Pattern-Based Reverse-Engineering of Design Components

Rudolf K. Keller **Reinhard Schauer** **Sébastien Robitaille** **Patrick Pagé**
Département IRO
Université de Montréal
C.P. 6128, succursale Centre-ville
Montréal, Québec H3C 3J7, Canada
+1 514 343 6782
{keller,schauer,robitais,pagepa}@iro.umontreal.ca

ABSTRACT
Many reverse-engineering tools have been developed to derive abstract representations from source code. Yet, most of these tools completely ignore recovery of the all-important rationale behind the design decisions that have lead to its physical shape. Design patterns capture the rationale behind proven design solutions and discuss the trade-offs among their alternatives. We argue that it is these patterns of thought that are at the root of many of the key elements of large-scale software systems, and that, in order to comprehend these systems, we need to recover and understand the patterns on which they were built. In this paper, we present our environment for the reverse engineering of design components based on the structural descriptions of design patterns. We give an overview of the environment, explain three case studies, and discuss how pattern-based reverse-engineering helped gain insight into the design rationale of some of the pieces of three large-scale C++ software systems.

Keywords
Reverse-engineering, design recovery, design component, design pattern, object-oriented design, visualization, tool support.

1 INTRODUCTION

Reverse-engineering is "the process of analyzing a subject system to (a) identify the system's components and their interrelationships and (b) create representations of a system in another form at a higher level of abstraction" [11]. The goal is to develop a more global picture on the subject system, which is the first major step towards its understanding or transformation into a system that better reflects the quality needs of the application domain at hand. One necessity to achieve this goal is a clear representation of the system's physical and logical structure; but this is still insufficient for a developer to fully comprehend the purpose of a given piece of software [5]. Underlining this statement, Booch estimates that "it takes a professional programmer about 6-9 months to become really proficient with a larger framework", and he adds that "this rate increases rather exponential to the complexity of software" [6]. We agree with Beck and Johnson that one reason for this gigantic effort for software comprehension and evolution is that "existing design notations focus on communicating the *what* of designs, but almost completely ignore the *why*" [4]. They argue that comprehension of the rationale behind the design decisions is as much important as thorough understanding of the software's structural and logical constituents. Yet, for the most part, current reverse-engineering tools completely neglect recovery of the design rationale.

Design patterns capture the rationale behind recurringly proven design solutions and illuminate the trade-offs that are inherent in almost any solution to a non-trivial design problem. In forward engineering, the advantages of design patterns are widely accepted [4], but in reverse-engineering their usefulness encounters strong resistance throughout both the pattern and the reverse-engineering communities [8]. The main arguments are that patterns can be implemented in many different ways without ever being the same twice, and that the same structure may recur with widely different intents. In addition, existing studies that were aimed at detecting design patterns in existing software systems [1, 22] failed to convey the usefulness of this approach to reverse-engineering, considering the minimal results of recovered pattern instances. Nevertheless, it is these patterns of thought that comprise the rationale of many pieces of an existing software system, and to comprehend the software, we need to recover these patterns, be it automatically or manually.

This research was supported by the SPOOL project organized by CSER (Consortium for Software Engineering Research) which is funded by Bell Canada, NSERC (National Sciences and Research Council of Canada), and NRC (National Research Council of Canada).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '99 Los Angeles CA
Copyright ACM 1999 1-58113-074-0/99/05...$5.00

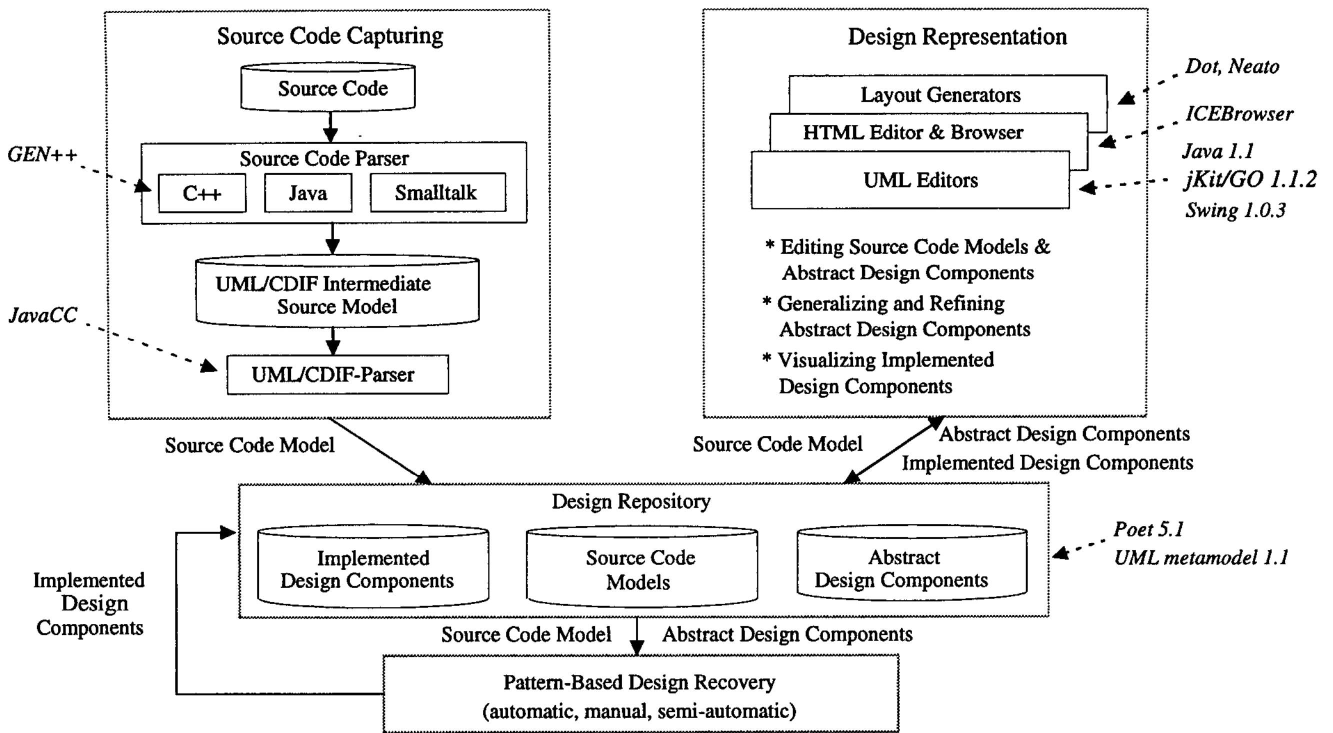

Figure 1: Overview of the SPOOL environment.

In the *SPOOL* project (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems), a joint industry/university collaboration between the software quality assessment team of *Bell Canada* and the *GELO* group at the *University of Montreal*, we have addressed this problem by implanting into the SPOOL environment for design pattern engineering [27] functionality for supporting the recovery of design patterns. Note that with "support" we underline the purpose of the environment as an aid for gaining a pattern-based overview of the software system at hand. It would be pretentious to argue that the environment itself can comprehend the rationale behind a design, "which would go far beyond the current state-of-the-art in artificial intelligence" [7]; however, by generating appropriate views, it may lead a human analyzer to the recovery of the rationale behind some of its most critical parts. Using the environment, the analyzer can zoom into these *design components*[1] that resemble patterns, extract them as diagrams in their own right, contrast the pattern description with the implemented structures, or, in the case of a false positive, dismiss the existence of the automatically identified pattern instance.

1. Note that we introduced the term *design component* as the reification of design elements, such as patterns, idioms, or application-specific solutions, and their provision as software components (*JavaBeans*, *COM objects*, or the like), which are manipulated via specialization, adaptation, assembly, and revision. We refer to [20] for further details on this approach to software composition. For the purpose of this paper we use the term *design component* as a package of structural model descriptions together with informal documentation, such as intent, applicability, or known-uses.

In this paper, we apply our environment to the reverse engineering of design components that are based on some of the design pattern descriptions defined by Gamma et al. [14]. The purpose is to introduce pattern-based reverse engineering as a valuable technique for software comprehension and thus counter the widely-held believe that design patterns are only meaningful in forward engineering. Applying our approach to several case studies extracted from industrial, large-scale software, we show that pattern-based reverse-engineering of design components is helpful for understanding software-in-the-large. In Section 2, we explain the architecture of the SPOOL environment. In section 3, we describe the three C++ systems which we used for experimentation, present three case studies that show how we applied pattern-based reverse-engineering of design components, and discuss the findings of our experiments. Section 4 compares our approach with related techniques. Section 5 concludes the paper and provides an outlook into future work.

2 REVERSE-ENGINEERING ENVIRONMENT

The purpose of the SPOOL reverse-engineering environment is to help understand software by its organization around patterns. It consists of techniques and tools for source code capturing, a design repository, and functionality for pattern-based design recovery and design representation (see Figure 1).

Source Code Capturing

The purpose of source code capturing is to extract an initial model from existing source code. At this time, we support only C++. Using the C++ source code analysis system *GEN++* [12] (*Source Code Parser*), our environment generates an ASCII-based representation of the relevant source code elements (*UML/CDIF Intermediate Source Model*). The purpose of this intermediate representation is to make the environment independent of any specific programming language, and to provide a data exchange mechanism for Bell Canada's suite of software comprehension tools. We adopted the *CDIF transfer format* [10] as the syntax and the UML metamodel 1.1 [30] as the semantic model of the intermediate format. Note that we had to extend the UML metamodel 1.1 to cover the facets of C++ we deemed essential for the recovery of pattern-based design components. An import utility (*UML/CDIF parser*), which we developed with the parser generator *JavaCC* [17], parses this UML-based CDIF format and stores the data into the design repository. At the current state of development, we capture and manage in the repository the source code information (*Source Code Model*) as listed in Table 1.

1.	*Files* (name, directory).
2.	*Classifiers - classes, structures, unions, anonymous unions, primitive types* (char, int, float, etc.), *enumerations* [name, file, visibility]. Class declarations are resolved to point to their definitions.
3.	*Generalization* relationships [superclass, subclass, visibility].
4.	*Attributes* [name, type, owner, visibility]. Global and static variables are stored in *utility* classes (as suggested by the UML), one associated to each file. Variable declarations are resolved to point to their definitions.
5.	*Operations* and *methods* [name, visibility, polymorphic, kind]. Methods are the implementations of operations. Free functions and operators are stored in *utility* classes (as suggested by the UML), one associated to each file. *Kind* stands for *constructor, destructor, standard,* or *operator.*
5.1.	*Parameters* [name, type]. The type is a *classifier.*
5.2.	*Return types* [name, type]. The type is a *classifier.*
5.3.	*Call actions* - [operation, sender, receiver]. The receiver points to the class to which a request (operation) is sent. The sender is the classifier that owns the method of the call action.
5.4.	*Create actions.* These represent object instantiations.
5.5.	*Variable use* within a method. This set contains all member attributes, parameters, and local attributes used by the method.
6.	*Friendship relationships* between classes and operations.
7.	*Class and function template instantiations.* These are stored as normal *classes* resp. *operations* and *methods.*

Table 1: Source code information managed in the repository.

Design Repository

The purpose of the design repository is to provide for centralized storage, manipulation, and querying of the *source code models*, the *abstract design components* that are to be recovered (e.g, "off-the-shelf" design patterns as found in the literature and described in template format), and the *implemented design components* within the source code models. The schema of the design repository is based on our extended *UML metamodel 1.1* [30]. The object-oriented database management system *Poet 5.1* [26] serves as the repository backend. The schema is represented as a *Java 1.1* class hierarchy. The classes within this hierarchy constitute the models of the MVC-based graphic editor of the tool. Using the precompiler of *Poet 5.1's Java Tight Binding*, an object-oriented database can be generated from this class hierarchy.

Pattern-Based Design Recovery

The purpose of pattern-based design recovery is to help structure parts of class diagrams to resemble *pattern diagrams* (see Figure 2, window 4). We envision three techniques to support this task: automatic design recovery, manual design recovery, and semi-automatic design recovery. *Automatic design recovery* relates to the fully automated structuring of software designs according to pattern descriptions, which are stored in the repository as abstract design components. We have implemented query mechanisms that can recognize the structural descriptions in the source code models, extract these from the source code, and visualize them within the class hierarchies. This technique will be further detailed in Section 3. *Manual design recovery* relates to the structuring of software designs by manually grouping design elements, such as classes, methods, attributes, or relationships, to reflect a pattern. Our environment allows the developer to select model elements and associate them with the roles of the respective pattern elements. Manual design recovery gives the human analyzer the possibility to look at a model from their own perspectives and cluster design elements to design components. It provides the flexibility that is necessary to group and communicate ad-hoc solutions as proto-patterns [2], which may at some time even become patterns. *Semi-automatic design recovery* combines both strategies, automatic and manual recovery. It may be implemented as a multi-phase recovery process. The first phase consists of the automatic detection of low-level idioms or the general core of pattern descriptions. Subsequent phases match the identified instances with more specific implementation details, which may be provided interactively by the analyzer who is in control of the recovery process. He or she may interrupt recovery runs to confirm or decline the existence of a pattern occurrence, and to manually supply specifics that are not covered by the default recovery queries. At the current stage of development, we have implemented the techniques for automated and manual design recovery.

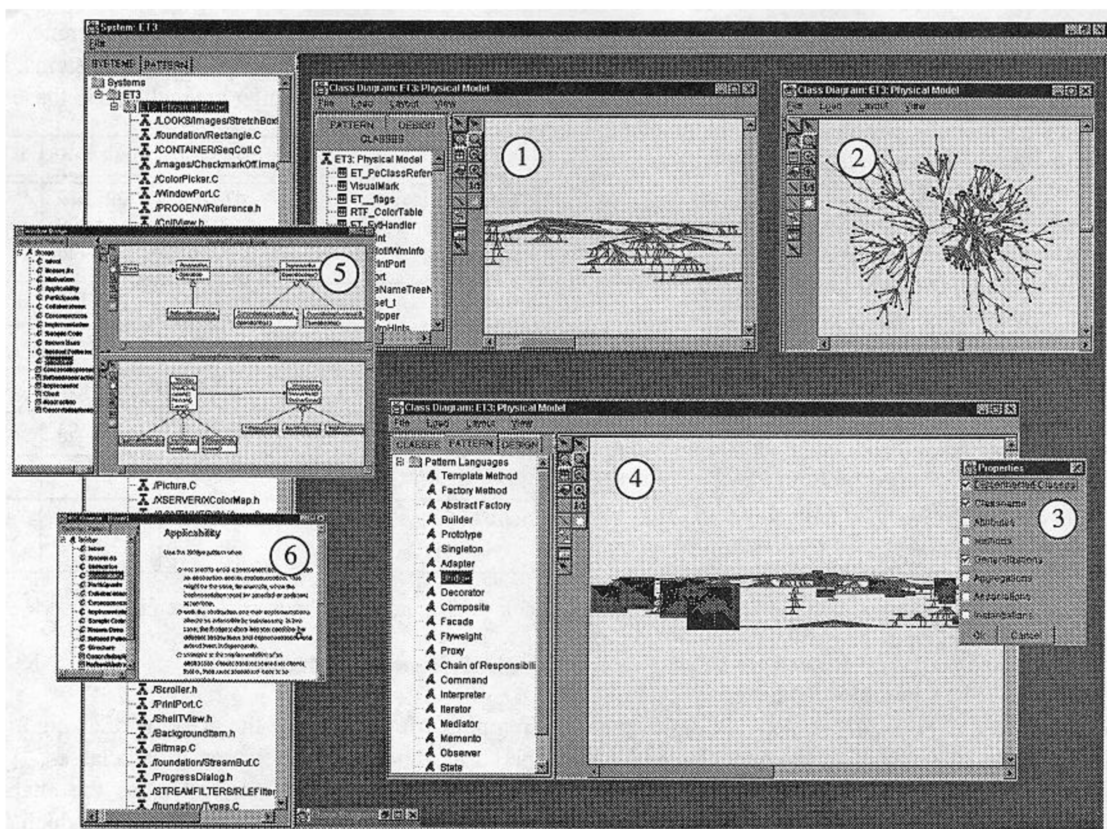

Figure 2: Graphic user interface of SPOOL environment: inheritance graph diagrams with tree layout (window l), spring layout (window 2), and property sheet (window 3) to control a diagram's content; visualization of abstract component roles in the source code model (window 4); extraction of implemented design components and comparison with the respective abstract design component (window 5); display of informal design component constituents (window 6).

Design representation

The purpose of design representation is to provide for the interactive visualization and refinement of source code models, abstract design components, and implemented components. It is our contention that only the interplay among human cognition, automatic information matching and filtering, visual representations, and flexible visual transformations can lead to the all-important why behind the key design decisions in large-scale software systems. Figure 2 illustrates our graphic environment which we developed to this end.

Windows 1 and 2 show the inheritance hierarchy of ET++ [15] (tree layout generated with Dot [21] and spring layout generated with ***Neato*** [24]). Via the property sheet associated with such diagrams (window 3), all the other association relationships stored in the repository, such as instantiation or aggregation relationships, can be illustrated as well, in both separate or combined forms. Different colors distinguish the different kinds of association relationships. On the left hand side of each window, a tree view can be optionally displayed (windows 1, 4, 5, and 6) to convey in textual form the source code models, abstract design components, or implemented design components. Through a diagram's pop-up menu, design queries on the information contents of the diagram at hand can be launched, with subsequent visualization of the query results (window 4). In our environment, each of the supported abstract design components (the pattern-like structures to be discovered) comprises a so-called reference class. This is the class in the component's structure diagram that is considered most characteristic of the component's nature*. Upon design recovery, we draw incrementally bounding boxes around the reference classes of the implementations of an abstract design component (window 4). In this way, a class that is the reference class for several of these implemented design components ("multiple reference class") will exhibit a taller bounding box than a class that is just part of a single component. Keeping the size of these bounding boxes constant during zooming leads to the effect that once their diagrams are sufficiently zoomed out (window 4), multiple reference classes will protrude from the diagram. The implemented design components can then be extracted into a separate diagram and related to the classes, methods, and attributes of their respective abstract design components (window 5), which in this study represent the descriptions of design patterns. The more informal constitu-

2. The reference class of an abstract design component can be changed interactively at the discretion of the user of the environment.

ents, such as intent, motivation, or applicability, can be viewed in the same or in separate diagrams (window 6). These informal descriptions are crucial for understanding the design, as they capture the rationale that may be at the root of the automatically identified design component.

Design representation also encompasses interactive description of design components. Using the UML class diagram notation and HTML, our environment allows for the modeling, documenting, and storing of new abstract design components in the design repository. The environment also supports the refinement and generalization of existing abstract components. This is essential as design components can be rendered in different forms. For example, a design component representing an *Adapter* pattern can be refined into a *Class Adapter* or an *Object Adapter*, and similarly, a *Composite* may be specialized into a *Transparent Composite* or a *Safe Composite* component [20].

The user interface of the SPOOL environment is implemented based on *Java 1.1*, the *Swing 1.0.3* framework for user interface widgets, and the graphic editor application framework *jKit/GO* [18]. At the current stage of development, we have implemented a class diagram editor based on the UML notation 1.1 [30]. The informal constituents of design patterns are described with *HTML*. For visualizing the HTML code, we use the *ICEBrowser* [16] JavaBeans component. To generate initial layouts of the system at hand, we developed an interface to external *layout generators*. We integrated *Dot* [21] for hierarchy layouts and *Neato* [24] for spring layouts.

3 APPLYING PATTERN-BASED REVERSE ENGINEERING

The purpose of this section is to point out the importance of pattern-based reverse-engineering of design components for the comprehension of large-scale software. We chose a case study approach to illustrate and discuss some of our findings when analyzing three industrial systems. We have selected the following abstract design components, which we based on the corresponding descriptions in the pattern catalogue of Gamma et al.: Template Method, Factory Method and Bridge [14]. Below, we first describe the three industrial systems which we analyzed. Then, we show how we reverse-engineered the selected components in System-A, System-B, and ET++, respectively.

Industrial Systems

To assess the feasibility of pattern-based reverse engineering and the usefulness of the SPOOL environment, we analyzed the source code of three industrial C++ systems. Bell Canada provided us with two large-scale systems from the domain of telecommunications. For confidentiality reasons, we call these systems *System-A* and *System-B*. Our third test system is the well-known application framework ET++ 3.0 [15], as included in the SNiFF+ development environment [29]. Table 2 shows some size metrics for these systems. Note that header files from the compiler are included in these numbers.

	System-A	System-B	ET++
Lines of code / Lines of pure comments / Blank lines	472,824 60,256 80,463	291,619 71,209 90,426	70,796 3,494 12,892
# of files (.C / .h)	1,900	1,153	485
# of classes	3,103	1,420	722
# of generalizations	1,422	941	466
# of methods	17,634	8,594	6255
# of attributes	28,360	13,624	4460
size of the system in the repository	63.1 MB	41.0 MB	19.3 MB

Table 2: Size metrics of industrial systems.

Case #1: Template Method

"Template Methods define the skeleton of an algorithm in an operation, deferring some steps to subclasses." [14] Template methods are often referred to as the characterizing building blocks of *white box* frameworks, which let clients extend the framework by overriding pre-defined *hook methods* that are called by the framework [13]. The rationale behind a Template Method is to make the steps of an algorithm easily exchangeable. The trade-off is that if not used with care, Template Methods can contribute to overly complex software, especially when the hook methods themselves are Template Methods deferring functionality to other hook methods. In large, framework-based application software, such as System-A, knowledge about the existence and location of Template Methods is crucial for the judicious evolution of the applications.

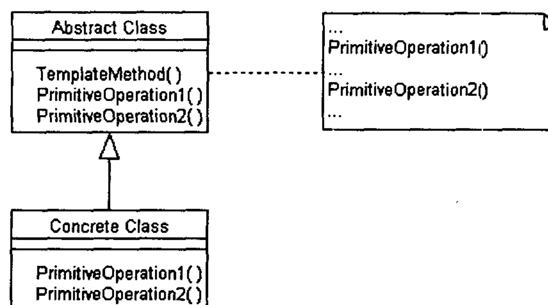

Figure 3: Structure of Template Method [14].

We reified the *Template Method* pattern (Figure 3 shows its structure) as an abstract design component, stored it in our repository, and associated it with a query that searches the source code models for the component's structure. The default implementation of the Template Method query traverses all classes (*AbstractClass*), goes into each method

(*TemplateMethod*), looks up the operation call tree for local operation calls (*PrimitiveOperation*), and verifies if *PrimitiveOperation* is polymorphic. If all conditions are met, all relevant information is passed to a *Design Component Builder* object, which creates an Implemented Design Component containing references to the identified elements in the source code model. Note that through query options, the human analyzer can specify deviations from the default behavior of the query, for instance, to recover only those TemplateMethods in which *PrimitiveOperation* in *AbstractClass* is *pure virtual* (in the case of a C++ system), or to check if *PrimitiveOperation* is overridden by at least one class (*ConcreteClass*) in the *Abstract Class's* subclass hierarchy.

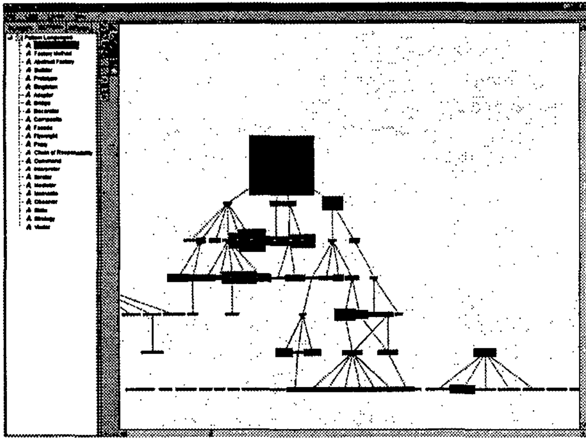

Figure 4: Template Methods in System-A.

Figure 4 illustrates the recovered Template Methods in one class tree of System-A (note that the reference class of Template Method is *AbstractClass*). This diagram clearly shows the key players within this part of the application, and conveys an impression of how many such mini-algorithms, which may be refined in subclasses, exist in the class tree. For instance, the main class, clearly visible on top of the diagram, contains 43 Template Methods. More detailed information can be recovered by zooming into the diagram, showing operations and attributes, or by spawning another diagram that shows the implementation of one particular Template Method only.

It is our experience that knowledge on both the rationale and the existence of Template Methods is essential to develop an understanding on how to hook into the mechanisms that are enforced by a framework-like architecture. Such knowledge may be of great help in flattening the learning curve of a framework.

Case #2: Factory Method

"Factory Methods define an interface for creating an object, but let subclasses decide which class to instantiate." [14] Factory Methods are specialized Template Methods in that the *PrimitiveOperation* in the *ConcreteClass* instantiates a concrete product (see Figure 5). Factory Methods are often used when different objects have the same construction process. The construction algorithm is coded in the *Creator* class, and the steps that instantiate the objects are deferred to the subclasses.

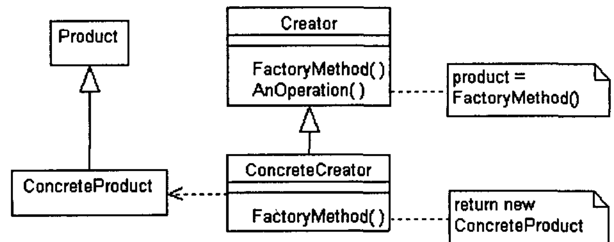

Figure 5: Structure of Factory Method [14].

The query for the Factory Method is, obviously, similar to that of the Template Method, except for the condition that the *FactoryMethod* in *ConcreteCreator* is required to instantiate a *ConcreteProduct*. By default, the query does not enforce that *ConcreteProduct* be a subclass of another class (*Product*), but this additional constraint can be specified through query options.

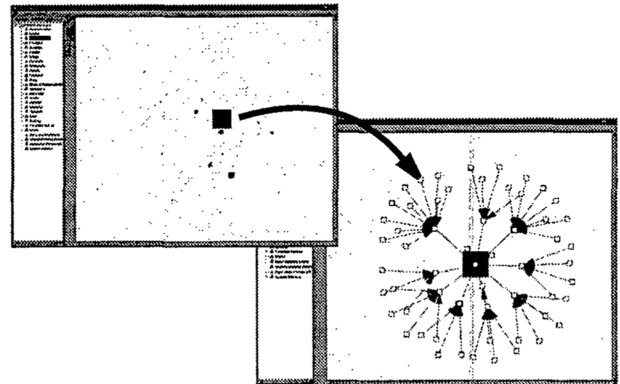

Figure 6: Factory methods in System-B: overview diagram (upper window); extracted Factory Methods (lower window).

Figure 6 illustrates the results of the Factory Method query as applied to System-B. The upper window shows the inheritance tree of all classes of System-B, which we layed out with Neato. Due to the high zooming ratio (the small points constitute large inheritance trees), the recovered design components protrude from the diagram. This is crucial information that can help find a basis for understanding a complex piece of software, which is presented in the lower window of Figure 6. We zoomed into the tallest bounding box and extracted the detailed information into a separate diagram (lower window). It illustrates a central Creator class, which defines 13 abstract Factory Method operations, and an overall 57 subclasses, which implement these operations.

This automatically generated diagram provides essential information about the rationale behind the design at hand. The developers designed this part of System-B for easy extension with new classes. This was necessary as this part of the sys-

tem deals with user interface forms and input tables, which by nature change very fast. The diagram also tells us that the designers decided to instantiate objects in the same classes that provide the functionality for their manipulation. In the example at hand, a better solution would have been the use of an Abstract Factory, which "provides an interface for creating families of related or dependent objects" [14]. This would have provided for more flexibility as the manipulation functionality could have evolved independently from the object created by the factory. Thus, a different family of objects, which may reflect changed user requirements or a different user interface platform, could have been plugged into the class hierarchy without the need of subclassing existing classes. This would have reduced the number of classes from 57 to about 30, improving understandability and maintainability.

This case study illustrates pattern-based reverse-engineering of design components as a technique that can help a human analyzer not only to comprehend a complex piece of software, but also to make substantial design improvements.

Case #3: Bridge

The intent of a Bridge pattern is to "decouple an abstraction from its implementation so that the two can vary independently." [14] The Bridge is a design technique that can avoid combinatorial explosion of class hierarchies if a domain concept in different variations can be implemented in multiple ways. If realized using inheritance, each variation would have a subclass for each of the possible implementations. To avoid this, the Bridge suggests separate class hierarchies for the abstraction and the implementation (Figure 7).

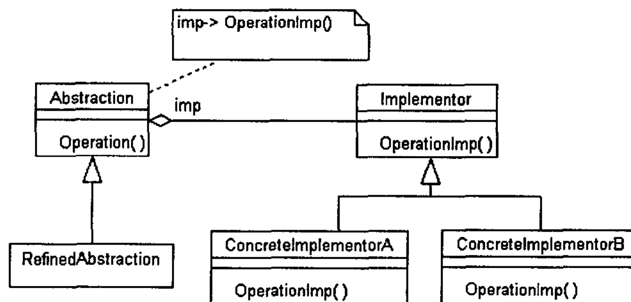

Figure 7: Structure of Bridge [14].

We include the Bridge as one of those patterns that demand human insight to be recovered from source code. The Bridge is a semantic concept that can have many forms of physical appearance in the source code. For instance, we have identified Bridges with *Abstractions* that are not subclassed, *ConcreteImplementors* that do not have a common superclass, or *OperationImps* that constitute Template Methods (see Section 4.1) in which not *OperationImp*, but its hook method is overridden. Our Bridge query captures these cases, and as an additional heuristic verifies that *Abstraction* and *Implementor* are not in the same path of the inheritance tree, which otherwise would be counter to the very intent of the Bridge. The final result was 46 Bridge-based design components in ET++, which not unsurprisingly included many false positives. It is our contention that the systematic discovery of the Bridge pattern within source code needs human insight into the problem domain of the software at hand. However, as Figure 8 illustrates, a machine can generate appropriate diagrams that are of great value for the human analyzer to identify instances of the Bridge.

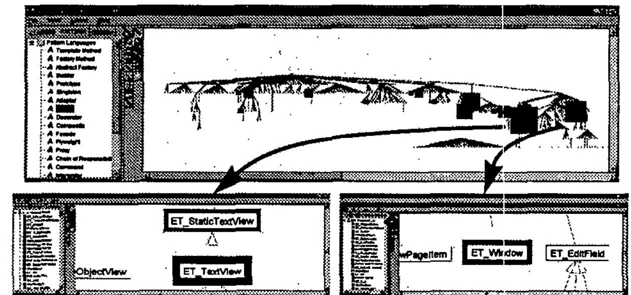

Figure 8: Bridges in ET++: overview diagram (upper window); `ET_TextView` class (lower left window); `ET_Window` class (lower right window).

In the upper window of Figure 8, we illustrate all recovered Bridges in ET++. *Abstraction* serves as the reference class, which is decorated with a bounding box for each *Operation* that delegates functionality to a subclass of the abstract *Implementor* that is the target of the maximum number of delegations. More specifically, our default Bridge query looks for classes with an instance variable (*imp*) of a type *Implementor*. It then goes into the operation call tree of each method (*Operation*) in *Abstraction*, and verifies if the receiver of an operation call (*OperationImp*) is of type *Implementor* and is overridden by at least one subclass of *Implementor* (*ConcreteImplementor*). By default, we also allow that *OperationImp* be a Template Method, meaning that not *OperationImp* itself is overridden, but one of its polymorphic hook methods (see Case #1). We discovered many Bridge *Implementor*'s in industrial code that were based on Template Methods.

Our query reported 46 Bridge-based design components in ET++, yet most of the visualized Bridges had only up to three bounding boxes (i.e., operation calls to *Implementor*), meaning that most probably these automatically recovered implementations of Bridge reflect only its structure, but not its intent. Clearly visible in Figure 8 are a few reference classes with tall bounding boxes (right side of upper window). The lower windows of Figure 8 illustrate the three reference classes that exhibit the most bounding boxes. The lower left window shows `ET_TextView` with its superclass `ET_StaticTextView`, both delegating multiple methods to `ET_Text` (not displayed in Figure 8). The documentation of ET++ [15] describes `ET_TextView` and `ET_Text` as the view and model of the MVC architectural design pattern,

which is in this example applied to text handling. In other words, subclasses of ET_TextView provide different rendering strategies for instances of ET_Text, thus serving as the abstractions for ET_Text implementors, which is the very intent of the Bridge design pattern. The lower right window of Figure 8 shows the ET_Window class with 11 bounding boxes. Gamma et al. [14] describe this case as one of the *known uses* of Bridge. In ET++, the ET_WindowPort class serves as the abstract *Implementor* for different kinds of windows, and ET_XWindowPort and ET_SunWindowPort as the *ConcreteImplementors.*

Discussion of case studies

The purpose of our work is to provide a technique that can supplement current reverse-engineering tools with the support to recover the all-important rationale behind the design decisions. We based this technique on design patterns and presented three case studies, each illustrating a different pattern on a different industrial system. Related studies on pattern detection [1, 22] provided tables indicating numbers for the detected patterns and the true pattern implementations in the investigated systems. We argue that these numbers are misleading as they neither express quality of the analyzed software or the detection tool, nor convey the rationale behind the pattern-based design (see Section 4 for further discussion). We believe in the strength of visualization and the integration of the human into the recovery process. Therefore, we selected a case study approach to convey the practicability of pattern-based reverse-engineering. However, for comparison purposes, we summarize the results of our default recovery queries in Table 2.

	System-A	System-B	ET++
Template Method	3,243	1,857	1,022
Factory Method	247	168	44
Bridge	108	95	46

Table 2: Implemented pattern-based design components.

As the structures of Template Method (Figure 3) and Factory Method (Figure 5) unambiguously reflect the intent of the respective pattern, and in light of our rich software repository, which includes information on both operation calls and polymorphic methods, we can rely on the recovered design components for both patterns being correct. The Bridge pattern, on the other hand, requires human judgement. It is one of those patterns that can be implemented in many different ways. We captured some of these implementations, and, as case study 3 illustrates, used the technique of growing bounding boxes to visually identify those *Abstractions* that delegate many operations to an *Implementor.* In System-A, for example, the reference classes of 13 out of 108 discovered Bridge design components exhibited more than 5 bounding boxes; 6 of these were surrounded by more than 50 bounding boxes, which was clearly visible in the diagram. 4 design components were real Bridge pattern implementations, the 2 others delegated many operations to another class, which provided much functionality, but did not have the semantics of an *Implementor* for the *Abstraction* at hand.

4 RELATED WORK

Below, we will briefly review a number of studies dealing with the detection and the identification of design patterns. Also, we will discuss related work addressing fine-grained design recovery. Finally, we will reflect on the added value of our approach in the realm of documentation with patterns.

Several studies reported in the literature aim at detecting design patterns in object-oriented software based on structural descriptions. Kraemer and Prechelt [22] developed a Prolog based front-end to the *Paradigm Plus* CASE tool. They observed a precision ranging from 14 to 50 percent. Similar results are reported by Antoniol et al. [1]. However, as the number of patterns found in the analyzed software was close to zero, the precision factor has little significance. Moreover, both studies report that only the header files of C++ programs were analyzed, meaning that their experiments were conducted in the absence of information on function calls and object instantiations. Moreover, Kraemer and Prechelt [22] do not report whether they considered polymorphism in their tool, and Antoniol et al. [1] mention that they do not handle polymorphism, information which we consider indispensable for the identification of pattern-like structures in source code models. Note that we consider the information currently managed by our repository (Table 1) as the minimum for serious recovery of pattern-based design components. Finally, we believe that only by the direct involvement of the human analyzer in the recovery process interesting pattern-based design components may be found.

Other studies that have influenced our work report on identifying patterns in existing software. Brown [7] reviews the Gamma et al. patterns and provides an overview of how to identify each pattern in Smalltalk software. He discusses the difficulties of recovery of patterns in existing software, but also stresses the feasibility of detecting useful patterns in source code. Martin [23] summarizes his experience when manually looking for patterns in existing software. Despite the fact that the application that his team investigated had been designed without any formal knowledge on patterns, they discovered that "in one or other form every pattern of Gamma et al. was used." Both studies convey the message that it is the human analyzer who needs to be in control of the detection process.

The recovery of design components has been subject of active research under varying terminology. Rich and Waters coin the term cliché for "commonly used combinations of elements with familiar names" [28]. Similarly, Baniassad and Murphy [3] define conceptual modules as "a set of lines of

source code that are to be treated as a logical unit." The difference between these techniques and pattern-based recovery of design components is in the level of abstraction. Whereas clichés and conceptual modules represent only small algorithms or data structures, patterns illustrate the complex relationships among the large pieces of software and, equally important, embody informal explications of the rationale behind the suggested designs. It is our contention that reverse-engineering of large-scale software needs to put more emphasize onto discovering these well-known patterns of thought. Revisiting the statement of Johnson in our introduction, it is the rationale behind the design decisions (the *why*) that needs to be recovered to gain insight into more complex pieces of software. Clichés, conceptual modules, and alike cannot convey the *why*, but certainly are much needed building blocks for achieving more elaborate recovery of pattern-based design components.

Many authors have discussed the advantages of documenting software, and in particular frameworks, with pattern [9, 19, 25, 27]. Johnson brings their cause to the point: "Patterns can describe the purpose of a framework, can let application programmers use a framework without having to understand in detail how it works, and can teach many of the design details embodied in the framework" [19]. We claim that only the *visualization* of the implemented patterns in the *context* of the application at hand will make documentation with patterns truly effective, elucidate the rationale behind the framework's design and make the applied patterns more tangible and understandable. In reverse-engineering, pattern-based re-documentation of existing frameworks and large-scale software needs sophisticated tool support allowing the human analyzer to look at the software from different perspectives, and thus gain a more-encompassing picture of the complex relationships among the system's constituents.

5 CONCLUSION AND FUTURE WORK

Design patterns capture the subtle design decisions that have proven successful in many software development projects. They document the rationale behind the design, which is so important to understand when evolving a software system to meet the continuously changing requirements. Our experience when manually analyzing parts of two telecommunications software systems of Bell Canada confirm the findings of Martin that most of the design patterns of Gamma et al. [14] are present in sizeable software systems [23]. However, we also learned that the effort for the manual recovery of a significant number of design patterns in large-scale systems is infeasible, even with the use of state-of-the-art software comprehension tools, such as SNiFF+. It is our contention, that effective pattern-based reverse-engineering of sizable software systems requires support from both pattern analysis tools and techniques, as well as the cognitive strength of the human analyzer.

In this paper we have discussed the SPOOL environment for pattern-based reverse-engineering of design components. We assessed our technology based on three case studies taken from industrial C++ software systems. The visualization technique of growing bounding boxes around the reference classes of pattern-based design components proved very helpful to gain an immediate understanding about the nature of the pattern in the software at hand. In most cases, the size of the bounding box indicated if the recovered design component also carried the intent of the respective pattern. Advanced tool support comprising extraction of the design component into a separate diagram helped verify the existence of the pattern.

Beyond extending the SPOOL environment with additional visual aids, we plan to work in five areas directly related to this study. First, we will continue extending our repository to capture all major constructs of C++ and to cover additional programming languages. The schema of the repository will be based on multiple logical layers, each increasing the level of abstraction of the source code models. Second, we are in the process of conceptualizing a generic mapping algorithm between abstract and implemented design components. This algorithm takes as input the structure diagrams of abstract design components in UML format (implemented as Java classes) and generates optimized search strategies matching these specifications with the source code model in the repository. Optional model elements and priorities among the model elements in the search strategy can be specified as UML stereotypes. Thus, the human analyzer will be able to visually specify any combination of classes, methods, and attributes to be identified in the source code models. Third, we will supplement our current visualization technique, which is based on bounding boxes around the reference classes of pattern-based design components, with alternative techniques. This includes the UML-style pattern notation [30] and customized rendering for the component at hand. For example, to convey the essence of the *Layers* architectural pattern [8], its classes should be illustrated top-down according to their association with a layer, or, once jKit/GO [18] supports three-dimensional graphic objects, within three dimensional space, each layer being a two-dimensional structure diagram and the connections among the layers being represented in the third dimension. Fourth, we have launched a Master's project to investigate recovery of pattern-based design components with full-text, pattern-matching techniques. We believe that much information about patterns can be retrieved by analyzing the names of identifiers and comments. Fifth, we will integrate our environment with the suite of software comprehension tools of Bell Canada, including source code parsers for several programming languages, a tool for clone detection, and an environment for metric analysis. Such integration will provide the software quality assessment team of Bell Canada with an industrial-strength environment that can support them in the assessment of supplier software for maintenance and evolution.

ACKNOWLEDGEMENTS

We would like to thank the following organizations for providing us with licenses of their tools, thus assisting us in the development part of our research: *Aonix* for their *Software-ThroughPictures* CASE tool, *Lucent Technologies* for their C++ source code analyzer *GEN++* and the layout generators *Dot* and *Neato*, *Instantiations* for their graphic editor framework *jKit/GO*, *Poet* for their object-oriented database management system *Poet 5.1*, and *TakeFive Software* for their *SNiFF+* software development environment.

REFERENCES

[1] Antoniol, G., Fiutem, R. and Cristoforetti, L. Design pattern recovery in object-oriented software. In *6th International Workshop on Program Comprehension* (Ischia, Italy, June 1998), 153-160.

[2] Appleton, B. Patterns and software: Essential concepts and terminology. On-line at <http://www.enteract.com/~bradapp/docs/patterns-intro.html/>.

[3] Baniassad, E. L. A. and Murphy, G. Conceptual module querying for software reengineering. In *Proc. of the 20th International Conference on Software Engineering* (Kyoto, Japan, April 1998), 64-73.

[4] Beck, K. and Johnson, R. Patterns generate architectures. In *Proc. of the 13th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science nr. 821. (1994) Springer Verlag, 139-149.

[5] Biggerstaff, T. J. Design recovery for maintenance and reuse. *IEEE Computer 22*, 7 (July 1989), 36-49.

[6] Booch, G. *Object Solutions: Managing the Object-Oriented Project.* (1996) Addison-Wesley, Menlo Park, CA.

[7] Brown, K. Design reverse-engineering and automated design pattern detection in Smalltalk. On-line at <http://hillside.net/patterns/papers/>.

[8] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern-Oriented Software Architecture - A System of Patterns.* (1996) John Wiley and Sons.

[9] Butler, G., Keller, R. K. and Mili, H. A framework for framework documentation. *ACM Computing Surveys 30*, 4 (Dec. 1998). to appear.

[10] CDIF Transfer Format. Electronic Industries Association. On-line at <http://www.cdif.org/>.

[11] Chikofsky, E. J. and Cross II, J. H. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 7, 1 (Jan. 1990), 13-17.

[12] Devanbu, P. T. GENOA - a customizable, language- and front-end independent code analyzer. In *Proc. of the 14th International Conference of Software Engineering* (Melbourne, Australia, 1992), 307–317.

[13] Fayad, M. and Schmidt, D. C. Object-oriented application frameworks. *Communications of the ACM 40*, 10, (October 1997), 32-38.

[14] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* (1995) Addison-Wesley, Menlo Park, CA.

[15] Gamma, E. and Weinand, A. ET++: A portable C++ class library for a UNIX environment. Union Bank of Switzerland. Workshop at OOPSLA'90, Ottowa, Canada, 1990.

[16] ICEBrowser. Online documentation. ICESoftt A/S, Bergen, Norway. On-line at <http://www.icesoft.no/>.

[17] JavaCC. The Java parser generator. Sun Microsystems, Palo Alto, CA. On-line at <http://www.sun.com/suntest/>.

[18] jKit/GO online documentation. Instantiations, Tualatin, OR. On-line at <http://www.instantiations.com/>

[19] Johnson, R. Documenting frameworks with patterns. In *OOPSLA'92, Sigplan Notices 27*, 10 (October 1992), 63-76.

[20] Keller, R. K. and Schauer, R. Design components: Towards software composition at the design level. In *Proc. of the 20th International Conference on Software Engineering* (Kyoto, Japan, April 1998), 302-310.

[21] Kontsofios, E. and North S. C. Drawing graphs with Dot. AT&T Bell Laboratories, Murray Hill, NJ. On-line at <http://www.research.att.com/sw/tools/graphviz/>

[22] Kraemer, C. and Prechelt, L. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. of the Working Conference on Reverse Engineering* (Monterey, CA, November 1996), 208-215.

[23] Martin, R. Discovering design patterns in existing applications. In J. Coplien and D.C. Schmidt (1995, eds.) *Pattern Languages of Program Design*, Addison-Wesley, 365-393.

[24] North S. C. NEATO User's Manual. AT&T Bell Laboratories, Murray Hill, NJ. On-line at <http://www.research.att.com/sw/tools/graphviz/>

[25] Odenthal, G. and Quibeldey-Cirkel, K. Using patterns for design and documentation. In *Proceedings of the 11th European Conference on Object-Oriented Programming* (Jyvaskyla, Finland, June 1997), 511-529.

[26] POET Java ODMG Binding. Online documentation. Poet Software Corporation, San Mateo, CA. On-line at <http://www.poet.com/>.

[27] Schauer, R. and Keller, R. K. Pattern visualization for software comprehension. In *6th International Workshop on Program Comprehension* (Ischia, Italy, June 1998), 153-160.

[28] Rich, C. and Waters R. The programmer's apprentice: A research overview. *IEEE Computer 21*, 11 (November 1988), 11-24.

[29] SNiFF+. Documentation set. On-line at <http://www.takefive.com/>.

[30] UML. Documentation set version 1.1 (Sept. 1997). On-line at <http://www.rational.com/>.