Reverse Engineering of Framework Design using a Meta-Patterns-based Approach

Nuno Flores, Ademar Aguiar
Faculdade de Engenharia da Universidade do Porto
{nuno.flores, ademar.aguiar}@fe.up.pt

Abstract

Object-oriented frameworks are a powerful reuse technique but they are also very complex and difficult to design. Framework’s design aims at separating the invariant aspects across several applications in a domain – frozen spots – from the aspects that vary among applications and thus must be kept flexible and customizable – hot spots. The flexibility and extensibility provided at hot spots is usually achieved by following common design patterns, which are often hard and tiresome to identify without proper documentation. This paper proposes a reverse engineering approach to identify the design patterns used in a framework, using a high-level hot spot representation. The goals of this work include: researching a design approach that produces usable intermediate reuse information; defining a representation for design patterns based on meta-patterns; and developing a supporting tool to automate the reverse engineering process.

1. Introduction

"We have experienced a significant increase in software reusability and an overall improvement in software quality due to the application of object-oriented programming concepts in the development and (re)use of semi finished software architectures rather than just single components." [20].

Object-oriented frameworks are a powerful technique for large-scale software development capable of delivering very high levels of design reuse and code reuse [7] [8] [20].

Before being able to efficiently reuse a framework, software engineers must invest time on understanding and learning how to use it.

But frameworks are hard to learn. This difficulty is mainly due to framework design being very complex (abstract, incomplete, highly flexible and obscure), making frameworks hard to communicate, a problem aggravated by the lack of proper design documentation.

Due to its complexity, object-oriented frameworks are also very difficult to design, the reason why, when designing a new framework, developers benefit from grasping the design of other existing frameworks with the goal of identifying those parts that might be useful to reuse.

The main concern of framework design is to separate the aspects that are invariant along several applications in a domain – the frozen spots – from the other domain aspects that vary among applications and thus must be kept flexible and customizable – the hot spots.

Design patterns represent proven solutions to recurrent design problems and are extremely useful to provide the flexibility and extensibility required at hot spots.

Therefore, uncovering the design patterns used in a framework is very useful to improve the effectiveness of framework reuse.

This paper presents research work been carried on to develop a semi-automated approach to extract and reuse design patterns from existing frameworks. The approach uses a high-level hot spot representation based on meta-patterns.

The paper describes the proposed reverse-engineering process and its supporting tool, after an overview of the key design elements of frameworks, from low-level template-hook mechanisms to abstract design patterns. It concludes with an outline of the work in progress and discusses it in relation to other existing approaches.

2. Framework Design

A framework can be shortly defined as a reusable design of an application together with an implementation [6] [7] [8] [13] [17].

The definitions for a framework are not consensual and vary from author to author. In few words, a framework can be defined as a semi-complete design and implementation for an application in a given problem domain.

Frameworks are firmly in the middle of the reuse techniques. They are more abstract and flexible (and harder to learn) than components, but more concrete and
easier to reuse than a raw design (but less flexible and less likely to be applicable).

The design of a framework is typically complex [5]:
• very abstract, to factor out commonality;
• incomplete, requiring additional classes to create a working application;
• more flexible than the strictly needed by the application at hands;
• obscure, in the sense that it usually hides existing dependencies and interactions between classes.

Shortly, frameworks are considered a powerful reuse technique because they lead to one of the most important kinds of reuse, the reuse of design.

The key elements used to design a framework can be divided in three levels of abstraction: template-hook mechanisms, meta-patterns, and design patterns.

2.1. Hot Spots, Template and Hook methods

Frameworks provide their flexibility at hot spots using two essential constructs: templates and hooks. Template and hook methods are two kinds of methods extensively used in the implementation of frameworks. These terms are commonly used by several authors in [9] [19] [20] [26].

Template methods are implemented based on hook methods, and call at least one other method. A hook method is an elementary method in the context where the particular hook is used, and can be either an abstract method, a regular method, or another template method. An abstract method is a method for which only the interface is provided, and thus lacks an implementation. A regular method is a method that doesn’t call hook or template methods, but only provides a meaningful implementation.

Generally, template methods are used to implement the frozen spots of a framework, and hook methods are used to implement the hot spots. Opposite to the latter, the frozen spots are aspects that are invariant along several applications in a domain, possibly representing abstract behavior, generic flow of control, or common object relationships.

The difficulty of good framework design resides exactly on the identification of the appropriate hot spots that provide the best level of flexibility required by framework users. More hot spots offers more flexibility, but results in a framework more difficult to design and use, so somewhere in between resides a balanced design.

2.2. Meta-Patterns

The possible ways of composing template and hook classes in the hot spots of a framework were catalogued

and presented under the form of a set of design patterns, which were called meta-patterns [20] [21].

Meta-patterns are a useful abstraction that can be applied to categorize and describe framework hot spots on a meta-level.

Template and hook methods can be organized in several ways. Although they can be unified in a single class, in most of the situations it is better to put frozen spots and hot spots into separate classes. When using separate classes, the class that contains the hook method(s) is considered the hook class of the class containing the corresponding template method(s) – the template class. We can consider that hook classes parameterize the corresponding template class. The hook methods on which a template method is based can also be organized in different ways. They can be defined all in the same class, or in separate classes, in a superclass or subclass of the template class, or in any other class.

In [20], a clear distinction between the level of abstraction of design patterns and meta-patterns is made. Design patterns describe the design of specific frameworks on an abstraction level higher then the underlying programming language, albeit they provide no means of capturing the design independently of a more or less specific framework example. By using meta-patterns, the design is captured one step higher, independently of its particular application. Meta-patterns express how the required flexibility – represented by the hot spots – is gained in a particular framework.

Furthermore, seven composition meta-patterns were identified that relate templates with hooks, as depicted in Figure 1. These repeatedly occur in frameworks and, thus, in its constituent design patterns.

2.2. Design Patterns

Frameworks and design patterns are concepts closely related, representing two different categories of high-level design abstractions [14]. A single framework typically
encompasses several design patterns. Patterns provide an intermediate level of abstraction between the application level and the level of classes and objects.

A design pattern is commonly defined as a generic solution to a recurring design problem that might arise in a given context [2] [4] [10]. The relationships between individual patterns unfold in the application domain naturally and form a high level language, called a pattern language [2]. A pattern language represents the essential design knowledge of a specific application domain, i.e. the experience gained by many designers in solving a class of similar problems.

Design patterns can be viewed as a valuable means to document existing frameworks and develop new reusable object-oriented software architectures [20]. Design patterns and pattern languages are particularly good to document frameworks because they capture design experience and enclose meta-knowledge about how the flexibility was incorporated. Pattern languages help document the application domain of the framework, the design of the framework in terms of classes, objects and their relationships, and also the specifications of important framework classes. The combined use of frameworks with patterns and components is very effective, significantly helping to increase software quality and reduce development effort [8].

Although meta-patterns can be directly used to document the roles of framework participants, they are much more useful to document the roles of the participants involved in a design pattern. Generally, it is preferred to attach meta-patterns to design patterns, and design-patterns to concrete participants that instantiate them, instead of attaching meta-patterns directly to concrete participants, mainly due to the redundancy introduced, and the level of detail being too fine to be useful.

3. Reverse Engineering of Framework Design

As in any high-level software reverse engineering process, the main goal is to begin with the source code (in this case), being it a snippet, a module, package, component or framework and progressively evolve it until we’ve reached the desired level of information, namely, the design. From the source code to the constituent design patterns it implements, several consecutive steps are performed, each providing results for the next step, like a Pipes & Filters architectural style.

The approach presented here relies on capturing and analyzing information regarding the flexibility and reusability proneness of the source code, grouping it into composing structures of increasing level of abstraction and obtaining the design patterns identified with some level of certainty.

3.1. Goals

The work in progress stated in this paper thrives to reach three goals.

To emphasize reuse information by adopting an approached based on meta-patterns, where each step of the process should produce “deliverable” relevant reuse information for the user.

To define a meta-patterns-based design pattern representation through a representation of design patterns based on its meta-pattern composition. It should be generic enough to be able to represent any design pattern, whether already discovered or yet to unfold.

To provide automation and tool support, by developing a tool to automate the reverse engineering process, flexible, interactive and capable of visually deliver its results to the user.

3.2. Meta-Patterns approach

As described earlier, meta-patterns capture how templates and hooks are composed together. Also stated, design patterns can be characterized by these meta-patterns, emphasizing the flexible aspects of these framework composing elements. Taking these two facts into consideration, a reverse engineering process was devised in order to detect design patterns using these reusability-based properties.

The phased process unfolds as follows, each step leading into the next level of abstraction.

- **Source Code parsing:** at a first stage, the source code is parsed and all relevant information is gathered concerning candidate template and hook methods.
- **Hot spots detection:** a second step groups the templates and hook methods into hot spots. At this level, information regarding the framework flexibility areas is available to use (An already co-developed tool by the authors called “HotSpotter” exists that performs this step. Their intention is to adapt it to this process).
- **Meta-Patterns Detection:** the templates and hooks are analyzed and grouped into the known existing seven meta-patterns.
- **Micro-Architectures Detection:** the meta-patterns are then grouped themselves into possible micro-architectures. These are the candidate design patterns.
- **Pattern Recognizer:** this step tries to match the micro-architectures with a design pattern representation based on meta-patterns. This representation was previously stored in a repository or knowledge-base, through a Pattern Loader. This Pattern Loader is a semi-automated tool that allows a new discovered design pattern...
to be added to the repository, by producing its meta-patterns-based representation.

- **Documentation production**: the final step will consist on producing documentation of the framework flexibility areas, possibly in several formats, such as UML 2 [24], XSDoc [27] or Javadoc [12].

One of the main advantages of this approach is that, despite the final results concerning the detection of the design patterns, the data produced at intermediate steps may be “delivered”, as it is already imbued with reusability information. Notice that the emphasis goes to the reusability aspects of the framework. Already at the Hot spots detection step, there is useful and relevant information concerning the adaptable parts of the source code, namely the hot spots. Thus the process is, itself, flexible, leaving the user with the decision to proceed, or not, to the next level of abstraction.

### 3.2. Tool support

One of the pursued goals is to develop an automated (where possible) tool to support this reverse engineering process. Existing available tool solutions don’t address the reusability issue in such a straightforward manner and are rather available for use. The main concerns regarding this tool are:

i. It should be flexible, that is, it should be able to plug into an integrated development environment.
ii. It should be interactive. The user should be able to control its process and customize its behavior and preferences.
iii. It should be able to generate visual information, whether graphics or text, to store the produced data.

The work plan proposes to develop a supporting tool as an Eclipse plug-in [28] that parses Java source code and produces the earlier described results. Figure 2 depicts an overview of the tool architecture.

![Figure 2 – Tool architecture](image)

### 4. Related Work

The subject of reverse engineering design patterns has already undertaken several approaches in recent years. From automated detection tools to pattern extraction heuristics, most have a variable range of results regarding availability, flexibility, effectiveness and efficiency. Next, a brief review is made over the existing approaches catalogued in the literature.

**Pat** [16]: the Pat system is a design recovery tool for C++. Extract design information from C++ header files and stores it in a repository. Patterns are defined as PROLOG rules and the design information is translated into facts. The actual matching work is done by a PROLOG engine. Its limitations lie on dealing with behavioral patterns, since too much semantic information is required.

**KT** [3]: KT is a tool that can reverse-engineer design diagrams from Smalltalk code and use this information to detect patterns. It supports both static and dynamic modeling information of design patterns. The methods used for the detection of patterns are hard-coded directly into the KT source, thus constraining its flexibility.

**Seeman-Gudenberg** [22]: this approach introduces several ideas on how to recover design information from Java source code. The method proceeds along successive steps, revealing different layers of abstraction. In a first step, a graph is generated after a source code parsing to collect information about inheritance, method calling and naming conventions. Next, this graph is transformed by graph grammar productions until pattern detection is finally applied. This approach relies on a Pipes & Filters architectural style.

**SPOOL** [15]: the SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) project is a joint industry/university collaboration. The SPOOL reverse engineering environment has a three tier architecture: object-oriented database, repository schema and end-user tools. The lower tier provides physical storage of the reverse engineering model and design information. The middle tier contains the object-oriented schema of the reverse engineering model, comprising static structure and dynamic behavior. The upper-tier consists of end-user tools implementing domain specific functions such as source code capturing and visualization analysis.

**Albin-Amiot & Guéhéneuc** [1]: this approach bases itself on the definition of a meta-model to represent design patterns. This representation allows design patterns to be detected relying on their structural and behavioral aspects. The source code is parsed and an instantiation of the meta-model is produced accordingly. This instance is then matched against previously stored design pattern instantiations with the same meta-model.

**JBOOTRET** [18]: JBOOTRET (Jade Bird Object-Oriented Reverse Engineering Tool) uses a parser to extract the higher-level design information and conceptual model from system artifacts. The version for C++ consists of three major components: a data extractor, a knowledge manager and an information presenter. The
approach intends to separate data extraction from information representation, thus preventing repeating the analysis process for each higher-level model extraction. This design gives a enhanced degree of flexibility, enabling an easy adaptation to other programming languages.

Heuzeroth-Holl-Hogstrom-Lowe [11]: this approach presents a way to automatically detect patterns by combining both static and dynamic analysis. The former restricts the code construction and the latter the runtime behavior. This analysis does not depend on coding or naming convention. A pattern instance is defined by a tuple of program elements such as classes, methods or attributes. These elements must conform to the rules of certain design patterns. A step-by-step process filters those elements more likely to be identifies as design patterns. Nevertheless, this automated process doesn’t eliminate human intervention to ascertain the reliability of the results.

SPQR [23]: a Pipes & Filters based approach where the source code is progressively filtered and converted into intermediate notations until it finally reaches a state suitable for Formal Proof assertions of design patterns. This process requires an extensive formalization of the design patterns made a priori, based on its structures and relationships between its components. Its representation relies on production rules. It is relevant to point out the reasonable high-abstraction level of this approach, similar to the meta-pattern level.

DPVK [25]: reverse engineering tool to detect design patterns in Eiffel source code. This approach relies on a phased process starting on a static behavior analysis of candidate design patterns and ending on a dynamic behavior analysis. It relies on information stored in a repository where the design patterns have previously been catalogued according to those two aspects of behavior. It presents itself as a plug-in for IBM’s Eclipse development environment.

All these approaches deal with the aspect of flexibility and software reuse in a very shallow way, if at all. The authors are convinced that a design recovery approach relying on flexibility and reusability aspects should bring a new insight to the matter.

5. Conclusions

One of the main goals of framework development is software reuse. In order to efficiently reuse a framework, one must be aware of its design. Most commonly, the accompanying design documentation of frameworks is poor or even inexistent, thus leading to a steep learning curve to the framework (re)user.

Uncovering the design of an existing framework is a hard and tiresome task, whereas an automated aiding tool comes in order. Several solutions exist to this problem, yet none deals with the aspect of reusability in a clear, straightforward manner, and are unable to provide useful intermediate results.

The approach proposed focuses purely on identifying the elements responsible to provide framework reusability and flexibility, at various levels of abstraction. Aimed at obtaining the same results as other existing approaches it provides however usable intermediate results (hot-spots, template-hooks, and meta-patterns), even if the final results (design patterns) aren’t sufficiently accurate.

The proposed multi-phase process supports itself on the concept of meta-patterns, a meta-level representation of an abstract design pattern that describes the relationship between the elementary template and hook methods.

The intended automated tool supporting the proposed approach should be flexible, interactive and visually deliver its results to the user. An Eclipse plug-in was decided to be the preferred way to satisfy such requirements.

It is the authors’ conviction that this approach will bring new insights to the subject of reverse engineering of design patterns used in frameworks.

6. References


