol`, and returns the classifier `GermanSalesOrder`. Then, the operation `sol.owl2uml(GermanSalesOrder)` maps the OWL-DL identifier to the corresponding UML identifier. The operation `sol.af.oclAsType(GermanSalesOrder)` casts the object of the UML type `AbstractFactory` as the UML type `GermanSalesOrder`.

### 3.2.4 Comparison

In the Strategy and Abstract Factory solution, the decision of which variant to use is left to the client or to the Configuration object. It requires associations from these classes (TaskController and Configuration respectively) with the concepts (Tax and AbstractFactory respectively). Furthermore, the conditions are hard-coded in the clients operations.
The TwoUse-based solution cuts these couplings, as the selection is done at the OWL-DL concept level, without any impact on the UML level, allowing the OWL-DL part of the model to be extended independently.

The descriptions of the classes USSalesOrder and GermanSalesOrder are used for the Reasoner to classify the object dynamically whenever the operation owlMostSpecNamedClass asks for. As the classification occurs at the OWL level, the operation owl2uml maps the resulting class onto a UML class. Hence, the conditions are clearly specified as logical descriptions.

When evolving from Fig. 1 to Fig. 2, the OWL-DL part of the model does not change, just the mappings. Thus, new patterns can be applied without additional effort in modeling the OWL-DL domain.

4 The Selector Pattern

After analyzing the use case of composing OWL-DL and design patterns in Sect. 3, we abstract repeatable arrangements of entities and propose a design pattern supported by OWL-DL to address decision of variations — the Selector Pattern.

The Selector Pattern provides an interface for handling variations of context. It enables the context to select the most appropriated variants based on their descriptions. Selections in the Selector Pattern are encapsulated in appropriate OCL-DL-queries against the concept, facilitating a clear separation between the base of decision and the decision itself.

![Figure 7: Structure, participants and collaborations in the Selector Pattern.](image-url)
4.1 Participants and Collaborations

The Selector Pattern is composed by a context (e.g. SalesOrder in Fig. 5), the specific variants (e.g. USAbsFact and GermanAbsFact in Fig. 5) of this context and their respective descriptions, and the concept (e.g. AbstractFactory in Fig. 5), which provides a common interface for the variations (Fig. 7). Its participants are:

- **Context** maintains a reference to the Concept object.
- **Concept** declares an abstract method behavior common to all variants.
- **Variants** implement the method behavior of the class Concept.

The **Context** has the operation select, which uses OCL-DL operations to call the reasoner and dynamically classify the object according to the logical descriptions of the variants. A **Variant** is returned as result (Fig. 7). Then, the **Context** establishes an association with the **Concept**, which interfaces the variation.

4.2 Applicability

The Selector Pattern is applicable:

- when the Strategy Pattern is applicable (cf. [GHJV95]);
- when the decision of what variant to use appears as multiple conditional statements in the operations;
- when exposing complex, case-specific data structures must be avoided.

The Selector Pattern preserves the interactions of the patterns Strategy and Abstract Factory, studied in this paper. The following steps guide the application of the Selector Pattern:

1. Design the OWL-DL part of the model using a UML profile for OWL, identifying the concept and logically describing the variations;
2. Map the overlapping classes in UML and in OWL using a UML profile;
3. Write the operation in the **Context** class corresponding to the operation selector using OCL-DL expressions.

4.3 Drawbacks

The proposed solution may seem complex for practitioners. Indeed, Applying the Selector Pattern requires sufficiently deep understanding by the developers about topics like Open
and Closed World Assumption, property restriction and satisfiability, in addition to the knowledge about the OCL-DL library. Moreover, the diagram presented by Fig. 5 is visibly more complex than the corresponding version without patterns, although applying aspect oriented techniques can minimize this problem.

Further, calls from OCL to the OWL reasoner may in general return OWL classes that are not part of the TwoUse model. This implies a dynamic diffusion of OWL classes into the UML model which must either be accommodated dynamically or which may need to raise an exception (the latter would be a good, valid solution in our running example).

Therefore, class descriptions must be sufficient for the reasoner to classify the variant, i.e. all classes and properties needed to describe the variants must also exist at the OWL level. When it is not possible, the reasoner may not be able to classify the variants correctly.

Finally, the specification of design patterns with OWL-DL is currently restricted to UML class diagrams, to the of usage OCL query operation specifications and to the adoption of non standard surface syntax for the UML Action Semantics. In fact, other UML diagrams, e.g. state machine diagrams, might be useful to model different aspects of design patterns. These other diagrams can benefit from observing reasoning behavior, as we are currently investigating.

4.4 Advantages

The application of the Selector Pattern presents some consequences, that we discuss as follows:

**Reuse.** The knowledge represented in OWL-DL can be reused independently of platform or programming language.

**Flexibility.** The knowledge encoded in OWL-DL can be modeled and evolved independently of the execution logic.

**Testability.** The OWL-DL part of the model can be automatically tested by logical unit tests, independently of the UML development.

**Ease of Adoption.** Expanding Fig. 2 with Fig. 5 and Fig. 3 with Fig. 6 in the motivating example, show that the changes required by applying the Selector Pattern in existing practices are indeed minor.

**UML paradigm dominance.** The concrete cases are bound to the context only in OWL-DL. It has no impact on the UML part of the model. The programmer freely specifies the OCL-DL operation calls when applicable.
4.5 Related Works

State-of-the-art approaches require hard-coding the conditions of selecting a particular variant to solve problems like the one given in [ST02]. Our approach relies on OWL-DL modeling and reasoning to dynamically subclassify an object when required.

Another kind of design patterns has been considered for semantic web content [Gan05]. These patterns do not address the composition of OWL-models with object-oriented software and, therefore, do not support representation of behavior as required here.

The composition of OWL with object-oriented software has been addressed by some work like [Knu04] and [PT05]. We address this composition at the modeling level in a platform independent manner [KWB02].

5 Conclusion

We have proposed a novel way of reducing coupling in important design patterns by including OWL-DL modeling. It provides a framework which integrates ontologies and UML approaches, i.e., TwoUse. We have proposed an OWL-based design pattern called Selector Pattern and discuss the impact of adopting the new approach.

We are currently extending the application of TwoUse to other design patterns concerning variant management and control of execution and method selection and the preliminary results are encouraging. Design patterns that factor out commonality of related objects, like Prototype, Factory Method and Template Method, are good candidates. New OWL-based patterns may be required to support different design patterns.

References


