A Study on Detecting Refactoring Opportunities to Introduce the Abstract Factory Design Pattern in Object-Oriented Code

Călin Jebelean

Department of Computer Science, "Politehnica" University Timişoara, Faculty of Automation and Computers, Bd. V. Pârvan 2, Timişoara, Romania

Phone: (+40) 256-403214, Fax: (+40) 256-403214, E-mail: calin@cs.utt.ro, WWW: http://www.cs.utt.ro/~calin

Abstract: In the world of software systems, the process of evolving is an essential one. If this process represents no challenge on small systems, it becomes unbearable if the size of the system exceeds a certain limit. Therefore, the need arises for software tools that can carry out the necessary transformations automatically. This paper addresses the possibility to detect spots within an object-oriented system where an Abstract Factory design pattern solution was ignored.

Keywords: design-patterns, object-oriented programming, detection of design flaws, design heuristics

1 Introduction

Software systems tend to evolve over time in a process called restructuring or refactoring (if it involves object-oriented software systems). Refactoring means changing the code without changing its actual behavior ([1]). This inherent process was not so long ago performed manually by programmers. Today, there are software systems of such complexity that a manual approach is no longer feasible. More than that, it is prone to errors such that one would also need to perform regression tests after the modifications have been carried out.

In his PhD thesis, Opdyke introduced the idea that the process of refactoring an object-oriented software system can be successfully performed by automated tools. He proposed some common-sense refactorings that can be applied to object-oriented systems and started a chain reaction that gave birth to many automated tools capable of modifying software systems.

A typical refactoring process consists of a number of distinct phases ([3]):

- identify spots within the software system where it needs to be refactored
- determine the refactorings that may be applied in these spots
- apply the refactorings (whether manual or automatic)
- use some automated tools (metric-based, for instance) to make sure the quality of the design has actually increased as an effect of the refactoring
- synchronize the refactored code with other related software artifacts (documentation, tests, etc.)

The first two steps are somewhat related, since an identification strategy is usualy driven by the refactoring that is to be applied there.

A modern trend in object-oriented software construction involves the use of patterns at the design level. Design patterns provide better design-level reusability, flexibility and maintainability of software systems which is why experts recommend that they should be used whenever possible. The appearance of design patterns has boosted the possibilities of the domain of software refactoring. Tools have been built to support the automatic introduction of design patterns in object-oriented systems (e. g. [4], [6], [7]) by using lower-level code transformations. However, there was little interest in automatic detection of hot spots within object-oriented systems where such refactorings were needed. The human operator had to find those spots manually and feed them to some tool that applied the transformation automatically.

This paper wishes to challenge exactly the aspect mentioned above. The work we are going to present here is therefore supporting the first two phases of the refactoring process. We describe some of the issues involved in detecting places within object-oriented code where a design pattern should have been used but wasn't. We take as an example the Abstract Factory design pattern, since it seems to be one of the most promising design pattern to study. Also, we choose Java as the target language, although most of the remainder of this paper is independent of a target language. However, the implementation we currently have ([8]) is focused on Java.

2 The Design Problem

The Abstract Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes ([2]). A typical application of an Abstract Factory is presented in figure 1.

The Client class wishes to use instances of ProductA1 and ProductB1, for example, but in the future it may wish to evolve and use instances of ProductA2 and ProductB2 instead of ProductA1 and ProductB1 respectively. Therefore,

Client code should stay independent on concrete names like ProductA1 or ProductB1. It should only depend on abstractions (Factory, AbstractProductA, AbstractProductB). A different kind of Factory produces different kinds of products. Should the client evolve, the only thing to do would be to change the Factory it currently uses.

A more detailed description of the issues involved is given in [8].

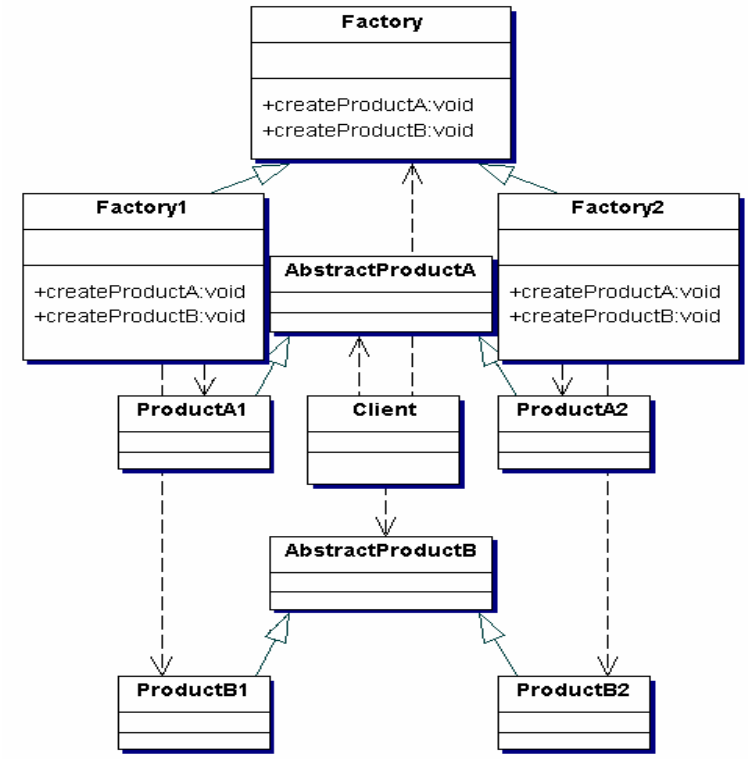

Figure 1
A typical application of an Abstract Factory design pattern

We don't expect to encounter the pattern presented in figure 1 within the code we analyze. Instead, we wish to detect its antipattern, that is, a situation where the recommended use of an Abstract Factory has been completely ignored. Then, we plan to automate the detection process.

The antipattern in this case is not difficult to obtain, in its most simple form. A suspect would be a class which directly instantiates concrete classes like ProductA1 or ProductB1, ignoring the "program to an interface, not to an implementation" paradigm ([2]). We sketched the antipattern in figure 2.

Basically, client code is full of *new ProductA1(...)* and *new ProductB1(...)* statements. If there are more than two products instantiated, there is no problem. However, there must be at least two to start the inquiry. In the future, the software system may evolve and the maintainer may decide that ProductA1 and ProductB1 are obsolete. As an object-oriented software engineer, he deals with the problem by creating brothers for each of them and use these brothers instead of the initial products. This implies a lot of work for classes such as the Client class.

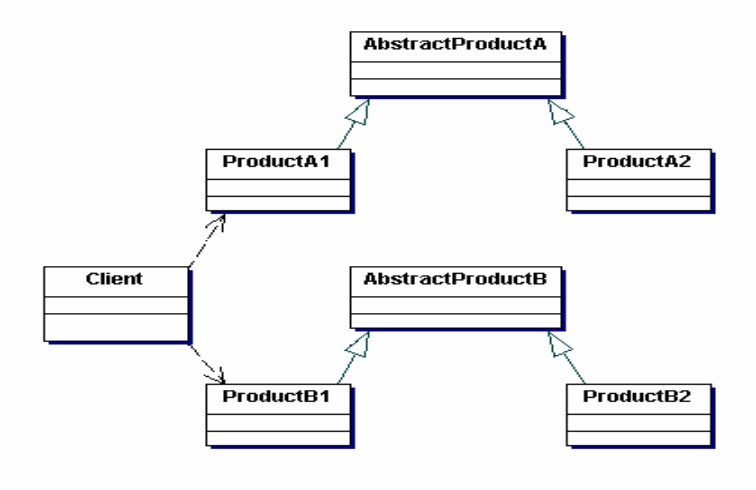

Figure 2
The Abstract Factory antipattern

The solution is to detect situations like the one described above and refactor them by introducing an Abstract Factory design pattern instead.

3 The Detection Strategy

We handle the problem at different levels of complexity. Of course, a higher complexity will lead to more accurate results, since the method we use is rather heuristic than deterministic. We identified three acceptable levels of complexity and will present them in the following sections.

3.1 Method Level Blind Search (MLBS)

A client class is seen as a suspect if it contains a method where it instantiates a ProductA1 and a ProductB1, regardless of the location of the creational statements within the method's code. This situation is depicted in figure 3.

```
class C {
  ...
  public method m(...) {
    ...
    new ProductA1(...);
    ...
    new ProductB1(...);
    ...
  }
  ...
}
```

Figure 3
A MLBS example

Such a situation is easy to detect, however, it may lead to false positives, that is, classes reported as suspects although they are not. ProductA1 and ProductB1 are both instantiated within method m of class C, but only one of them could be instantiated at runtime because we ignore the control path information for method m. For example, the creational statements for ProductA1 and ProdutB1 may be on different branches of the same conditional statement. Although it may be a bad design decision to intensively instantiate concrete classes, there is not the case to use an Abstract Factory design pattern there. If not used together, ProductA1 and ProductB1 may not form a family of products which would normally scream for an Abstract Factory refactoring.

In spite of the false positives, though, MLBS may prove useful in many cases because it detects suspects (false or not) very quickly. Human intervention is then required to process the results.

3.2 Method Level Control Path Search (MLCPS)

A client class is a suspect if it contains a method where it instantiates a ProductA1 and a ProductB1 along the same control path. Figure 4 presents the situation.

```
class C {
  ...
  public method m(...) {
    ...
    new ProductB1(...);
    if (...) {
```

```
    new ProductA1(...);
  } else {
    new ProductB1(...);
  }
  ...
}
...
}
```

Figure 4
A MLCPS example

Method m of class C contains two visible control paths, because of the conditional statement. There is one control path along which both ProductA1 and ProductB1 are instantiated so class C is a serious suspect.

2.3 Class Level Control Path Search (CLCPS)

MLCPS is complex enough to detect most of the situations of missing Abstract Factory. However, a tricky programmer can easily fool it by placing creational statements in different methods and calling these methods successively. The situation is described in figure 5.

Although no method of class C instantiates both a ProductA1 and a ProductB1, a call to method m of class C will eventually create both these products, so class C is a potential suspect.

```
Class C {
  ...
 public method ma(...) {
   ...
   new ProductA1(...);
   ...
 }
 ...
 public method mb(...) {
   ...
   new ProductB1(...);
   ...
 }
```

```
 ...
 public method m(...) {
   ...
   ma(...);
   ...
   mb(...);
   ...
 }
 ...
}
```

Figure 5
A CLCPS example

All these strategies are ultimately searching for places where both a ProductA1 and a ProductB1 are instantiated. However, in a large project there may be a lot of such places, many of them being false positives. To drastically reduce the number of false positives, we propose a simetrical approach. If a spot is detected where both a ProductA1 and a ProductB1 are instantiated, we find another spot within the same project where both a ProductA2 and a ProductB2 are instantiated. For example, if we use MLCPS and find a suspect method m1 in a class C1 instantiating ProductA1 and ProductB1, we must find another suspect method m2 in a class C2 instantiating ProductA2 and ProductB2. ProductA1 and ProductA2 must be relatives on different inheritance branches and so must be ProductB1 and ProductB2. This makes both method m1 of class C1 and method m2 of class C2 suspects.

4 Implementation Issues

While implementing the above-mentioned detection strategies, we found MLCPS to be most representative for what we aim. Although more precise, CLCPS has a complexity which discourages any efforts, for the moment.

Implementation of the MLCPS strategy requires for each method in each class of the project the computation of control paths. Each control path will be associated with a set of classes instantiated along that control path. A method will normally

contain more than one control path, so for each method a list of sets of classes is computed ([8]).

Currently, we assign to each class in the project a unique numeric identifier, such that a class can be unambigously identified using only an integer instead of a string of characters representing its name. Thus, a set of classes can be seen as a bit vector. A class is a member within the set if the bit corresponding to its numeric identifier is set. A set of sets of classes is implemented as a vector of bit vectors but we now study the possibility to use multiple-valued decision diagrams instead, to improve the memory consumption ([9]).

Information about the project under analysis, including the lists of instantiations for each method in each class, is gathered as a Prolog knowledge base and the search for ProductAi and ProductBi instantiations along some control path is performed using the Prolog inference machine, as described in [8].

5 Related Work

The only work we found so far regarding these issues is presented in [5]. They address the same aspect of finding spots for refactoring to introduce the Abstract Factory design pattern. However, they don't consider the control paths of a method, being in that respect somewhat similar to the MLBS strategy presented above. Instead, they base their assumptions on analyzing two or more versions of the project.

Conclusions

In this paper we show how automated tool support can be provided for detecting design flaws within object-oriented systems. The issue of detecting design flaws that require the introduction of a design pattern has been very rarely challenged. We have already developed a tool that can process the Abstract Factory design pattern in the manner described above. We have also identified detection strategies for a small number of other design patterns and we plan to implement them as future work.

References

[1] W. F. Opdyke: Refactoring Object-Oriented Frameworks, PhD thesis, University of Illinois at Urbana Champaign, 1992

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

[3] T. Mens, T. Tourwe: A Survey of Software Refactoring, IEEE Transactions on Software Engineering, Vol. 30, No. 2, February 2004

[4] L. Tokuda, D. Batory: Automated Software Evolution via Design Pattern Transformations, Proceedings of the 3rd International Symposium on Applied Corporate Computing, Monterrey, Mexico, 1995

[5] S. U. Jeon, J. S. Lee, D. H. Bae: An Automated Refactoring Approach to Design Pattern-Based Program Transformations in Java Programs, Proceedings of the 9th Asia-Pacific Software Engineering Conference, Gold Coast, Australia, December 2002

[6] M. O'Cinneide: Automated Application of Design Patterns: A Refactoring Approach, PhD Thesis, University of Dublin, Trinity College, 2000

[7] T. Mens, T. Tourwe: A Declarative Evolution Framework for Object-Oriented Design Patterns, Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society, 2001

[8] C. Jebelean: Automatic Detection of Missing Abstract Factory Design Pattern in Object-Oriented Code, International Conference on Technical Informatics, CONTI, Politehnica University Timişoara, May 2004

[9] T. Kam, T. Villa, R. K. Brayton, A. L. Sangiovanni-Vincentelli: Multiple-Valued Decision Diagrams: Theory and Applications, International Journal on Multiple-Valued Logic, 1998