

# Hot Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods

Reinhard Schauer

Sébastien Robitaille

François Martel

Rudolf K. Keller

Département IRO

Université de Montréal

C.P. 6128, succursale Centre-ville

Montréal, Québec H3C 3J7, Canada

+1 514 343 6111 x1805

{schauer, robitaits, martelf, keller}@iro.umontreal.ca

## Abstract

*The success of an object-oriented software development project highly depends on how well the designers can capture the Hot Spots of the application domain, that is, those aspects that should be kept flexible to accommodate reuse and change. Yet, all too often, Hot Spots are hardly documented, and over years of software evolution, the source code that reifies them becomes increasingly entangled with the application specific code. This blurring of the flexible with the rigid parts makes an application hard to maintain, prone to unexpected change impact, and immobile for reuse in related areas. In this paper, we apply SPOOL, our prototype environment for reverse engineering, to the recovery of Hot Spots in C++ software. We base the technique for Hot Spot recovery on the design concept of template methods. We present the approach and the interactive analysis capabilities of SPOOL to visualize, browse, and inspect Hot Spots in both separate and contextual form. The findings are validated based on two industrial systems.*

**Keywords:** Hot Spot, reverse engineering, template method, design component, design pattern, object-oriented design.

## 1. Introduction

“A satisfactory modular decomposition ... should be both open and closed” [14]. Using this phrase, Meyer coined one of the key principles of object-oriented software design, the

*Open-Closed Principle*. With *open* he refers to the capability of a module, or class, to be extensible to behave in different ways; with *closed* he suggests that the source code of such a module be immutable. The rationale behind this principle is to keep the implementation of an abstraction, and in particular those parts that often change, open for adaptation to different context-dependent variations or unforeseeable requirements while providing a stable core on which different applications can rely. These aspects of an application domain that should be kept resilient to adaptation and extension are often referred to as *Hot Spots*, and the organization of an application around such Hot Spots largely determines how well it is closed for modification, yet open for extension [18, 21].

Hot Spots may be considered the critical parts of an object-oriented design and must be thoroughly understood when reusing and maintaining the design and its application specific extensions. Yet, all too often, object-oriented designs are released without proper documentation of their Hot Spots, and this makes them difficult to understand and prone to misuse. We believe that knowledge about the Hot Spots and how they are accessed by the client software is crucial for nearly all activities of software development and maintenance, be it construction, comprehension, or evolution.

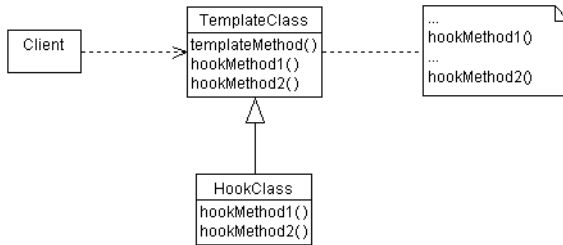
The notions of *template methods* and *hook methods* are often applied to distinguish the two kinds of methods that are used to implement Hot Spots. Template methods implement the invariant parts of a Hot Spot and glue the variant hook methods together. They support adherence to the Open-Closed Principle as they are part of the *closed* framework of an object-oriented design and since they define the hook methods, which are *open* for overriding in application-specific classes. Depending on the level of flexibility that is desired for the Hot Spot at hand, template methods can be joined with their respective hook methods by either inherit-

---

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).

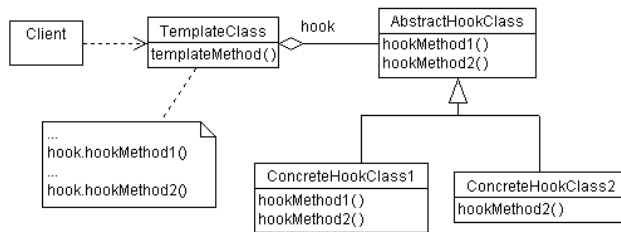
ance or composition. In the literature, such a combination of template and hook methods is often referred to as the *Template Method* design pattern [4, 18, 27]<sup>1</sup>. Template Methods and Hot Spots differ in that a Template Method emphasizes an invariant template method and associates the variant hook methods that it invokes, whereas a Hot Spot stresses a variant hook method and associates the template methods from which it is called.

*Inheritance Template Methods* let clients extend a framework by overriding abstract hook methods that are defined in the same class as the template method (*TemplateClass*) or inherited from a superclass of *TemplateClass* (Figure 1).



**Figure 1: Structure of the Inheritance Template Method.**

*Composition Template Methods*<sup>2</sup> provide interfaces (*AbstractHookClass*) with operations that are to be overridden by the client code. Instances of the *ConcreteHookClass* can be plugged into the *TemplateClass* (Figure 2). Note that many design patterns in Gamma et al. are based on Composition Template Methods, such as Builder, Strategy, or Bridge [4].



**Figure 2: Structure of the Composition Template Method.**

Inheritance Template Methods and Composition Template Methods are basic techniques for implementing Hot Spots. Knowledge about the existence of Hot Spots, and in particular the location of the invariant template methods and

the variant hook methods, is crucial for the comprehension, maintenance, and evolution of a framework and its derived applications. To this end, we extended the SPOOL environment [10, 11, 20] with recovery, visualization, and analysis support for Hot Spots and applied it to the study of two industrial systems. In Section 2, we introduce the SPOOL environment and describe the extensions made for the purpose of this study. Section 3 summarizes the research approach taken. Section 4 provides an analysis of the results based on two industrial systems. Section 5 reviews related work and, finally, Section 6 rounds up the paper with a conclusion and an outlook into future work.

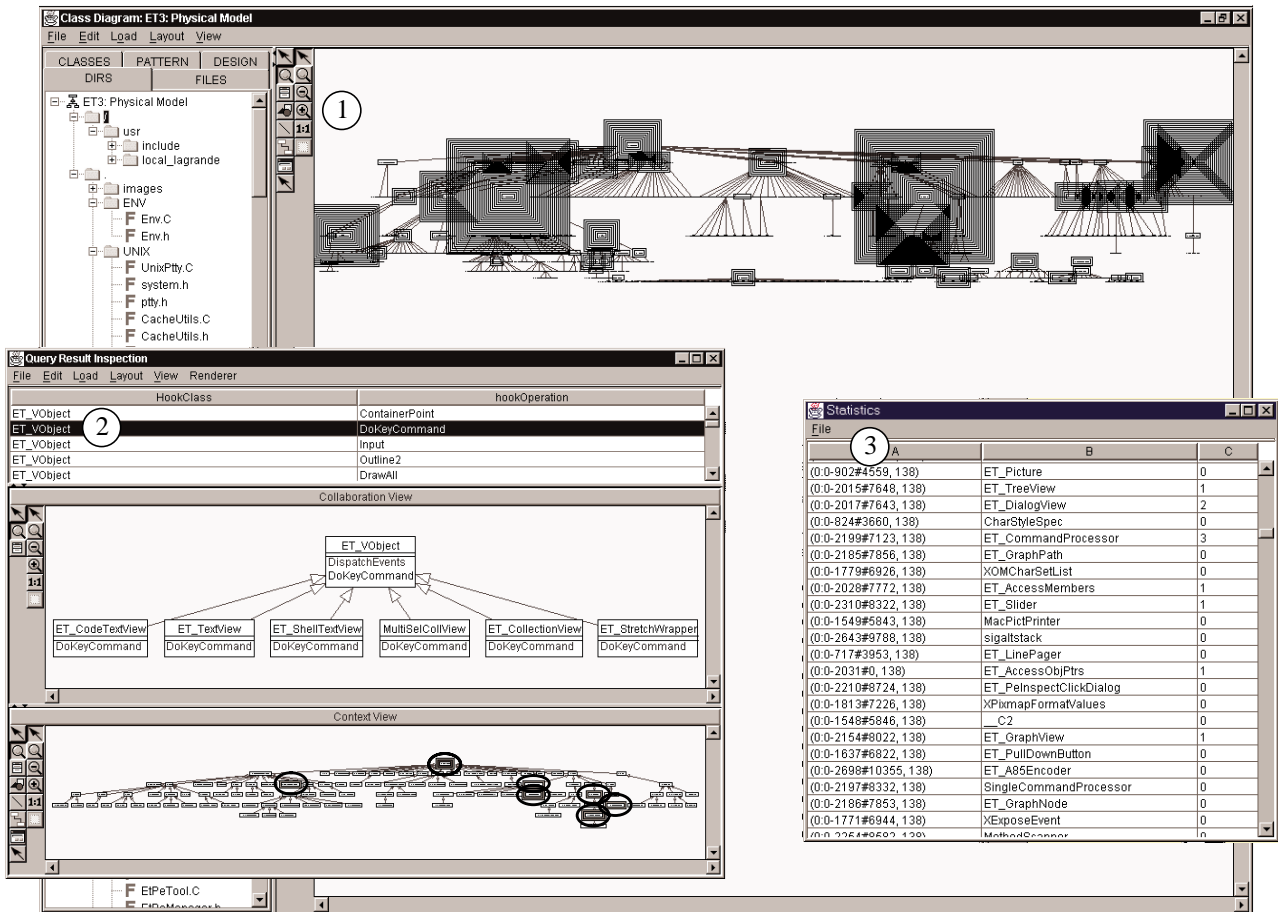
## 2. Tool support for Hot Spot recovery

Automated tool support is crucial for the comprehension of large-scale, object-oriented software and involves compressing and clustering of the vast amount of information that is contained in the source code; however, software comprehension demands more than mere understanding of the static structure of the source code. We believe that it is more important to comprehend the rationale and the requirements that have led to the design solutions that are implemented in the system at hand [11]; yet, the recovery of these facets can hardly be fully automated. Each application domain, design technique, and programming language has its own particularities, and a machine would need to capture analysis heuristics for the many different combinations to be effective. Hence, we based the design of the SPOOL environment on the premise that only the tight integration of the human analyzer into the recovery process can lead to valuable design information. The tool should provide support for the automation of analysis heuristics, but it is the human who eventually decides what techniques to use, how to apply and join them, and how to interpret the results.

The SPOOL reverse engineering environment is written in *Java* and is based on a set of off-the-shelf components and application frameworks, which we extended and assembled. These include the source code analysis systems *GEN++* [3] and *Datrix* [13], the layout engines *Dot* [12] and *Neato* [15], the object-oriented database management system *Poet51* [17] for storing the UML 1.1 [25] based models extracted from the source code, the graphic editor framework *jKit/GO* [6] on which we built our UML editor, and *Swing* as the user interface framework. For a more detailed discussion of the SPOOL environment, refer to [11], where we also outline several case studies on pattern-based reverse engineering of design components. Note that the set of techniques supported by SPOOL aims at design component recovery as well as design composition; however, in this paper we describe only those features of the SPOOL environment that are relevant for Hot Spot recovery based on Template Methods.

<sup>1</sup> We use capital letters to distinguish the *Template Method* design pattern from its constituent *template method*.

<sup>2</sup> Note that the Strategy design pattern of Gamma et al. [4] has the same structure as the Composition Template Method, but a more specific intent. The Strategy is used to make whole algorithms easily interchangeable, whereas the Composition Template intends to let only the steps of an algorithm vary.



**Figure 3: Graphic user interface of SPOOL for Hot Spot recovery: visualization of Hot Spots in the source code model (window 1); extraction and inspection of Hot Spots (window 2); extraction of statistical data for further analyses (window 3).**

Figure 3 illustrates some features of the SPOOL environment applied to *ET++* [5], one of the C++ systems that we reverse engineered for the purpose of Hot Spot recovery. Window 1 shows the class hierarchy of the system being analyzed, layed out with *Dot* and overlaid with the recovered Hot Spots. Hot Spots are visualized by bounding boxes that are incrementally drawn around their so-called *reference class*, which can be selected at the discretion of the user; for Hot Spots based on the Inheritance Template Method (see Figure 1), the user can declare one of the classes *Client*, *TemplateClass*, or *HookClass* as the reference class, and for those based on the Composition Template Method (see Figure 2) similar options apply. A class that is the reference class (e.g. *HookClass*) in multiple Hot Spots is visualized with a bounding box for each of these Hot Spots. Thus, the size of the resultant outermost bounding box of a class is proportionate to the number of Hot Spots for which it is the reference class. Keeping the size of these bounding boxes constant during zooming leads to the visual effect that once the diagram is sufficiently zoomed out, the classes with

many implemented Hot Spots will protrude from the diagram.

Window 2 details the context information of a selected Hot Spot, by extracting it from the source code model into a separate diagram for further inspection. The upper part shows the list of Hot Spots for which the selected class in window 1 is the reference class. The middle part depicts the selected Hot Spot in form of an extended collaboration diagram [25] showing only its constituent parts, that is, the implementation of the *TemplateClass*, the *template methods*, the abstract *hook methods*, the *HookClasses*, and the concrete *hook methods*. The lower part shows these constituent classes of the same Hot Spot within the expanded class hierarchies of the system. This provides an overview of the context in which the Hot Spot is used.

Window 3 presents the statistics diagram, which allows for the extraction of Hot Spot occurrences. The content of this diagram can be copied in textual form into the clipboard of the computing device, from where it can be pasted into a spreadsheet or statistical package for further analyses.

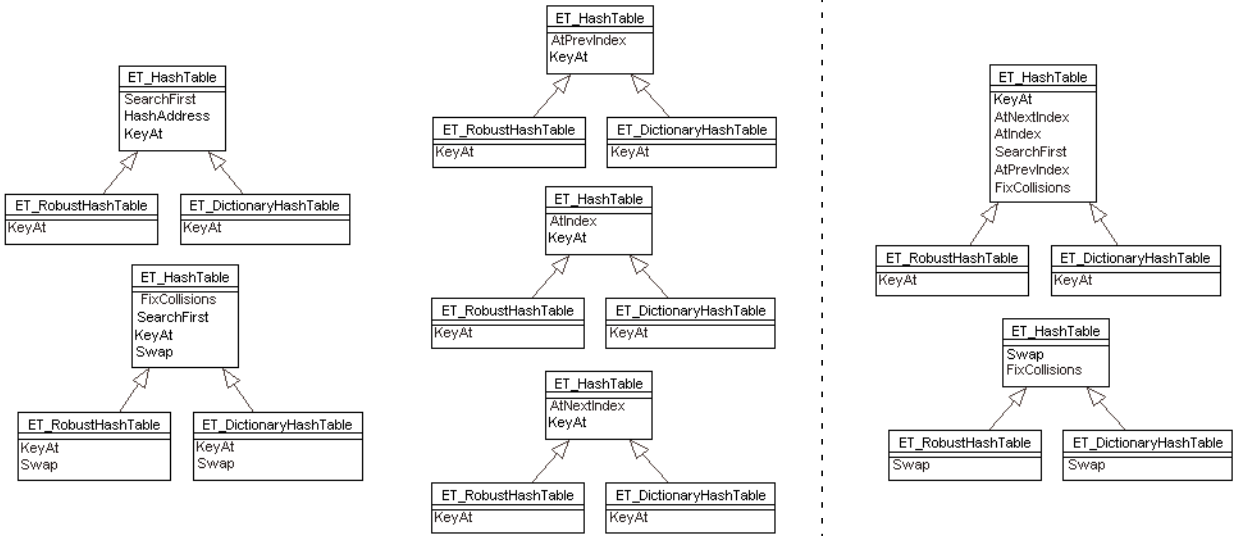


Figure 4: Template Method diagrams (left) versus Hot Spot diagrams (right).

### 3. Research Approach

Design analysis with the SPOOL environment is an iterative process driven by the interplay among automated analysis, visual presentation and transformation, and human interpretation; our research approach taken to the reverse engineering of Hot Spots reflects this process. In the following, we discuss the different steps of this research, which include the reverse engineering of source code, the definition of the notion of Hot Spots, the declaration of the research variables, the writing of queries on our source code repository that evaluate these research variables, and the final interpretation of the results.

The study started with an initial data gathering phase in which we reverse engineered the well-known application framework *ET++* [5], as delivered by the *SNiFF+ 3.0* software development environment [23], and *SystemX*<sup>3</sup>, a telecommunications system provided by Bell Canada. Both systems are implemented in C++, and Table 1 shows some of their size metrics. Information about the source code kept in our repository includes directories, files, aggregates (classes, structures, union), inheritance relationships, attributes, operations and methods, parameters, return types, operation calls, object instantiations, variable use, friendship relationships, and class and function templates. Refer to [11] for more detailed information:

The next step was to find a definition for the notion of Hot Spots. According to Pree [18], Hot Spots are those as-

<sup>3</sup> For confidentiality reasons, we are not allowed to name any identifier of *SystemX*.

	ET++	SystemX
Lines of code /	70,796	291,619
Lines of pure comments /	3,494	71,209
Blank lines	12,892	90,426
# of files (.C / .h)	485	1,153
# of classes	722	1,420
# of generalizations	466	941
# of methods	6255	8,594
# of attributes	4460	13,624
size of the system in the repository	19.3 MB	41.0 MB

Table 1: Size metrics of industrial systems.

pects of a design that have to be kept flexible for adaptation. Hence, to identify these flexible parts in existing source code, we have to focus on the hook methods. Using the *ET++* class *ET\_HashTable*, which is the *TemplateClass* of an Inheritance Template Method, Figure 4 illustrates the distinction between Template Methods and Hot Spots. The Template Method diagrams (Fig. 4, left part) put emphasis on the rigid template methods of a class and show the operations that they invoke together with the subclasses in which some of these operations are overridden. The top left diagram shows the template method *SearchFirst*, which calls the non-virtual method *HashAddress* and the overridden hook method *KeyAt*. The template methods *AtPrevIndex*, *AtIndex*, and *AtNextIndex* call the overridden method *KeyAt* only. *FixCollisions*, finally, calls *SearchFirst*, which is itself a template method, as well as the two overridden hook methods *KeyAt* and *Swap*. Hence, among the five Template

Methods *SearchFirst*, *FixCollisions*, *AtPrevIndex*, *AtIndex*, and *AtNextIndex*, the method *KeyAt* serves as the shared hook for extension. (Note that the tool distinguishes different kinds of methods using colored text.) On the other hand, the two Hot Spot diagrams (Fig. 4, right part) focus on the hook methods and associate their respective template methods. The top right diagram shows *KeyAt* as the hook method and puts all related template methods in its context. The bottom, right diagram shows the hook method *Swap*, which is invoked by *FixCollisions* only. For further explanation, Figure 5 illustrates the two Hot Spots of the above example in a more abstract form.

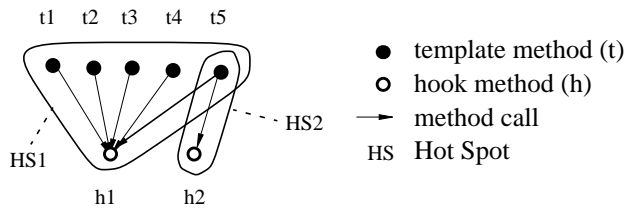


Figure 5: Hot Spot example.

Figure 6 extends the above example with a third hook method *h3*, which is called by the template method *t5*. Furthermore, it introduces method calls from *t4* to *h2* and *t4* to *h3*, which results in eight methods clustered into three Hot Spots, one for each hook method.

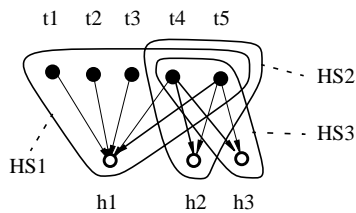


Figure 6: Extended Hot Spot example.

Based on the hook methods only, a definition for Hot Spot would lead to a rather fine-grained method clustering, as each hook method would define a separate Hot Spot. The challenge is to find a definition that leads to Hot Spots that are neither too fine-grained nor too course-grained, as either would weaken their expressiveness. In the example of Figure 6, the hook methods *h2* and *h3* are called by the same set of template methods (*t4* and *t5*), which suggests that they work together in different realms to perform some common task. Therefore, we capture those hook methods that are always called together into one single Hot Spot (Figure 7).

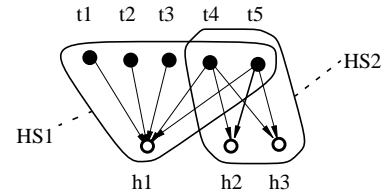


Figure 7: Hot Spot definition.

Whereas in Figure 6 a Hot Spot is defined for each hook method together with its associated template methods, in Figure 7 all the hook methods that are invoked by the same set of template methods are captured into one single Hot Spot. Browsing through the reverse engineered models of *ET++* and *SystemX*, we identified many instances where two or more hook methods were always called together. Accordingly, we came to define a Hot Spot as

*a set of hook methods and their associated template methods, in which each hook method is invoked by exactly the same set of template methods.*

This definition prohibits, for instance, putting *h1* into *HS2*, as *h1* is invoked by a superset of the template methods of *h2* and *h3*. Using the interactive capabilities of SPOOL, we investigated various alternative definitions for Hot Spot, but all of them led to either rather fine-grained or too coarse-grained Hot Spots. Therefore, we accepted the above definition as a compromise.

Based on the above definition of Hot Spot, we identified the research variables that are relevant for this study and developed queries on the SPOOL source code repository that would evaluate these variables in the reverse engineered source code of *ET++* and *SystemX*. The following list summarizes the research variables of the study:

- IHS (Inheritance Hot Spot) stands for a Hot Spot that is exclusively based on Inheritance Template Methods; that is, only template methods that invoke hook methods defined in the same class or inherited from a superclass are considered (see Figure 1).
- CHS (Composition Hot Spot) stands for a Hot Spot that is exclusively based on Composition Template Methods; that is, only template methods that invoke hook methods by delegation are considered (see Figure 2).
- HS stands for a Hot Spot that is based on either Inheritance or Composition Template Methods, or on both; that is, template methods that invoke hook methods defined in any class of the system are considered.

To quantify and explore the above list of research variables, we developed queries on the source code repository that recover instances of Hot Spots and extract each identified instance into a separate collaboration (see Figure 3,

window 2 for the visual representation). As an example, to recover Inheritance Hot Spots (IHS), we first recover all Inheritance Template Methods (see Figure 1). The associated query traverses all classes (*TemplateClass*), goes into each method (*templateMethod*), and verifies if the method calls a polymorphic operation (*hookMethod*) that is defined in the *TemplateClass* or any of its superclasses. Then, the query goes into the subclass hierarchy of the *TemplateClass* and looks up if *hookMethod* is overridden. During the traversal of the source code model, the query collects all relevant information and, if all conditions are met, passes them to a *Design Component Builder* object that creates a collaboration for the retrieved Inheritance Template Method. To recover the IHS, we then loop through all Inheritance Template Methods, gather those hook methods that are called by exactly the same set of template methods (according to our definition of Hot Spot), and create for the resulting set of hook and template method an IHS collaboration.

As a next step, we executed the queries and visualized the identified Hot Spot instances within the source code model (see Figure 3, window 1). The capability of the SPOOL environment to extract Hot Spots into separate diagrams (see Figure 3, window 2) was essential to verify the correctness of the queries and to gain an understanding of how Hot Spots were applied in the application at hand. Finally, we used the statistics diagram (see Figure 3, window 3) to extract for each class the number of Hot Spot instances that included this class as a reference class into a statistical analysis package (see Figure 12). We used *TemplateClass* as the reference class for Inheritance Hot Spots (see Figure 1) and *AbstractHookClass* as the reference class for Composition Hot Spots (see Figure 2), as these classes define the hook methods that are to be overridden in some subclass.

#### 4. Analysis of Hot Spots

The previous sections suggested that an effective software comprehension tool needs to integrate the human analyzer into the design recovery and analysis activities. We have not come across any automated approach that could make up for the expressiveness of visual design inspection carried out by the user to verify the validity of recovered design components and to gain a better understanding about the context in which they are implemented. It is our experience that all software engineering tasks, from comprehension to evolution, benefit vastly from a tool that allows for the interplay between automated design component recovery, visual presentation and inspection, and human interpretation. In the following we explain interactive inspection of Hot Spots in greater detail and interpret Hot Spot occurrences in *ET++* and *SystemX*.

#### 4.1. Interactive Hot Spot inspection

Running a query for one of the Hot Spot types results in a collection of collaborations, each describing an instance of the respective kind of Hot Spot (IHC, CHS, or HS). As a result, the tool automatically generates an important first view by overlaying the class diagrams with the recovered instances (Figure 8).

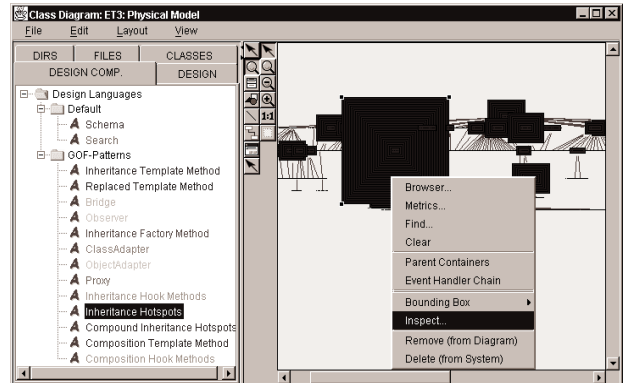


Figure 8: Inspection window invocation.

One purpose of this view is to provide the human analyzer with an initial overview of the number and the locations of the Hot Spots in the system at hand; as the reference classes for drawing the bounding boxes, we use those classes that implement the abstract hook methods of the Hot Spots. Another purpose is to provide the visual anchor points for the invocation of the so-called inspection window, which allows for the further exploration of the recovered Hot Spots (Figure 9).

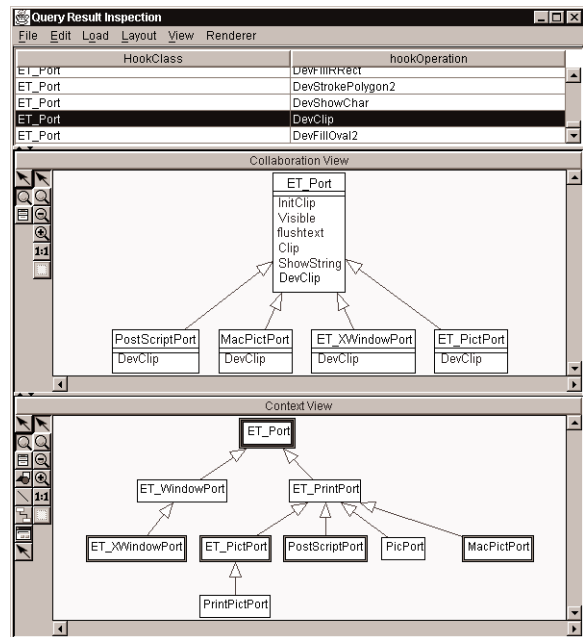


Figure 9: Hot Spot inspection in *ET++*.

The top part of this diagram lists the extracted Hot Spots of the class *ET\_Port*. The middle part shows one Hot Spot that is based on the hook method *DevClip* in form of an extended collaboration diagram [25]. To learn about the nature of this Hot Spot, that is to gain an understanding about the purpose of its template and hook methods, the collaboration diagram supports commands to blend in and out the context in which the selected class, template method, or hook method is used: if the analyzer selects a template or hook method, all related methods that call or are called by the selected one will be shown; if the human analyzer selects a class, all those classes that access or are accessed by the selected class (by method call, instantiation, or variable access) will appear. The bottom part depicts the classes in the collaboration diagram within the relevant class trees. It shows where the hook methods, in our example only *DevClip*, are overridden (that is, *ET\_XWindowPort*, *ET\_PictPort*, *PostScriptPort*, and *MacPictPort*) and, equally important, where the default implementation is used (*PicPort*). Such stepwise, human-controlled transformation of the information content of a Hot Spot diagram is essential for the analyzer to gather invaluable context information about how the Hot Spot is used in the application at hand. In the following, we give two examples that illustrate how interactive Hot Spot inspection can help understand, maintain, and evolve a system.

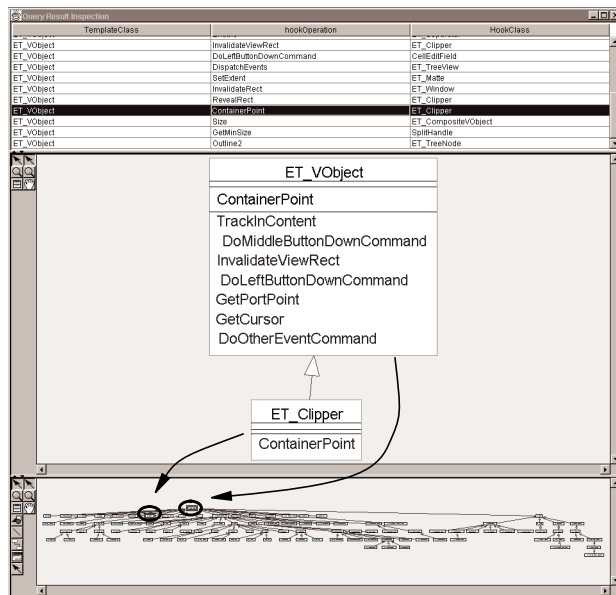


Figure 10: Inheritance Hot Spot.

Figure 10 shows an Inheritance Hot Spot that is based on only one hook method (*ContainerPoint*), which is called by seven template methods and overridden in one subclass (*ET\_Clipper*). This information, together with the location of the overridden hook method as indicated in the lower part

of the diagram, is crucial for the comprehension and the well-informed maintenance of the software system at hand. All too often, we have experienced that changes in the behavior of some template method of a framework broke the client code, as the framework maintainers were not aware of the particularities of the adaptations made in the overridden hook methods of the client code. The Hot Spot diagram locates all template methods on which a hook method is dependent. Thus, it can help identify the possible impact of a change in a template method or, vice versa, help explore the causes of a sudden malfunction of the client code after the installation of a new release of the framework. Surprisingly, none of the case tools we have worked with or evaluated [19] support this kind of design recovery and inspection.

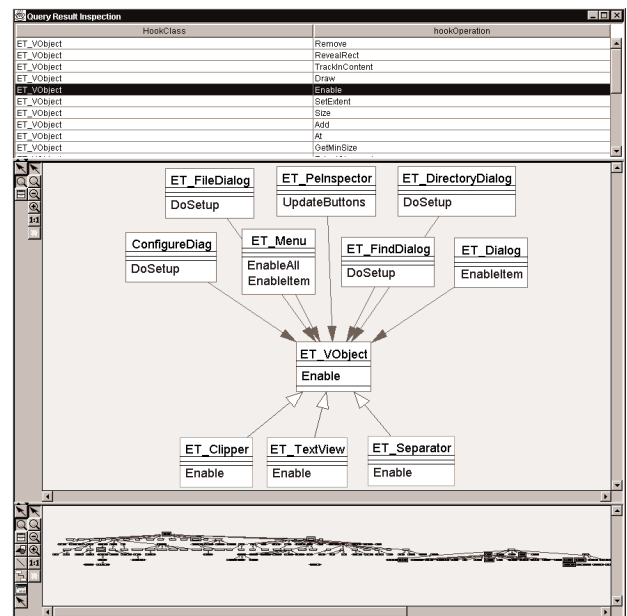


Figure 11: Composition Hot Spot.

Figure 11 presents a Composition Hot Spot that is based on the hook method *Enable* of *ET\_Object*. In the top two rows of the middle part, all template methods that call *Enable* are depicted. As the classes of these template methods can be far away from the hook class within the overall class hierarchies, a software design and maintenance tool that gathers the Hot Spot related information and presents them in one diagram greatly facilitates software comprehension. The strength of such a representation is that it puts together the different contexts in which a hook method is used.

#### 4.2. Analysis of Hot Spot occurrences

Using the statistics tool (see Figure 3, window 3), we summarized the occurrences for the different kinds of Hot Spots and exported them into a spreadsheet where we did all further analyses. Recall that for each class we calculate the

number of Hot Spots for which the class at hand implements an abstract hook method. Figure 12 lists the numbers for the 25 classes of *ET++* and *SystemX* that exhibit the most Hot Spots.

ET++				SystemX			
Class Name	HS	IHS	CHS	Class Name	HS	IHS	CHS
ET_Port	35	30	7	Class01	29	7	26
ET_VObject	30	20	21	Class02	24	10	18
ET_Text	22	7	21	Class03	23	7	19
ET_System	21	3	19	Class04	22	17	9
ET_SeqCollection	16	8	9	Class05	19	2	18
ET_WindowPort	16	16	1	Class06	18	12	9
ET_Object	13	9	4	Class07	14	6	10
ET_Manager	13	12	3	Class08	14	4	12
ET_HashTable	11	8	4	Class09	11	1	10
ET_View	9	5	6	Class10	10	1	9
ET_EvtHandler	8	2	7	Class11	10	6	4
ET_Data	8	3	5	Class12	10	7	6
ET_DevBitmap	8	5	7	Class13	9	7	3
ET_Printer	7	2	6	Class14	9	3	7
ET_Command	7	3	4	Class15	9	6	3
VisualMark	7	1	7	Class16	9	3	7
ET_WindowSystem	7	2	4	Class17	9	0	9
ColorMapper	7	0	7	Class18	8	8	2
ET_Container	5	5	4	Class19	8	1	8
ET_PttyConnection	5	0	5	Class20	7	5	3
ET_TypeMatcher	5	4	1	Class21	7	4	4
ET_Progress	5	4	1	Class22	6	4	2
ET_TreeView	4	1	3	Class23	6	2	4
ET_Converter	4	2	3	Class24	6	6	0
ET_TextView	4	5	3	Class25	5	1	4
<b>722 classes</b>	<b>361</b>	<b>223</b>	<b>201</b>	<b>1420 classes</b>	<b>497</b>	<b>265</b>	<b>310</b>

**Figure 12: Hot Spot occurrences in *ET++* (left) and *SystemX* (right; class names are changed for confidentiality).**

In the following, we address two aspects of Hot Spot driven source code analysis, which are supported by the above occurrences in *ET++* and *SystemX*: (a) the identification of the key abstractions in an object-oriented design; and (b) the identification of different implementation styles of these key abstractions.

#### Key design abstractions

Our project partner, the software quality assessment team of Bell Canada, analyzes software products of potential suppliers for maintainability. One of the first steps in the assessment process is to gain an understanding of the key abstractions of the design. To this end, we manually analyzed the responsibilities of those classes that are at the top of the lists displayed in Figure 12 and identified these classes as the key abstractions in both *ET++* and *SystemX*. For *ET++* the significant role of *ET\_Port*, *ET\_VObject*, or *ET\_Text*, to mention only the top three classes, is documented in various sources [1, 4, 5, 26]. As such documentation was not available for *SystemX*, we reviewed the class models using our SPOOL environment, and for detailed code inspection we used *SNiFF+* [23]. The results confirmed that those classes that exhibit the most Hot Spots are crucial for

effective software maintenance. Based on the experience we gathered when studying the undocumented design of *SystemX*, we argue that the visualization of the automatically recovered Hot Spots (see Figure 3, window 2) is an invaluable aid to gain a first understanding about the nature of the software system at hand. Such a view together with the Hot Spot inspection mechanisms allows the analyzer to build up incrementally the knowledge that he or she needs to maintain the system.

#### Hot Spot implementation styles

Besides knowledge about the location of the key abstractions of a software system, an understanding of how these abstractions are to be adapted, extended, and maintained is equally important. For maintenance it is crucial to know if a class provides its responsibilities, for instance, by delegating functionality to other classes, by implementing it within the class, or by providing internal hooks that need to be overridden. Figure 12 uncovers the different styles in which the key abstractions implement flexibility for adaptation; they tend to be oriented towards either inheritance (IHS) or composition (CHS), but hardly exhibit a large number of both kinds of Hot Spots. Only one class with more than ten Hot Spots, *ET\_VObject* of *ET++*, comprises many Inheritance Hot Spots as well as many Composition Hot Spots. Such knowledge about how the Hot Spots of the key classes are implemented, with inheritance or composition, is critical to maintain the original design intent and implementation style of the designers that created these classes.

## 5. Related work

Below, we briefly review a number of studies that address the concept of Hot Spots from the viewpoints of forward engineering, redocumentation, and design recovery.

Johnson addresses the techniques that support the creation of reusable classes and frameworks [7, 8, 9]. He elaborates on the dangers of class inheritance and advocates that only the well-thought combination of class inheritance and class composition can lead to reuse in-the-large. These early works inspired the well-known design pattern catalogues of Gamma et al. [4], Buschman et al. [1], and Schmidt [22], who discuss some of the key building blocks for reusable object-oriented design. We argue that it is these patterns that break apart the stable parts of an application from those parts whose functionality can hardly be anticipated. For these parts, which need to be kept flexible to accommodate reuse and adaptability, Pree introduced the notion of Hot Spots [18].

Several authors have discussed the advantages of documenting object-oriented software with patterns. Johnson articulates the strengths of pattern-based design

documentation: “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” [7]. To this end, Schauer and Keller [20] report on tool support for interactive and automated redocumentation of object-oriented designs with patterns. To make such documentation tools more effective for large-scale, object-oriented frameworks, Odenthal and Quibeldey-Cirkel [16] suggest a standard format for object-oriented framework documentation, which includes among other items a clear description of the Hot Spots. Many other authors have argued similarly, but the state-of-practice in object-oriented design documentation seems to ignore them. Most frameworks we have encountered were insufficiently documented and none of them explicitly stated its Hot Spots for reuse and adaptation.

We have identified only one study that addresses design recovery based on Hot Spots. Demeyer [2] reports on their experiments with *HotDraw*, a Smalltalk framework for the construction of graphic editors. In contrast to our definition of Hot Spot, they use the rigid parts of Template Methods to infer Hot Spots. In our work, we look at the flexible parts of a Template Method, the hook methods, which can significantly reduce the noise that is reported by Demeyer. Furthermore, the clustering of the hook methods that are always called together in the context of different template methods helps identify Hot Spots of a larger scale. Demeyer reports that they found Hot Spots that were not documented and, vice versa, some of the Hot Spots listed in the documentation could not be reconstructed. He argues that this was due to the outdated documentation of *HotDraw*. Demeyer does not provide any numbers about occurrences, nor does he report on visual inspection capabilities of the tool which they applied to recover Hot Spots. We consider interactive, human-controlled analysis in addition to automated recovery support for Hot Spots indispensable to gain a broader picture of the context in which they are embedded.

## 6. Conclusion and future work

Hot Spots capture the flexible parts of an application domain and need to be clearly identified and well-understood to warrant the guided evolution of the software system at hand. Very often, however, these flexible parts are undocumented and, as a consequence, application programmers disregard or misuse them when extending or adapting an object-oriented design for domain-specific requirements. In this paper, we addressed this blurring of Hot Spots during software evolution by providing a technique and tool support for their recovery within C++ source code. We defined the notion of Hot Spots and applied our design visualization technique of growing bounding boxes to gain an overview

of the key providers of framework flexibility based on Hot Spots. We described the design inspection tool that we implemented for the extraction of Hot Spots into separate diagrams and summarized the occurrences of Hot Spots in *ET++* and *SystemX*. We inferred that Hot Spots can be used to identify the key design abstractions as well as their implementation style, based on inheritance or composition.

Besides continuing the development of the SPOOL environment for reverse engineering, and in particular pattern-based recovery of design components [11], we plan to work in three areas that are directly related to this study. First, we will be implementing tool support that allows for the creation of *bridges* between Hot Spots to navigate easily from one occurrence to another. Second, we will analyze several large-scale application frameworks as well as telecommunications software provided by Bell Canada, with the goal to quantify, compare, and correlate the use of Hot Spots in different object topologies, that is architectural styles, frameworks, toolkits, and application-specific designs [24]. This should eventually result in a suite of metrics for Hot Spots in object-oriented designs, considering domain-specific application characteristics and sufficiently validated with empirical data. Third, we will study Hot Spots in respect to design patterns [4], architectural patterns [1], and domain-specific telecommunications patterns [22]. We expect that this will lead to improved understanding of how design patterns support reusability, adaptability, and changeability in the specific applications of Bell Canada.

## 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: *Bell Canada* for the source code parser *Datrix*, *Lucent Technologies* for the C++ source code analyzer *GEN++* and the layout generators *Dot* and *Neato*, *Instantiations* for the graphic editor framework *jKit/GO*, *Poet* for the object-oriented database management system *Poet 5.1*, and *TakeFive Software* for the *SNiFF+* software development environment.

## References

- [1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons, 1996.
- [2] S. Demeyer. Analysis of Overridden Methods to Infer Hot Spots. *Object Oriented Technology (ECOOP'98 Workshop Reader)*, Serge Demeyer and Jan Bosch (Ed.), LNCS 1543, Springer Verlag, 1998.
- [3] P. T. Devanbu. GENOA - A Customizable, Language- and Front-End Independent Code Analyzer. In *Proc. of the 14th*

- International Conference of Software Engineering*, Melbourne, Australia, pages 307-317, 1992.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [5] E. Gamma and A. Weinand. *ET++: A Portable C++ Class Library for a UNIX environment*. Union Bank of Switzerland. Workshop at OOPSLA'90, Ottawa, Canada, 1990.
- [6] jKit/GO. Online documentation. Instantiations, Tualatin, OR. On-line at <<http://www.instantiations.com/>>.
- [7] R. Johnson. Documenting Frameworks with Patterns. In *OOPSLA'92, Sigplan Notices*, Vol. 27, No. 10, pages 63-76, October 1992.
- [8] R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, Vol. 1, No. 2, pages 22-35, June/July 1988.
- [9] R. Johnson and W. Opdyke. Refactoring and Aggregation. In *Proceedings of ISOTAS '93 (LNCS 742)*, Springer-Verlag, pages 264-278, 1993.
- [10] R. K. Keller and R. Schauer. Design Components: Towards Software Composition at the Design Level. In *Proc. of the 20th International Conference on Software Engineering*, Kyoto, Japan, pages 302-310, April 1998.
- [11] R. K. Keller, R. Schauer, S. Robitaille, and Patrick Pagé. Pattern-Based Reverse-Engineering of Design Components. In *Proc. of the 21th International Conference on Software Engineering*, Los Angeles, CA, pages 226-235, May 1999.
- [12] E. Kontsofios and S. C. North. Drawing graphs with Dot. AT&T Bell Laboratories, Murray Hill, NJ. On-line at <<http://www.research.att.com/sw/tools/graphviz/>>.
- [13] C. Leduc and D. Proulx. *DATRIX Abstract Semantic Graph - Reference Manual*. Bell Canada, January 1999. On-line at <<http://www.iro.umontreal.ca/labs/gelo/datrix>>.
- [14] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
- [15] S. C. North. Neato User's Manual. AT&T Bell Laboratories, Murray Hill, NJ. On-line at <<http://www.research.att.com/sw/tools/graphviz/>>.
- [16] G. Odenthal and K. Quibeldey-Cirkel. Using Patterns for Design and Documentation. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, Jyväskylä, Finland, pages 511-529, June 1997.
- [17] POET Java ODMG Binding. Online documentation. Poet Software Corporation, San Mateo, CA. On-line at <<http://www.poet.com/>>.
- [18] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesely, 1994.
- [19] S. Robitaille. *Evaluation of CASE Tools for Teaching Software Engineering*. Technical Report GELO-95, University of Montreal, Quebec, Canada, February 1999. in French.
- [20] R. Schauer and R. K. Keller. Pattern Visualization for Software Comprehension. In *6th International Workshop on Program Comprehension*, Ischia, Italy, pages 4-12, June 1998.
- [21] H. A. Schmid. Systematic Framework Design by Generalization. In *Communications of the ACM*, Vol. 40, No. 10, pages 48-51, October 1997.
- [22] D. Schmidt. *Design patterns for concurrent, parallel, and distributed systems*. Online at <<http://siesta.cs.wustl.edu/~schmidt/patterns-ace.html>>.
- [23] SNiFF+. *Documentation set*. Takefive Software, Salzburg, Austria. On-line at <<http://www.takefive.com/>>.
- [24] W. M. Tepfenhart and J. J Cusick. A Unified Object Topology. *IEEE Software*, Vol. 14, No. 1, pages 32-35, 1997.
- [25] UML. *Documentation set version 1.1*. September 1997. On-line at <<http://www.rational.com/>>.
- [26] A. Weinand. *Objektorientierter Entwurf und Implementierung portabler Fensterumgebungen am Beispiel des Application-Frameworks ET++*. Doctoral Thesis, University of Zurich, 1991. in German.
- [27] R. J. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.