Three Reverse Engineering Patterns

Serge Demeyer, Matthias Rieger, Sander Tichelaar
Software Composition Group, Universität Bern
{demeyer,rieger,tichel}@iam.unibe.ch
http://www.iam.unibe.ch/∼scg/

April 30, 1998

Abstract

Whereas a design pattern describes and discusses a solution to a design problem, a *reverse engineering pattern* describes how to understand aspects of an object-oriented design and how to identify problems in that design. In the context of a project developing a methodology for reengineering object-oriented legacy systems into frameworks, we are working on a pattern language for reengineering. This paper presents three samples of that pattern language, all dealing with reverse engineering.[1]

1 Introduction

The ability to reengineer object-oriented legacy systems has become a vital matter in today's software industry. Early adopters of the object-oriented programming paradigm are now facing the problem of transforming their object-oriented ”legacy” systems into full-fledged frameworks. Dealing with programs exceeding 10,000 lines of poorly documented code definitely requires support from tools as well as methodologies.

In the context of the FAMOOS project (http://www.iam.unibe.ch/∼famoos/), we are working on a reengineering handbook consisting of good reengineering practices. During our visits at the industrial sites, we noticed that quite a few reengineers appreciated the GOF-handbook [GHJV95] and applied it in their day-to-day working habits. Part of the appreciation was due to the problem/solution form of patterns, as it helped reengineers to quickly assess the applicability of a given design pattern. Thus, we decided to adopt what we call *reengineering patterns* in our handbook, hoping that it would help reengineers to quickly identify which technique might be applicable on the code they are reengineering. Another thing we learned from our industrial site visits is that tool support is crucial in detecting the what to reengineer: without proper tool support it is likely that a piece of legacy code remains untouched until it is thrown away. Thus –and this is in contrast with other pattern catalogues– reengineering patterns emphasise more on tools.

What we present here are three sample patterns of our forthcoming reengineering handbook. Except for the fact that they all deal with ”reverse engineering”, they have not so much in common. Yet, they are offering solutions for problems encountered by our industrial partners. Namely, **Detectiong Duplicated Code** describes a particularly appealing approach to detect copy/paste code by means of dot plots, **Architectural Extraction using Prototyping** reports on our experiences with prototyping as a way to extract architectural information, while **Inferring Hot Spots from Overridden Methods** explains how to reverse engineer where a framework can be extended. Together with other patterns we are currently working on, they will become part of a pattern language for object-oriented reengineering.

[1] Note that some colleagues of ours have submitted another document discussing patterns for the related problem of *reengineering*. This other submission is entitled ”Type-Check Elimination: Two Reengineering Patterns”.

2 Code Duplication Detection

Intent

Support the detection of duplicated code using a matrix display of duplicated parts of source code.

Applicability

Duplicated code should be detected and dealt with as a general quality assurance measure during the problem detection phase of a re-engineering project. The only prerequisite is the availability of the source code.

Motivation

The duplication of code occurs frequently during the development phase (there are a number of reasons for it we will not discuss here). Since it is an *ad hoc*/copy&paste activity more than something that is planned for, occurrences of duplication are not documented and have to be detected.

Process

Since any part of source code can be copied, we cannot search for specific program clichés but rather have to deduce them from the input itself by comparing every line with every other line of the system under scrutiny. This comparison produces an enormous amount of two-dimensional data, i.e. a matrix where each cell stands for a comparison between two lines of code and contains a value unless the two lines did not match. The next step is then to quickly find zones of interest in the matrix. The matrix can be examined either using statistical means or visual exploration:

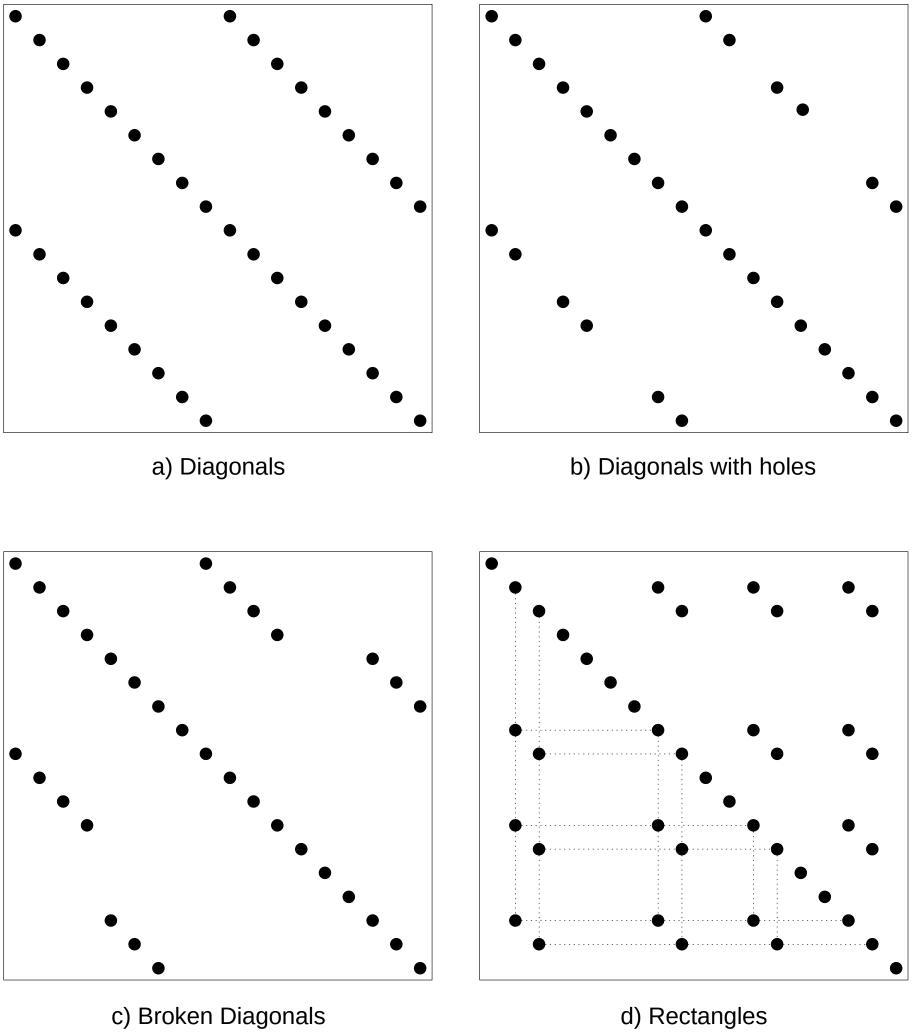
- **Automatic Examination:** A tool searches for some known configurations of dots in the matrix and delivers a report of the instances that were found.

  This approach is efficient in that it finds interesting spots automatically which is convenient for large amounts of data. However, the dot configurations to look for must be programmed beforehand. The objects of the automated search can range from simple diagonals with holes like in the example *b)* of Figure 1 to dot clusters found using statistical methods.

- **Visual Exploration:** Using a tool that displays the comparison matrices graphically, the reverse engineer browses through the matrix, zooms in on interesting spots, and, by clicking on the dots, examines the source code that belongs to a certain match. This approach is very much exploratory and can lead to unexpected findings (see the examples in [Hel95]).

Some example configurations formed by the dots in the matrices are the following:

- diagonals of dots indicate copied sequences of source code (see example *a)* in Figure 1).
- sequences that have holes in them indicate that a portion of a copied sequences has been changed (see example *b)* in Figure 1).
- broken sequences with lower parts shifted indicate that a new portion of code has been inserted (see example *c)* in Figure 1)
- rectangular configurations indicate periodic occurrences of the same code. An example is the `break` at the end of the individual cases in a C/C++ switch statement or recurring preprocessor commands like #ifdef SOME_CONDITION (see example *d)* in Figure 1).



a) Diagonals

b) Diagonals with holes

c) Broken Diagonals

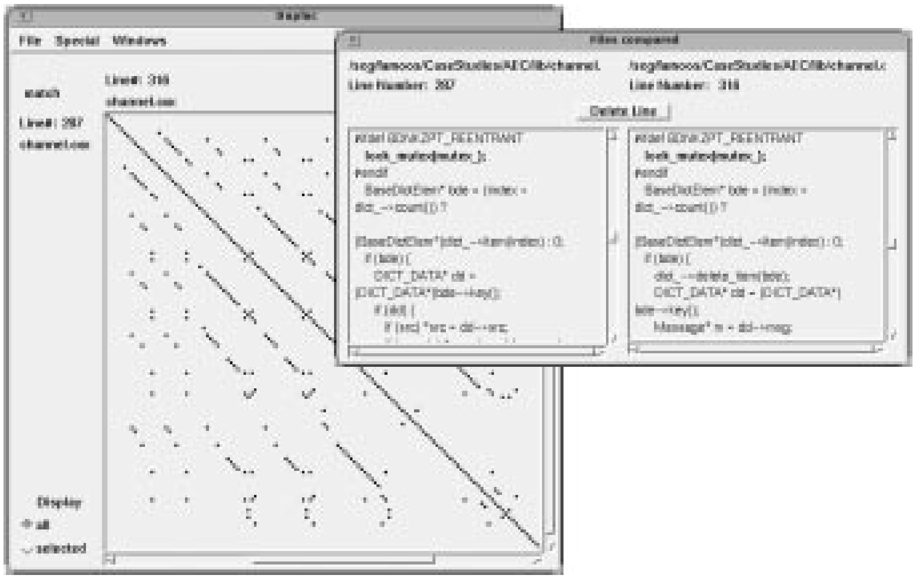
d) Rectangles

Figure 1: Different Configurations of Dots.

Recipe. The applicability of the recipe is based on the availability of a tool for duplication detection.

1. Start with an automatic search for certain configurations of dots. The tool creates a report on all configurations that were found in the matrices. These configurations have a well known interpretation and lead the reverse engineer to the problem areas with an idea of what the problem might be.
2. More general statistical data, like number of matched lines for a particular file-to-file-comparison could be a hint to examine the comparison matrix more closely through a visual representation.
3. A scatter-plot of the whole matrix gives an idea of the overall density of duplication, and how it is distributed. Using a zoom function, the reverse engineer can more closely examine areas that seem interesting.
4. Once the user has zoomed in enough that individual dots can be discerned, the user can, by pointing and clicking, look at the corresponding source code and see, what exactly has been duplicated.

Possible Difficulties and Limitations. Using this approach, one cannot detect duplicated *functionality*.



/sog/famoos/CaseStudies/AEC/lib/channel.c
Line Number: 297

```
#ifdef GDNK2PT_REENTRANT
  lock_mutex(mutex_);
#endif
  BaseDictElem* bde = (index <
dict_->count()) ?
(BaseDictElem*)(dict_->item(index)) : 0;
  if (bde) {
    DICT_DATA* dd =
(DICT_DATA*)(bde->key());
    if (dd) {
      if (src) *src = dd->src;
```

/sog/famoos/CaseStudies/AEC/lib/channel.c
Line Number: 316

```
#ifdef GDNK2PT_REENTRANT
  lock_mutex(mutex_);
#endif
  BaseDictElem* bde = (index <
dict_->count()) ?
(BaseDictElem*)(dict_->item(index)) : 0;
  if (bde) {
    dict_->delete_item(bde);
    DICT_DATA* dd = (DICT_DATA*)
bde->key();
    Message* m = dd->msg;
```

Figure 2: Screenshot of the DUPLOC tool

Language Specific Issues. Language dependency can be contained in the lexer that transforms the source code in the form that is compared by the tool. Depending on the format the comparison algorithm needs the source code to be transformed in, the lexer can be of variable complexity, which correllates directly with language dependency. For example, comparing the source code as text (i.e. after having removed comments and superfluous white space) only needs a very simple lexer, which keeps language dependency at a low level. Comparing the source code in token format requires a language specific lexer, however.

Tool Support

Tool support is vital for applying this pattern.

- We have implemented a SMALLTALK tool called DUPLOC (see Figure 2), which is specifically aimed at supporting a visual approach of code duplication detection. At the moment, the tool uses textual comparisons only. It allows the user to compare source code file by file, enabling to examine the source code by clicking on the dots. Noise filtering can be done by removing uninteresting lines.
- DUP [Bak92] is a tool that detects parameterized matches and generates reports on the found matches. It can also generate scatter-plots of found matches.
- DOTPLOT [Hel95] is a tool for displaying large scatter-plots. It has been used to compare source code, but also filenames and literary texts.

The approach we have taken with DUPLOC has the following advantages:

- It is lightweight: it does not use complicated algorithms like elaborate parsing techniques, and it has an intuitive interface which users learn quickly.
- It is language independent: Since we use textual comparison, the tool is language independent to a high degree and can be used for C, C++ and Java without a change.
- It is visual: the human eye is built to detect configurations and this can be fully exploited with a visual representation.
- It supports an exploratory approach to code duplication detection. Since not all configurations of code duplication are known beforehand, an automated configuration match will always overlook possibly meaningful configurations.

Discussion

The algorithm used to compare the source lines determines what level of fuzziness is allowed for a match to be counted as one. The simplest algorithm which compares the source lines character per character finds only exact matches. More complicated algorithms (see for example [Bak95]) can find *parameterized matches*. Parameterized matches point out the possiblity to refactor code into a function, where exact matches emphasize more the repetitive structures in the source code.

3 Architectural Extraction using Prototyping

Intent

Extract the architecture of a legacy system through the process of building a prototype. This process helps in understanding and documenting the original system and provides a testbed for future reengineering.

Applicability

This pattern is applicable when a lightweight means of extracting the architecture of a system is not available. Building a prototype is a lot of work, because it is building yet another system. This effort can be minimized by wrapping well-designed parts of the original system and the parts of the system that are not targeted for reengineering.

Symptoms. This pattern is applicable when a combination of the following symptoms occurs:

- limited knowledge of the system's architecture
- it is not clear how the functionality is implemented by the legacy system (different kinds of functionality may be mixed).
- lack of useful documentation and comments in the code
- difficult access to the original developers

Reengineering Goals. The goal is to gain insight into how the software works (i.e. what is the architecture of the system).

Motivation

Consider a system that has poorly documented code and no other useful documentation. Through the years all kinds of features have been added and changed possibly resulting in a tangled mix of features. The idea is to build a prototype using any available information and focusing on the areas of the software that need to be reengineered (see figure 3). This prototype reflects the functionality of the legacy system and helps to extract knowledge about the legacy system in different ways:

- Building the prototype means that certain problems need to be understood to get the prototype running. This leads to understanding the legacy system, because the same problems must already have been solved.
- The prototype and the problems encountered while building the prototype can be used as a starting point for discussions with the (other) developers. In this way knowledge that developers have about a system will come out, because the prototype and its problems will serve as a catalyst to help them share their knowledge about the original system.

- The prototype design is used to document the legacy system by creating a mapping between the prototype and the legacy system. We show this in figure 4: the different parts of the prototype are mapped to the source code and design information of the legacy system. This helps by revealing which part of the legacy system implements the different functionality. Furthermore, by evaluating the prototype and the mapping, assumptions about the legacy system may be confirmed or rejected, thus improving the understanding of the legacy system.

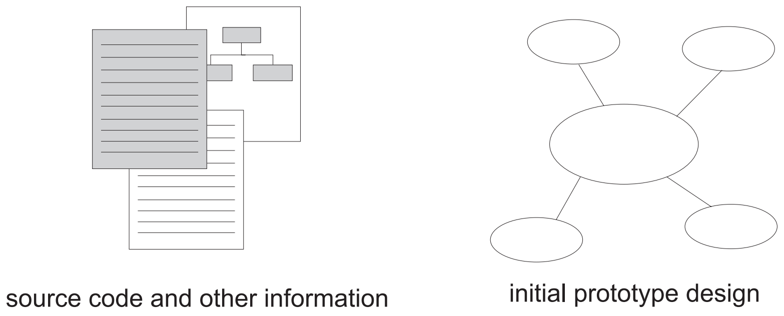


Figure 3: Initial situation

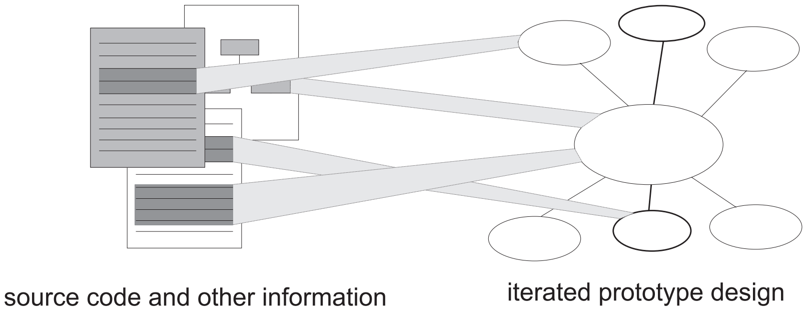


Figure 4: Situation after some iterations: the prototype has evolved and there exists a mapping to the legacy system

Process

We present the process as a recipe. The process is iterative: the knowledge of the legacy system will incrementally grow while the prototype is being evaluated and discussed with the original developers, and the documentation about the legacy system and the prototype itself are accordingly updated.

1. start to identify the aspects of the system that are relevant for the reengineering process. These will mainly be the aspects that are targeted for reengineering and the basic aspects of the original system that have to be taken into account to get the prototype to work.

2. create an initial design based on the available documentation and when possible discussions with the developers.

3. start building a prototype according to the initial design.

4. have a running prototype at all stages. This forces you to solve all the problems up to a certain point to be able to run the prototype. In this way you are automatically confronted with known and unknown problems the original system solves and the prototype should solve: knowledge is extracted.

5. compare the design with the legacy code and discuss with the developers about the design of the prototype. In this way you can discover design decisions and features of the system that are either unknown or undocumented.
6. alter the prototype and the inconsistent documentation to conform to your growing knowledge of the system. Next to this, update the map between the prototype and the original system.
7. iteration step: goto point 4....

Difficulties. Because the prototype is a software system itself, the method may fail on a poorly designed prototype. It is important to focus on the goals of the reengineering project and to document well.

Discussion

Advantages:

- It helps extracting the architecture and thus documenting the original system.
- It allows to focus on what to reengineer.
- It forces an awareness of the problems that are not clearly articulated by the documentation or in discussions with the developers.
- The prototype can serve as a test system for reengineered designs and future enhancements.

Disadvantages:

- *heavy-weight*: a working prototype may require a lot of effort.
- *ad-hoc*: model capture, problem detection, code analysis are mixed in one big process. The method is not very concrete, making it hard to define the exact process and to predict concrete results for a particular system.
- reverse engineering and forward engineering are *easily mixed*. It is probably tempting to focus on a good design of the prototype. Although this may be not bad in some situations, one can loose the focus on the main goal of the prototype, namely extracting the architecture. A developer will need the mapping from the legacy system to the prototype in order to be able to reengineer the original system in the end.

Tool Support

Everything that may help analyzing the legacy system (metrics, visualization) plus tools that help mapping the system to the prototype. Tools already available are a visualization tool for architectural extraction from Jerding and Rugaber[JR97] and the Reflection Model Tool of Gail Murphy[MN97].

Known Applications

We have tested this approach on a (remote) control system of mail sorting machines for one of the FAMOOS industrial partners. This user interface system consists of 350 KLOC and it should be reengineered to provide for a more flexible and scalable communication layer. The system is highly complex due to the flexibility of user interfaces and the intrinsic communication structure. The communication structure now runs into performance problems while being scaled up to cover more communication nodes. It should therefore be extensively adapted. This change is hard to achieve because of the complete lack of documentation and technical comments in the code and the fact that the primary developer has left the company.

The prototype clearly helped to extract the architecture of the system due to discussions with the remaining developers. Assumptions of the prototype developers made in the beginning of the reengineering process could be refined and discussions about the prototype provoked the developers of the original system to share their knowledge of the architecture and its features.

4 Inferring Hot Spots from Overridden Methods

Intent

Find potential hot spots by inspecting overridden methods. A hot spot represents a variation in the application domain within an object-oriented framework design. Detection of hot spots is necessary to understand and document object-oriented frameworks.

Applicability

You must have full access to the source code and you need a tool that allows you to inspect class hierarchies (especially overriding of methods) and query method invocations.

Symptoms. You have a well designed but undocumented object-oriented program and

- you want to learn where you should add a subclass
- you want to know which methods you should override

Reengineering Goals. Apply this pattern when you want to extend an object-oriented program which is probably well designed but undocumented. The pattern will help you to understand

- what parts of an object-oriented program are designed to be extended (i.e., where are the hot spots)
- how you are supposed to extend them (i.e., how should you fill in the hot spot)

Motivation

Object-oriented languages provide two mechanisms for defining and filling in hot spots: *class inheritance* and *object composition*. Table 1 compares the two mechanisms.

	Class Inheritance	**Object Composition**
Technique	Subclassing + Method Overriding	Late binding polymorphism
Encapsulation	”White Box” (has access to class internals)	”Black box” (has access to external interface only)

Table 1: Comparison of class inheritance vs. object composition hot spot

To illustrate both types of hot spots, we use an example of a framework for on-the-fly generation of HTML pages. The invariant part of the design states that it is possible to generate an HTML page based on information stored within a database. One variation in the design is what database maintains the information that will be used to render the HTML; our example implements this via a class inheritance hot spot. Another variation is what HTML version is accepted by the requesting web browser; this is implemented via an object composition hot spot.

Class Inheritance Hot Spot. Figure 5 shows the class diagram for a class inheritance hot spot, realised via the template method design pattern [GHJV95]. The invariant part of the design is implemented via the method generateHTML defined on the class Database. In the design pattern this method plays the role of the *template method*, invoking the abstract method fetchTable defined on the same class. This abstract method specifies the interface of the so-called *hook method*; by plugging in a different hook method one can implement a variation for that design. However, the hook method must respect its contractual obligations, like defined by the interface. In the example the hook method must return an instance of the class Table holding all of the records to be included into the generated HTML page based on the value of some string argument.

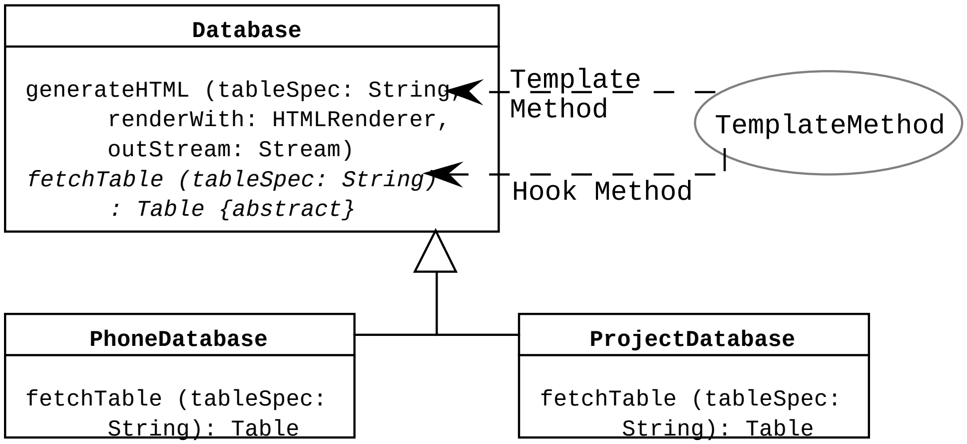


Figure 5: Example of a class inheritance hot spot. The template method and hook method are defined on the same class.

To fill in the hot spot, you must subclass Database, overriding the abstract hook method fetchTable. In the example, we include the subclasses PhoneDatabase and ProjectDatabase, each one providing a different implementation for the fetchTable method.

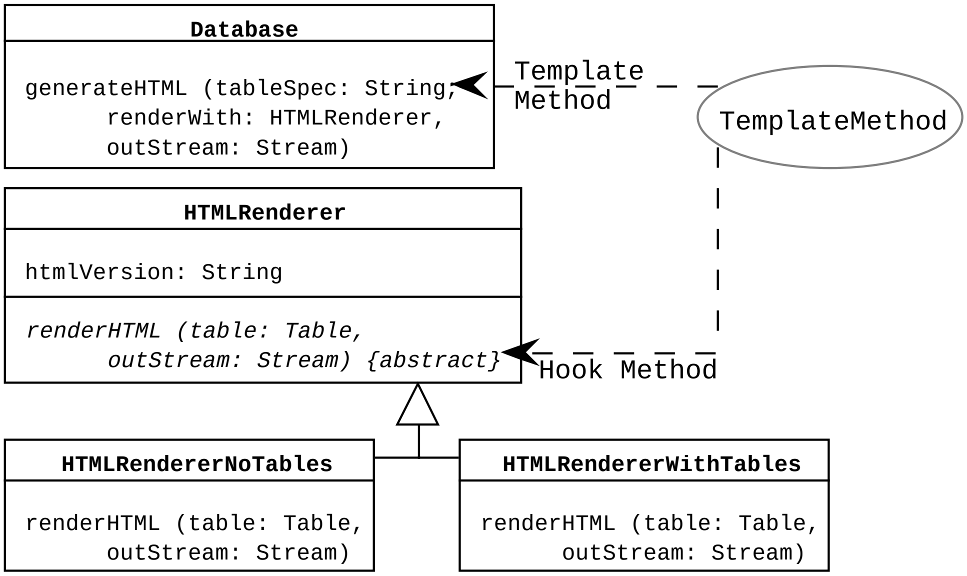


Figure 6: Example of an object composition hot spot. The template method and hook method are defined on a different class.

Object Composition Hot Spot. Figure 6 shows the class diagram for an object composition hot spot. Here as well, the invariant part of the design is defined in the template method generateHTML defined on the class Database. However, now the variant part follows from the invocation of a hook method renderHTML defined on *another* class named HTMLRenderer.[2]

To fill in the hot spot, you must do a bit more work than in the case of a class inheritance hot spot. Not

[2] Note that the pure form of the template method [GHJV95] design pattern does not discuss invoking a hook method defined on another class. [Pre95] however, discusses other configurations of template and hook methods.

only do you have to subclass HTMLRenderer, providing an implementation for the hook method renderHTML (we provide the subclasses HTMLRendererNoTables and HTMLRendererWithTables in the example). You also have to provide the appropriate association between an instance of class Database and an instance of the class HTMLRenderer (in the example this is achieved by argument passing: an instance of class HTMLRenderer must be passed as an argument when invoking the method generateHTML).

With the above example in mind, we see that it is possible to detect hot spots by looking for overridden methods. Overridden methods are symptoms for variations in the design, while the methods that invoke such methods correspond with the invariant aspects of a design.

Structure

Structure of a Class Inheritance Hot Spot

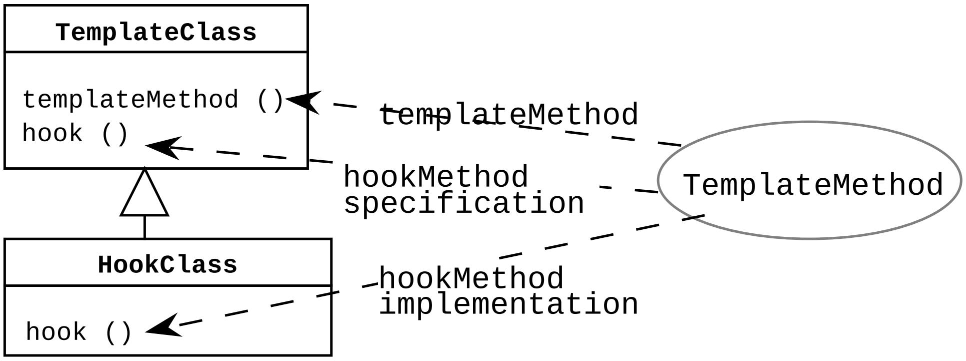


Figure 7: Generic structure for a class inheritance hot spot.

Participants.

- **TemplateClass** (Database)
  - Defines a template method which invokes the hook method.
  - Specifies the interface of the hook method (sometimes via an abstract method, sometimes via a method providing default behaviour).
- **HookClass** (PhoneDatabase, ProjectDatabase)
  - Is a subclass of (or the same class as) TemplateClass.
  - Implements the hook method, respecting its interface but providing the extended behaviour. This is achieved via method overriding, sometimes involving recursive invocations of the hook method of the superclass.

Structure of an Object Composition Hot Spot

Participants.

- **TemplateClass** (Database)
  - Defines a template method which invokes the hook method.

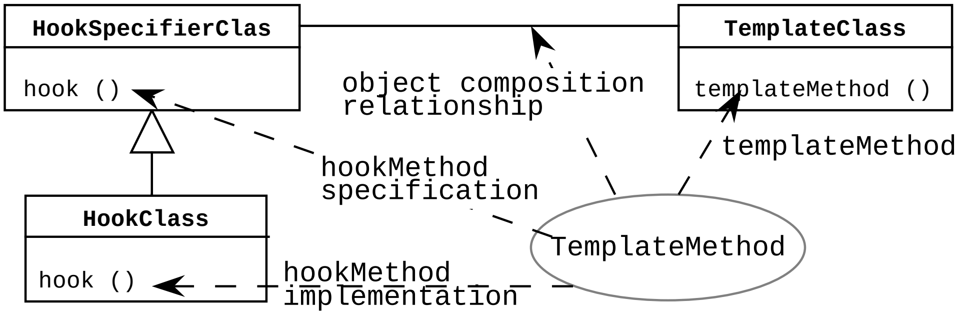


Figure 8: Generic structure for an object composition hot spot.

  - Implements an object composition relationship with the HookSpecifierClass. This may be achieved via (a) an instance variable refering to an instance of HookSpecifierClass; (b) a parameter of the template method refering to an instance of HookSpecifierClass; (c) a global variable refering to an instance of HookSpecifierClass; (d) a function invocation returning an instance of HookSpecifierClass.

- **HookSpecifierClass** (HTMLRenderer)
  - Specifies the interface of the hook method (sometimes via an abstract method, sometimes via a method providing default behaviour).

- **HookClass** (HTMLRendererNoTables, HTMLRendereWithTables)
  - Is a subclass of (or the same class as) HookSpecifierClass.
  - Implements the hook method, respecting its interface but providing the extended behaviour. This is achieved via method overriding, sometimes involving recursive invocations of the hook method of the superclass.

Process

To detect hot spots, you start by looking for a method overriding another method. Such a method is a likely candidate for serving as a hook method. To assess whether it really is a hook method you must find all methods invoking the candidate hook method and analyse their relationship. This analysis may lead to three conclusions: (i) it is not a hot spot: just ignore what you have seen; (ii) it is a class inheritance hot spot: document the involved methods as participating in a hot spot, marking that the hot spot should be filled in by creating a subclass; (iii) it is an object composition hot spot: document the involved methods as participating in a hot spot, marking that the hot spot should be filled in by creating a subclass and creating the appropriate object association.
The detailed steps involved in this process are explained in the recipe below; figure 9 provides a quick overview of the same recipe.

1. Search for a method overriding at least one other method. Call that method candidate_hook_method.
2. Enumerate all the methods that are overridden by candidate_hook_method; call the one defined on the most distant superclass the candidate_hook_specification and its class the candidate_HookSpecifierClass.
3. Assemble a list of all methods invoking candidate_hook_specification and strip all recursive invocations from the list (i.e., the ’super’ invocations). Call each item in the list candidate_template_method. If the list is empty then the candidate_hook_method did not qualify as a hook method and you can proceed with the next candidate_hook_method.[3]

[3] If the candidate_hook_specification is never invoked it is likely that you have detected a piece of dead code. Check to make sure that you have access to all of the source code before taking any further actions to remedy this.

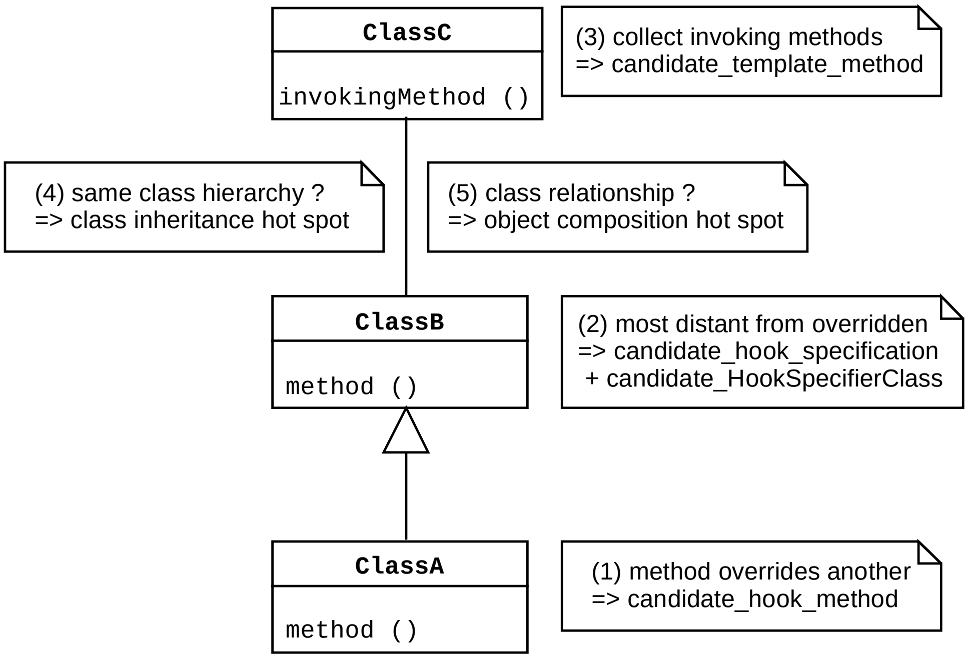

ClassC
invokingMethod ()
(3) collect invoking methods
=> candidate_template_method
(4) same class hierarchy ?
=> class inheritance hot spot
(5) class relationship ?
=> object composition hot spot
ClassB
method ()
(2) most distant from overridden
=> candidate_hook_specification
+ candidate_HookSpecifierClass
ClassA
method ()
(1) method overrides another
=> candidate_hook_method

Figure 9: Overview of the hot spot detection recipe.

4. Check whether the candidate_template_method and the candidate_hook_specification are both defined on the same class hierarchy. If so, you have detected a class inheritance hot spot: you may proceed with the next candidate_template_method. If not, you have discovered an object composition hot spot and furher analysis is required to learn how that hot spot must be filled in; you should proceed with the next step.

5. Analyze the implementation of the candidate_template_method to infer the precise relationship between the class defining the candidate_template_method and the candidate_HookSpecifierClass. You may find (a) an instance variable refering to an instance of candidate_HookSpecifierClass; (b) a parameter of the template method refering to an instance of candidate_HookSpecifierClass; (c) a global variable refering to an instance of candidate_HookSpecifierClass; (d) a function invocation returning an instance of candidate_HookSpecifierClass.

Possible Difficulties and Limitations.

- The process of detecting class inheritance or object composition hot spots, relies heavily on the discovery of method overriding. However, if the hot spot is never filled in (because the hook method provided the right default behaviour) this process will not be able to detect the hot spot.

- Quite often, different template methods are combined into a single hot spot – to fill in such a hot spot one must override several methods. The above process fails to say which template methods belong together.

- To find the TemplateClass once you have a candidate HookClass you must be able to query which method invokes the candidate hook method. Most programming environments come with code browsers that provide such functionality. However, since a lot of method invocations rely on late binding polymorphism, it may be difficult to pinpoint the exact TemplateClass.

- In the case of object composition hot spots, the analysis of the relationship between the HookClass and the TemplateClass requires considerable human intervention.

References

[Bak92] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[Bak95] Brenda S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. Second IEEE Working Conference on Reverse Engineering*, pages 86–95, July 1995.

[DDN$^+$97] Serge Demeyer, Stéphane Ducasse, Robb Nebbe, Oscar Nierstrasz, and Tamar Richner. Using restructuring transformations to reengineer object-oriented systems. technical report, May 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[Hel95] Jonathan Helfman. Dotplot Patterns: a Literal Look at Pattern Languages. *TAPOS*, 2(1):31–41, 1995.

[JR97] Dean Jerding and Spencer Rugaber. Using visualization for architectural localization and extraction. In *Proceeding of WCRE '97*, pages 56–64. IEEE, October 1997.

[MN97] Gail Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 17(2):29–36, aug 1997.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Development*. Addison-Wesley, 1995.

Acknowledgements. This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT programme Project no. 21975.

Many thanks to the people that have commented on earlier drafts of this document: Eduardo Casais, Jean-Guy Schneider and Oscar Nierstrasz. Thanks as well to Kent Beck –our shepherd at EuroPLOP'98– and Patrick Steyaert, who helped us with the development of the appropriate pattern form.