Selection and Reuse of Software Design Patterns Using CBR and WordNet

Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira
Carlos Bento
CISUC - Centro de Informática e Sistemas da Universidade de Coimbra.
E-mail: pgomes@dei.uc.pt

Abstract

Software engineers and programmers deal with repeated problems and situations in the course of software design. This lead to the development of software design patterns, which can be defined as a description of an abstract solution for abstract design problems. Existing approaches to pattern application using computer tools, need the help and guidance of a human designer to select which design pattern to apply. The automation of this task opens the possibility of CASE design tools providing complete automation for the application of design patterns, and the offering new functionalities that can help the software designer to improve systems, and do better software reuse. In this paper we present an approach that automates design pattern selection and application. This approach is based on Case-Based Reasoning and WordNet, showing how they are combined to generate evolved software design diagrams. We also present an experimental study of our approach.

1. Introduction

The complexity of software systems is increasing to the extent that software development teams have difficulty to deliver systems within the schedule accorded with clients. But this is not the only problem, complexity brings bugs and unforeseen situations by the system specifications. One way to attenuate this problem is to reuse software [11, 5], not only code, but other knowledge gathered during the software development process. Among the different types of knowledge involved in the software development, is the knowledge about prototypical situations, and how they can be efficiently solved. In the software engineering research area, the development of software design patterns [7] had the goal of cataloging these common situations that appear in most of all software systems.

Software design patterns are an elegant and efficient way of solving abstract problems. Each pattern describe a solution that comprises a set of steps that can be used to design a specific part of the system being developed. They condense knowledge about how the problem should be viewed and what are the consequences of using a specific pattern. The development of these patterns were mainly for human usage, but there were efforts from the research community to automate the application of design patterns. Most of the developed approaches need human intervention, at least for selection of the pattern to be applied. In this paper we propose an approach that automates completely the application of software design patterns.

There are several research works that have common aspects with our approach. Eden et. al. [6] has proposed an approach to the specification of design patterns, and a prototype tool that automates extensively their application. This approach considers design patterns as programs that manipulate other programs, thus they are viewed as metaprograms. Eden’s approach does not automates all the process of design application, since it is the user that has to select which pattern to apply. Another important issue is the abstraction level of application, which in Eden’s case is the code level, while in our approach is the design level. Tokuda and Batory [14] also present an approach in which patterns are expressed in the form of a series of parameterized program transformations applied to software code. Like Eden’s work, this work does not address the automation of which pattern to apply. Other works on specifying design patterns and automating its application are presented by Bär [3] and Cinneide [4]. These works also automate the application of design patterns, but do not select which pattern to apply, this is done by the user. Both works deal with design modification instead of code modification. Guéhéneuc and Jussien [8] developed an application of explanation-based constraint programming for the identification of design patterns in object-oriented source code.

Our approach is based on the idea that a system can learn to select and to apply design patterns if it can store and reuse experiences that encode the situation in which patterns are used. This idea is based on an area of Artificial Intelligence [13] called Case-Based Reasoning (CBR, [9]). CBR is based on the idea not only of reasoning from experi-
ence, but also to learn from it. If the application of a specific software design pattern can be represented in the form of a case, then CBR can be used for the automation of design patterns. Our approach considers a case to be a situation where a design pattern was applied to a specific software design (in the form of Unified Modelling Language - UML [12] class diagram). Cases are stored in a case library and indexed using a general ontology (WordNet [10]). The CBR framework that we propose selects which pattern to apply, regarding the target design problem, generating a new design. It can also learn new cases from the application of design patterns. This approach has been implemented in REBUILDER, a CASE tool which provides new functionalities based on CBR.

There are two fundamental concepts that need to be further explored: software design patterns detailed in section 2, and CBR described in section 3. Then we present the REBUILDER tool, which integrates our approach, followed by the detailed description of our approach. Finally we describe the experimental work made with REBUILDER using our approach in section 7 and conclude in section 8.

2. Software Design Patterns

A software design pattern describes a solution for an abstract design problem. This solution is described in terms of communicating objects that are customized to solve the design problem in a specific context. A pattern description comprises four main elements:

Name is the description which identifies the design pattern, and is essential for communication between designers.

Problem describes the application conditions of the design pattern and the problem situation that the pattern intends to solve. It also describes the application context through examples or object structures.

Solution describes the design elements that comprise the design solution, along with the relationships, responsibilities and collaborations. This is done at an abstract level, since a design pattern can be applied to different situations.

Outcome describes the consequences of the pattern application. Most of the times patterns present trade-offs to the designer, which need to be analyzed.

Eric et al. [7] describe a catalog comprising 23 design patterns, and give a more detailed description for each pattern consisting on: pattern name and classification, pattern intent, other well-known names for the pattern, motivation, applicability, structure, participants, collaborations, consequences, implementation example, sample code, known uses, and related patterns. From these items, we draw attention to the participants and the structure. The participants describe the objects that participate in the pattern, along with their responsibilities and roles. These objects play an important role in our approach. The structure is a graphical representation of the design pattern, where objects and relations between them are represented.

A Pattern is classified based on its function or goal, which categorizes patterns as: creational, structural, and behavioral. Creational patterns have the main goal of object creation, structural patterns deal with structural changes, and behavioral patterns deal with the way objects relate with each other, and the way they distribute responsibility.

As an example of a design pattern we briefly present the Abstract Factory design pattern (see [7], page 87). The intent of this pattern is to provide an interface for creation of families of objects without specifying their concrete classes. Basically there are two dimensions in objects: object types, and object families. Concerning the type of objects, each type represents a group of objects having the same conceptual classification, like window or scrollbar. The family of objects defines a group of objects that belong to a specific conceptual family, not the same class of objects. For example, Motif objects and MS Window objects, where Motif objects can be windows, scrollbars or buttons, which exist in MS Window objects but do not have the same visual characteristics.

Suppose now, that an user interface toolkit is being implemented. This toolkit provides several types of interface objects, like windows, scroll bars, buttons, and text boxes. The toolkit can support also different look-and-feel standards, for example, Motif, MS Windows, and Macintosh. In order for the toolkit to be portable, object creation must be flexible and cannot be hard coded. A solution to the flexible creation of objects depending on the look-and-feel, can be obtained through the application of the Abstract Factory design pattern. This pattern has five types of participating objects:

- The Abstract Factory object declares an interface for operations that create abstract products.
- Concrete Factory objects implement the operations to create concrete products.
- Abstract Product objects declare an interface for a type of product object.
- Concrete Product objects define a product object to be created by the corresponding concrete factory, and also implement the Abstract Product interface.
- Client objects use only interfaces declared by Abstract Factory and Abstract Product classes.
A possible solution structure for the problem posed by the interface toolkit is depicted in figure 1. The pattern participants are: abstract factory (WidgetFactory), concrete factories (MSWindowsFactory and MotifFactory), abstract products (Window and ScrollBar), concrete products (MSWindowsWindow, MotifWindow, MSWindowsScrollBar, and MotifScrollBar), and client (Client). The create methods in the factories are the only way that clients can create the interface objects, thus controlling and abstracting object creation. The main consequences of this pattern is that it isolates concrete classes, makes exchanging product families easy, and promotes consistency among products.

3. Case-Based Reasoning

Case-Based Reasoning can be viewed as a methodology for developing knowledge-based systems [2] that makes use of experience for reasoning about problems. Its main idea is to reuse past experiences to solve new situations or problems.

A case is a central concept in CBR, and it represents a chunk of experience in a format that can be reused by a CBR system. Usually a case comprises three main parts: problem, solution, and outcome. The problem is a description of the situation that the case is representing. This can be, for example, the symptoms of a patient in case of a medical situation, or a software system’s requirements, or any description that can characterize the situation being represented. The solution describes what was used to solve the situation described in the problem. For instance, in the medical domain it can be the treatments used to heal the patient, or in the software domain a design that complies with the system’s requirements. The outcome expresses the result of the application of the solution to the problem. This means that commonly there are two possible outcomes: success or failure. A success case represents a situation in which the solution worked well, while a failure case represents a situation where the solution did not work. There can be other parts of cases like the justification that relates problem with solution through causal relations.

Another important part of the CBR methodology is the case library. This is the place where all the cases are going to be stored and organized. Due to the high number of cases that the library can have, most of the CBR systems use indexing structures that enable fast retrieval of relevant cases from memory. So most of the times a case library is more than just the place to store cases, but it defines how they are stored and how they can be accessed.

At an abstract level CBR can be described by the reasoning cycle depicted in figure 2 [1]. The reasoning process starts with the problem description, which is then transformed into a target problem (or query case). The problem is provided by a system user or by another system. The first phase in the CBR cycle is to retrieve from the case library the cases that are relevant for the target problem. The relevance of a case must be defined by the system, but the most common definition is based on feature similarity. In the end of retrieval, the best retrieved case is returned and passes to the next phase along with the target case.

The reuse phase (also designated as adaptation phase) adapts the retrieved case to the target problem, yielding a solved case (or new case). This process can be performed with several inference techniques, and many work has been done on the subject [15]. The next step for a CBR system is to revise the new case returning a tested and repaired case. This phase usually comprises two parts: verification and evaluation. While verification checks the new case consistency and coherence, the evaluation phase assesses the performance characteristics of the new case. Finally, the retain phase learns the solved case by storing it in the case library. This phase is more complex than it seems, because not all cases should be stored. If a new case is equal or very

\[1\] There are systems that retrieve more than one case, depending on the system’s purpose.
similar to a case already in the library, then it should not be stored because it brings nothing new to the system and it degrades the system’s performance. This last phase closes the CBR cycle by feeding the system with new experiences, making the system capable of learning.

4. REBUILDER

REBUILDER has two main goals: centralize the corporation’s design knowledge, and provide the software designer with a design environment capable of promoting software design reuse. This is achieved in our approach with CBR as the main reasoning process, and with cases as the main knowledge pieces. This section describes REBUILDER, detailing it’s architecture, knowledge base and reasoning engine.

REBUILDER is based on a client-server architecture comprising two servers and two clients (see Figure 3). The knowledge base (KB) used in REBUILDER comprises the WordNet server and the file server, while the clients comprise similar modules. The main difference between clients is that the manager client has an extra module allowing the KB maintenance. There can only be one server of each type, and only one manager client. Designer clients can be several depending on hardware resources.

The WordNet server comprises the WordNet ontology and the Case Indexes. WordNet is a general ontology used in REBUILDER to index cases using semantics. It also enables the assessment of semantic similarity between concepts, used in REBUILDER for case similarity. The case indexes are used for fast retrieval of cases from the case library. These indexes are associated to cases and to pieces of cases also, enabling flexible retrieval.

Clients request the file server for cases, which are in a centralized repository called case library. Each file represents an UML design (see figure 4). This enables the client to work only with the strictly necessary cases. For the optimization of this process the case indexes play a crucial role.

The data type taxonomy is used for comparing data types, and is a very simple taxonomy of the main Java data types.

The UML editor, the KB manager module and CBR engine constitute the manager client (the designer client is equal to this client except that it does not have the manager module).

The UML editor is the front-end of REBUILDER and the environment where the software designer develops designs. Apart from the usual editor commands to manipulate UML objects, the editor integrates new commands capable of reusing design knowledge. These commands are directly related with the CBR engine capabilities.

The KB manager module is used by the administrator to manage the KB, keeping it consistent and updated. This module comprises all the case-base management functions. These are used by the KB administrator to update and modify the KB.

The CBR Engine is the reasoning module of REBUILDER. This module comprises six different parts: Retrieval, Design Composition, Analogy, Design Patterns, Verification, and Learning. The Retrieval sub-module retrieves cases from the case library based on the similarity with the target problem. The Design Composition sub-module modifies old cases to create new solutions. It can take pieces of one or more cases to build a new solution by composition of these pieces. The Design Patterns sub-module uses software design patterns and CBR for generation of new designs. Analogy establishes a mapping between problem and selected cases, which is then used to build a new design by knowledge transfer between the selected case and the target problem. Case Verification checks the coherence and consistency of the cases created or modified by the system. It revises a solution generated by REBUILDER before it is shown to the software designer. The last reasoning sub-module is the retain phase, where the system learns new cases. The cases generated by REBUILDER are stored in the case library and indexed using a memory structure.

5. CBR Approach to Software Design Patterns

This section presents how software design patterns can be applied to a target design using CBR. We start by describing the patterns module, and then we describe each of its parts in more detail.

5.1 Architecture

Figure 5 presents the architecture of the patterns module. It comprises three phases: retrieve applicable Design Pattern Application (DPA) cases, select best DPA case, and apply selected DPA case. A DPA case describes the application of a specific design pattern to a software design (the
Figure 4. Example of an UML class diagram.

Figure 5. Software design pattern application module.

next sub section describes the case representation in detail). This module is used when the user decides to apply design patterns to improve the current design.

The first phase uses a target class diagram as the problem, and searches the DPA case library for DPA cases that match the problem. Then the retrieved DPA cases are ranked and the best one is selected for application, which is performed in the next step. The application of the DPA case uses the design pattern operators and yields a new class diagram, which is then used to build a new DPA case. This new case is stored in the DPA case library.

5.2 DPA Case Representation

A DPA case describes a specific situation where a software design pattern was applied to a class diagram. Each DPA case comprises: a problem and a solution description. The problem describes the situation of application based on: the initial class diagram, and the mapped participants. The initial class diagram is the UML class diagram to which the software design pattern was applied. Like the name indicates, it is the pre-modification diagram. The mapped participants are specific elements that must be present in order for the software design pattern to be applicable. Participants can be: objects, methods or attributes. Each participant has a specific role in the design pattern and it is important for the correct application of the design pattern. Each pattern has its specific set of participants. Once the participants are identified the application of a design pattern follows a specific algorithm that embeds the pattern actions. Mapping the participants is performed to select a role for some of the objects, attributes and/or methods in initial class diagram.

It is important to describe the types of participants defined in our approach. Object participants can be classes or interfaces, attribute participants correspond to class attributes, and method participants correspond to object methods. Each participant has a set of properties:

- **Role (String)**: Role of the participant in the design pattern.
- **Object (class or interface)**: Object playing the role, or in case of attribute or method participant the object to which the attribute or method belongs.
- **Method (method)**: Method playing the role in case of a method participant.
- **Attribute (attribute)**: Attribute playing the role in case of an attribute participant.
- **Mandatory (Boolean)**: Optional or not, if the participant must exist in order for the design pattern to be applicable, or just optional.
- **Unique (Boolean)**: Unique or not, if there can be one or more participants of the Role type.

The solution description of a DPA case is the name of the design pattern applied, which is then used to select the corresponding software design pattern operator. Different DPA cases can have the same solution, because what a DPA case represents is the context of application of a design pattern, and there are infinite context situations.

5.3 DPA Case Library

The DPA cases are indexed using the context synsets of the object participants (see figure 6) and only the participants (objects, attributes and methods) can be used as retrieval indexes. The WordNet structure is used as an index structure enabling the search for DPA cases in an incremental way. Each case can be stored in a file, which can be read only when needed. In figure 6 there are four indexed objects, three of them corresponding to object participants, and one a method participant, indexed by the object comprising the method.

6 Software Design Pattern Operators

For each design pattern there is one operator, for instance, the Abstract Factory design pattern has a specific
The retrieval of DPA cases is done using the WordNet as the indexing structure. The retrieval algorithm (see figure 8) starts with the target class diagram (ClassDiagram).

The participants are key objects, methods or attributes that play an important and active role in a design pattern. For example, the participants specification for the Abstract Factory pattern operator are:

- **AbstractFactory** (Type: Object; Mandatory: no; Unique: yes): Declares an interface for operations that create abstract product objects.
- **ConcreteFactory** (Type: Object; Mandatory: no; Unique: no): Implements the operations to create concrete product objects.
- **AbstractProduct** (Type: Object; Mandatory: yes; Unique: no): Declares an interface for a type of product object.
- **ConcreteProduct** (Type: Object; Mandatory: yes; Unique: no): Defines a product object to be created by the corresponding concrete factory.
- **Client** (Type: Object; Mandatory: no; Unique: no): Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Application conditions define the constraints that must be met by participant objects in order to the operator to be applied. In the case of the abstract factory the application conditions are: there must be at least one ConcreteProduct, and that all ConcreteProducts must have no public attributes or static methods.

The pattern actions for Abstract Factory are defined in the algorithm 7. This algorithm transforms ClassDiagram into NewClassDiagram through the application of the Abstract Factory design pattern.

### 6.1 Retrieval of DPA Cases

The retrieval of DPA cases is done using the WordNet as the indexing structure. The retrieval algorithm (see figure 7) starts with the target class diagram (ClassDiagram).

REBUILDER only uses UML class diagrams for reasoning tasks, the other UML diagrams are not used as queries. Then it uses the context synsets of the objects in the target diagram as probes to search the WordNet structure. The algorithm initiates the search on the synset probes, and then expands the search to neighbor synsets using all the WordNet semantic relations (is-a, part-of, member-of, and substance-of). It runs until the number of cases to be retrieved (**NumberOfCases**) is reached, or the maximum search level (**MSL**) is reached, or the **Synsets** list is empty which corresponds to the exhaustive search of the WordNet. The DPA case retrieval algorithm receives the input parameters: **ClassDiagram**, **NumberOfCases**, and **MSL**: returning a set of retrieved cases (**SelectedCases**).

### 6.2 Selection of DPA Cases

After the retrieval of the relevant cases, they are ranked accordingly to its applicability to the target diagram (**ClassDiagram**). The selection algorithm (see figure 9) starts by mapping the **ClassDiagram** to each of the retrieved cases (**SelectedCases**), resulting in a mapping for each case. The mapping is performed from the case’s participants to the target class diagram (only the mandatory participants are mapped). Associated to each mapping there is a score, which is given by the number of mapped participants. So, what this score measures is the degree of participants mapping between the DPA case and the target diagram.

The next step in the algorithm is to rank the **SelectedCases** list based on the mapping scores. The final phase
consists on checking the applicability of the best DPA case, which is done using the design pattern operator associated with the DPA case. If the application conditions of this operator are not violated, then this DPA case is returned as the selected one. Otherwise, this case is discarded and the next best case goes through the same process, until one applicable case is found or it returns null.

6.3 Application of DPA Cases

Selected the DPA case, the next step is to apply it to the target class diagram generating a new class diagram and a new DPA case. Other UML diagrams are not used for reasoning purposes, thus are not changed. The application of a DPA case is done using the pattern operator corresponding to the software design pattern given as the solution of the DPA case. Starting with the participants mapping established before, the application of the pattern is done using the application algorithm of the pattern operator.

7. Experimental Work

Experiments were performed to evaluate the performance of our approach. We used a case-base of 60 DPA cases, each one representing an application of a software design pattern to an UML class diagram. Each DPA case was generated from a different class diagram. For these experiments five software design patterns were used, the names of the patterns accordingly to [7] are: Abstract Factory, Builder, Composite, Singleton and Prototype. Each of these patterns are implemented REBUILDER along with the participants definition and operators.

We also defined 25 test class diagrams to evaluate the precision of the retrieval mechanism. These diagrams comprise three to five objects and have no method or attributes. For each test diagram the algorithm retrieved 15 DPA cases, which were then evaluated. This evaluation consisted in defining if the patterns and participants chosen were correct or if they were not applicable. The results are presented in figure 10.

The precision results show that the retrieval mechanism for this set of problems achieves 76% of correct selected DPA cases with a retrieval set size of three (cumulative result), which is in our opinion a very good indicator. As expected, the non-cumulative results degrade with the increase in the rank of retrieved cases. This also indicates that the similarity metric used to rank the retrieved cases is performing as desired, choosing the best cases for the initial places of the ranking. The cumulative values show that the precision ranges from 76% (retrieval set size of 3 and 4), to 39% (retrieval set size of 15).

8. Conclusions

This paper presents an approach to the selection and application of software design patterns in an automated way. Using CBR and WordNet we are able to store sit-
Figure 10. The precision values for the DPA retrieval algorithm.

utations where design patterns were applied. These situations, called cases, can then be reused in similar situations to guide design pattern selection and application. This approach was implemented and tested in a CASE tool named REBUILDER, which uses UML to model software systems.

An obvious advantage of our approach is the complete automation of the application of design patterns. Our approach selects which pattern to apply based on DPA cases. This enables a CASE tool to offer new functionalities, aimed for design maintenance and reuse. For instance, it can suggest to the software designer several design alternatives based on the application of different design patterns.

One limitation of our approach is that the system performance depends on the quality and diversity of the case library, which will improve as time follows. Another limitation, is that the range of case application is always restricted, and it does not outperform a software designer ability to identify which pattern to apply. Despite this, we think that our approach can provide a good contribute for design improvement, especially in situations when the user has to deal with a huge amount of objects. In this situation, automation is possibly the only way to apply design patterns, since it is difficult for the designer to deal with such an amount of objects.

Acknowledgements

This work was partially supported by POSI - Programa Operacional Sociedade de Informação of Fundação Portuguesa para a Ciência e Tecnologia and European Union FEDER, under contract POSI/33399/SRI/2000, by program PRAXIS XXI.

References

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, 1995.