Design Patterns: the Devils in the Detail

Mel Ó Cinnéide
School of Computer Science and Informatics
University College Dublin
Dublin, Ireland
mel.ocinneide@ucd.ie

Paddy Fagan
Cúram Software
Dublin, Ireland
PFagan@curamsoftware.com

Abstract

The application of a design pattern in an industrial context is frequently a much more involved task than is described the pattern description itself. In this experience paper we report on a number of problems encountered in the application of several common patterns in commercial software systems. The problems we examine range in nature from details of the runtime environment that hamper pattern implementation (Singleton), to the software upgrade process breaking a pattern's promise (Abstract Factory), to the consequences of the tight source code coupling produced by pattern application (Facade).

Our conclusion is that while design patterns are indeed useful in industrial software development, there are more potential pitfalls in this area than is generally realised. In applying a design pattern, more must be taken into account than just the design context into which the pattern fits; issues to do with the low-level runtime environment as well as the higher-level software architecture, software process and social environment also play a role.

1. INTRODUCTION

Design Pattern texts invariably describe patterns in the context of small, readily-understandable systems [10]. This is entirely reasonable from a pedagogical perspective, as a designer encountering a pattern for the first time needs to comprehend the essence of the pattern, not every detail of its implementation. However, in the application of patterns in industrial systems, many issues arise that are not even mentioned in the standard patterns texts. These issues may appear minor in nature, but they can be the crucial factor in determining whether or not a pattern is really applicable in a given context.

In this paper we describe a number of industrial problems that one of the authors (PF) has encountered that have militated against the use of a certain design pattern, in a context where the pattern would have appeared eminently suitable. The design patterns we examine, and a brief description of the problems discussed, are as follows:

- *Singleton*: The manner in which the Java Virtual Machine manages class loading makes the implementation of a single-instance class problematic.
- *Abstract Factory*: A framework must usually be both upgradeable and customisable, but these properties make it hard for Abstract Factory to provide the flexibility it should.
- *Facade*: The tight coupling that is implicit in this pattern can lead to complex problems of source code contention.

This paper is arranged as follows. In section 2 we describe related work on the limitations of design patterns. In sections 3, 4 and 5 we describe issues with the Singleton, Abstract Factory and Facade design patterns respectively. Finally, the conclusions to be drawn from this work are presented in section 6.

2. BACKGROUND

In this section we describe the background to this work by describing related work on the limitations of design patterns.

2.1 Development languages

Much of the original work in the application design patterns was based on examples developed in the C++ language [10]. With the passage of time other object-oriented languages have become more popular and further work has been done to implement design patterns in these other languages. In many cases the implementation language has no impact on the design pattern being used, but there are key places where technical limitations of the language limit the applicability of certain design patterns.

Many books and papers have been written about the applicability of design patterns to certain programming languages [11, 19, 18]. Much less has been written about the situations where technical limitations of the language limit the applicability of certain design patterns; one such example is Fox's work on the Java language [8].

While the development language can play a major role in the details of the implementation of a design pattern, in general it does not affect the types of issues encountered in their use.

2.2 Critiques of design patterns

The widespread adoption of design patterns for software design also spawned a series of critiques of design patterns

which attempt to document the negative aspects of design patterns. A number of texts in this area have focused on the misapplication of design patterns, Kerievsky's work which comments on the overuse of the Singleton pattern is a recent example of this [15].

Several of the publications in this area have directly influenced the work presented in this paper in particular [15] and [7].

2.3 Industrial software development as a social process

The process of developing software involves people; large projects can involve hundreds of people. How and when these people interact have a key influence on the software that will be produced. These facts are often disregarded when the "nuts and bolts" of a software project (code, patterns and architecture etc.) are discussed, but this can cause certain key factors to be overlooked. DeMarco and Lister [5] provide many essays on the topic of building cohesive and productive teams in order to build good software and many other books and papers have been produced in this area.

A further facet of the social process of software development centres on the behavioural patterns of the people involved in the process as they respond to events. This is an area that has received relatively little study to date, and is usually mentioned only as an aside in works on the software development process [4, 13, 14]. However, these behavioural patterns are key to some of the arguments presented in this paper. Particularly how the application of certain design patterns gives rise to a particular code structure, which in turn influences how people will approach updating this code. In section 5 on the Facade pattern we will see an example of this.

2.4 Balance of market and technical forces

Striking a balance between financial and technical pressures is one of the key features of successful commercial software projects. This balance changes over the lifetime of a project, as the market, and the product's place in the market, evolve. A failure to get this balance right can clearly be disastrous for both the project and the company involved [13].

Given these factors one can expect the level of use of design patterns, and indeed the applicability of design patterns, to vary from project to project. For example, in a situation where time-to-market dominates all other factors, it may make commercial sense to eschew completely the use of design patterns in the initial release.

However, many projects have attempted to use design patterns in a consistent and well-defined way, with strong training and other support efforts to help the development teams. Beck et al [2], examined the experiences of a number of projects, and found in general the feedback from the development teams to be overwhelmingly positive.

It is vital to realise that in a commercial environment, a positive experience for the development team is not itself sufficient to justify the cost of such an approach, so some additional justification for the use of design patterns is needed. Hohmann [13], examines many of the trade-offs of developing software in a commercial environments in detail.

There is the well-documented belief that the use of design patterns can lead to lower maintenance costs [3]. It should also make it easier for new developers, or developers who have not seen a section of the code before, to familiarise themselves with the code, but there are times when neither of these arguments is as important as the timing of the initial deliveries.

In order for a commercial product to survive and prosper it must always create for itself a viable position in the market place. This drive to produce a successful product influences the use of design patterns and the changes that take place as a product matures. A robust design will always form part of those products that survive and prosper in the market place in the long term, and for these products the design patterns form part of that, even if they are not used extensively in the initial versions of the product.

In the following sections we describe a number of design pattern implementation issues that arise in industrial contexts. The patterns we are concerned with are Singleton, Abstract Factory and Facade.

3. SINGLETON

The Singleton pattern [10] is applicable when there should be only a single, globally-accessible instance of a class. Work by Hahsler [12] has shown the Singleton pattern to be very heavily used in industrial software[1]. In the authors' opinion there is a number of reasons for this popularity:

1. It is one of the easiest patterns from the GoF text [10] to understand and use.

2. Applying it to systems where needless object creation causes problems can lead to very significant performance gains.

3. Global variables occur widely in industrial software, even though they are generally considered to be poor style. Refactoring to the Singleton pattern to resolve this issue is one possible solution.

In spite of the popularity of Singleton, and the popularity of the Java language, there is a little-known property of the Java Virtual Machine that leads to fundamental problems with the implementation of the Singleton pattern in Java. We consider this problem in detail in the following section.

3.1 Singleton and the JVM

3.1.1 Introduction

The Singleton pattern relies on it being possible to code a class so that only one instance of that class can exist in the system. With certain Java Virtual Machines (JVMs), those that support multiple class loaders, this is no longer guaranteed and hence the Singleton pattern will no longer operate as the developer intended. A short description of this problem is given by Joshua Fox [8], but without any

[1]Hahsler's work involved searching software projects for pattern names. Singleton was in fact the third most common pattern name found in the projects analysed. However, the two most common pattern names found, "Command" and "State" are likely to appear in source code for reasons other than the implementation of those patterns. Since the term "Singleton" is not generally used outside the context of the Singleton pattern, it is probably the most popular of the design patterns considered by Hahsler.

analysis of the behavioural problems this may cause[2]. To explain how and why this happens, we will first give some background on how the Java language represents classes at runtime and how the runtime environment accesses them.

3.1.2 Background to the Singleton/JVM Problem

We first examine what happens when the Java runtime environment attempts to access a class. Full details of this aspect of the Java Virtual Machine are provided by Liang and Bracha [17]; here we provide only a summary of the information relevant to our work. Consider a simple Singleton class, `MySingleton`, as presented in Figure 1. After compilation, the resulting Java class file can be loaded by any JVM so that an instance of `MySingleton` can be created.

```
public class MySingleton {
  public static MySingleton getInstance() {
    if (instance == null) {
      instance = new MySingleton();
    }
    return instance;
  }
  private static MySingleton instance = null;
  private MySingleton() {}
}
```

Figure 1: Simple Singleton Implementation in Java

The JVM loads the class using a component known as a class loader. The class loader creates an instance of the internal Java class `Class`, to represent the class just loaded. All classes loaded by a class loader are held in a namespace associated with the class loader. This situation is represented in Figure 2.

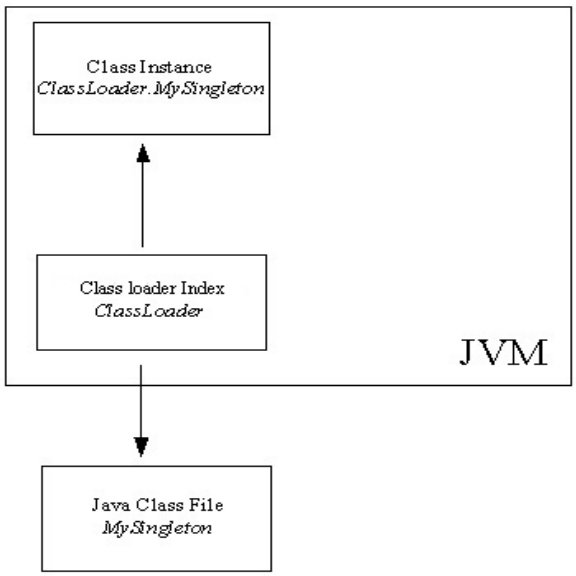

Figure 2: JVM with single class loader

[2]Other authors, such as Steven Metsker [9], note other problems with Singleton in multithreaded environments in Java, with similar behavioural issues, but with a differing underlying cause.

When there is only a single class loader instance inside the JVM everything works as might be expected. However it is not uncommon for a JVM to use multiple class loaders. For example, commercial Application Servers require the use of JVMs with multiple class loaders in order to offer advanced features such as the online upgrading of software components. This feature requires the reloading of a subset of classes already loaded in the running JVM. In order to reload an already loaded class it must be loaded by a different class loader, which means that multiple class loaders are required to support this feature.

When there are multiple class loaders in the JVM, any class can be loaded by a number of different class loaders. A client class will see only one version of a multiply-loaded class, namely the version that was loaded by the same class loader that loaded the client. Since no class can specify the class loader that loads it, this is entirely under the control of the virtual machine. This is represented in Figure 3.

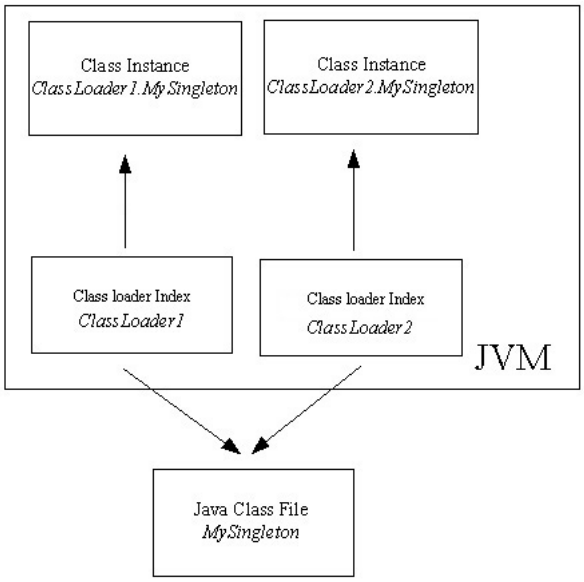

Figure 3: JVM with multiple class loaders

If this happens with a Singleton class, two or more instances of the Singleton will exist in the system, breaking the fundamental constraint of the pattern. This leads us to a discussion of the consequences of this problem.

3.1.3 Consequences of the Singleton/JVM Problem

Given that under certain situations multiple instances of a Singleton can be concurrently present in a system, what are the likely runtime issues? If the Singleton in question is a mutable object, then changes made by one client will not be seen by other clients, and run-time errors are likely to occur.

It is not uncommon for a Singleton to be immutable, but here also problems are also likely to occur. In the author's experience several distinct problems can manifest themselves in this situation. These include:

1. *Locking* If the Singleton class writes to an output stream that does not support multiple concurrent writers, only one instance can have the lock and hence some clients

will report errors using the Singleton where others will work correctly.

Of course, this pre-supposes that the error handling in the client/Singleton interaction is sufficient. However, because Singletons are seen as classes which will either work or fail in totality, error checking around them is often sub-standard.

2. *Overwriting* If the Singleton writes to an output stream that does support multiple concurrent writers, all instances can write simultaneously, in this case one writer will overwrite the others or the writes from the different writers may be interleaved or all the written data is corrupted.

   The implications of this issue can vary from the annoying, log files are incomplete or hard to read, to the fatal, where critical business information is lost or corrupted.

3. *Deadlocking* If the Singleton writes to a data store that supports multiple concurrent writers with some form of object locking within the data store (e.g., a relational database) it is possible that the different Singleton instances will deadlock with each other. This will occur when one Singleton instance obtains a lock on an object that a second instance wants to write to, while the second instance holds a lock on another object that the first instance wants to write to.

   Most modern databases include deadlock detection features, so that one of the transactions will be designated the "deadlock victim" and be rolled back with an error, while the other transaction will be allowed to complete. Even this is likely to be problematic however, as the developer is unlikely to have considered the possibility of the Singleton's operation failing due to a deadlock condition.

An even more insidious problem occurs when the multiple-instance problem exists, but the program operates correctly. This cannot be guaranteed to continue to be the case as the system is evolved and updated, and is likely to lead to errors at a later date.

Even worse the system may operate correctly most of the time and only suffer intermittent errors. In this case it may be extremely difficult for the developers to track down the error because of the difficulties in reproducing it[3].

3.1.4 Managing the Singleton/JVM Problem

At first, it may appear that current JVMs are flawed in how they allow and handle multiple class loaders. In fact this behaviour is essential to the correct operation of class loading, so it is incumbent on the Java developer to be able to deal with it.

The key to addressing this issue is understanding it. If the developers are aware that more than one instance of a Singleton can be present in the system concurrently there are steps they can take to prevent errors from occurring. There are two approaches that may be taken to manage this problem:

1. *Avoid using the Singleton pattern* This is the most principled approach. If there is any possibility that code will be used in a multiple class loader environment the developer should not apply the Singleton pattern. In this context the "Inline Singleton" refactoring described by Kerievsky [15] could be used to remove existing Singletons.

2. *Make the Singleton tolerant of multiple instances* In this approach the developer should code the Singleton so that it can cope with more than one instance of itself being present in the system concurrently. This is a rather vague principle of course, and is contradictory to what is normally understood by a Singleton.

   It is also critical that the error handling in and around Singletons be well thought-out and implemented. This will ensure that the system will sensibly report any errors that may occur should it be deployed in a JVM with multiple class loaders.

4. ABSTRACT FACTORY

The Abstract Factory pattern [10] enables a developer to defer the decision regarding which family of concrete classes is to be used until the last possible moment, and makes it possible to change this choice without requiring clients to be updated.

This kind of flexibility is particularly useful for example in creating a single code base that supports multiple operating systems and windowing systems, without littering the code with selection statements enumerating each possible configuration. Another key aspect of this pattern is that it separates the interface of a class library from its implementation, so they can be changed independently.

Such flexibility is highly desirable in the domain of industrial software where changing requirements, or the reinterpretation of existing requirements, can be a daily occurrence. However, in the context of Enterprise frameworks achieving this desired flexibility using the Abstract Factory pattern is not as straightforward as it initially appears. We examine this problem in more detail in the subsection below.

4.1 Abstract Factory and the Framework Upgrade Problem

Enterprise frameworks are becoming increasingly common in industrial software. The key requirement is to provide a cost effective pre-built solution for the vast majority of the customer requirements in a particular domain, with continuing upgrades, full support and proven performance. Fayad and Schmidt list the advantages of such frameworks as modularity, reusability, extensibility and inversion of control [6].

However, in order to provide the advantages listed above, enterprise frameworks have different development requirements from most software projects. In particular, the additional feature of upgradeability is also required, so that it is easy for customers to take on new versions of the framework without undue impact on their existing extensions/customisations.

These requirements combine to create unique pressures, requiring unique and innovative solutions. Design patterns can play a pivotal role in such solutions, but they cannot

[3]The author's experience with this problem occurred when the JUnit tests which he was using triggered multiple class loader behaviour. The problem manifested itself when a static member of a class was observed to be not equal to itself! It took several days, and the insight of a colleague, to track down the cause.

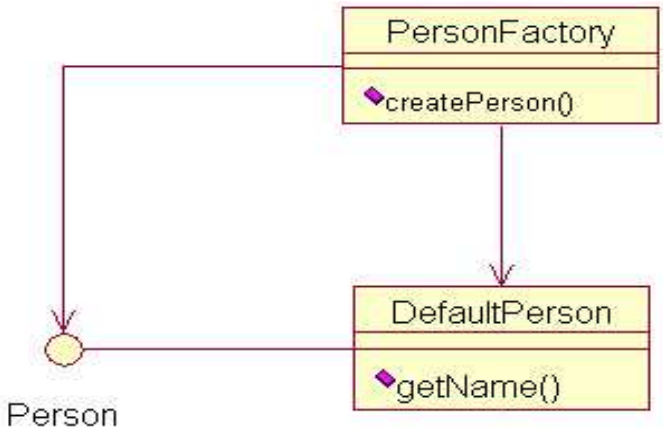

Figure 4: Example Factory Implementation in a Framework

meet all of the requirements without other supporting elements of the solution. Here we consider how Abstract Factory can be used in such frameworks and the trade-offs this implies.

4.1.1 Example Abstract Factory Implementation

Consider an Enterprise Framework in the Social Services domain that uses Abstract Factory in the creation of the products it uses (`Person`, `Grant`, etc.). To simplify the discussion, we'll consider only the `Person` class in the subsequent discussion. The framework ships with a default concrete factory `PersonFactory` that provides a method `createPerson` that creates and returns an instance of the `DefaultPerson` class. The framework uses an instance of the `PersonFactory` to create `Person` objects, as does the client code written by the developers who deploy the framework. This is shown in Figure 4.

In tailoring the framework, the client developers will need to create their own `Person` class, `CustomPerson`, which is a subclass of `DefaultPerson`. This requires the creation of a new `CustomPersonFactory` class, either by subclassing or by code generation. In either case the client code must install this `CustomPersonFactory` object so that both the framework code and the developer code uses it. This is shown in Figure 5.

The issue we are concerned with arises when a new version of the framework is shipped. The developers of the framework are of course unaware of changes in the client code, and may have changed some of the classes that implement or use the `Person` class. As a result, the client code may no longer work with the updated version of the framework, either because of changes to the clients of the `Person` class or because of changes to the `Person` class itself. With the example above it is worth considering the impact of a new version of `Person` class which implements a `getAge()` method with a different return type to that specified in the `CustomPerson` class.

The problem described here is, of course, not as a result of applying the Abstract Factory pattern, but can occur whenever clients customise a framework and later receive an upgraded version of the framework. However, the framework upgrade problem has an particularly negative impact on the implementation of the Abstract Factory pattern. The promise of this pattern, the separation of clients from product classes, cannot be guaranteed.

4.1.2 Consequences of the Framework Upgrade Problem

The Framework Upgrade problem described above imposes a number of key limitations on customers using the framework:

1. Customers need to examine their customisation if an update to the framework changes a method they have overridden.

2. In a large application a large amount of code is required to support this approach, particularly given that a single customer is unlikely to have overridden more than a handful of methods.

Some of the traditional issues with the use of the Abstract Factory pattern can present other limitations. For example, if there is any direct instantiation of the `Person` class, this approach will no longer work. Also, any changes to the `Person` interface can cause problems, as the original implementation class will no longer implement the interface.

4.1.3 Addressing the Framework Upgrade Problem

Using the Pluggable Factory pattern [20] in this situation would remove the need to change the factory when tailoring the framework, as the factory could simply be parameterised with the new implementation class as the prototype. However, this approach is more susceptible to problems where the appropriate prototype has not been passed to the Factory. Nor does this solution address the other consequences noted above.

Probably the other key alternative in this situation is simply to not apply any design pattern at all on the grounds that it can never be a total solution, and instead rely on other approaches to meet the requirements. This would put a very large focus on documentation and developer discipline to ensure that the selected approach is followed.

4.1.4 Conclusions

If Abstract Factory is to be used in an enterprise framework, which still provides the modularity, reusability, exten-

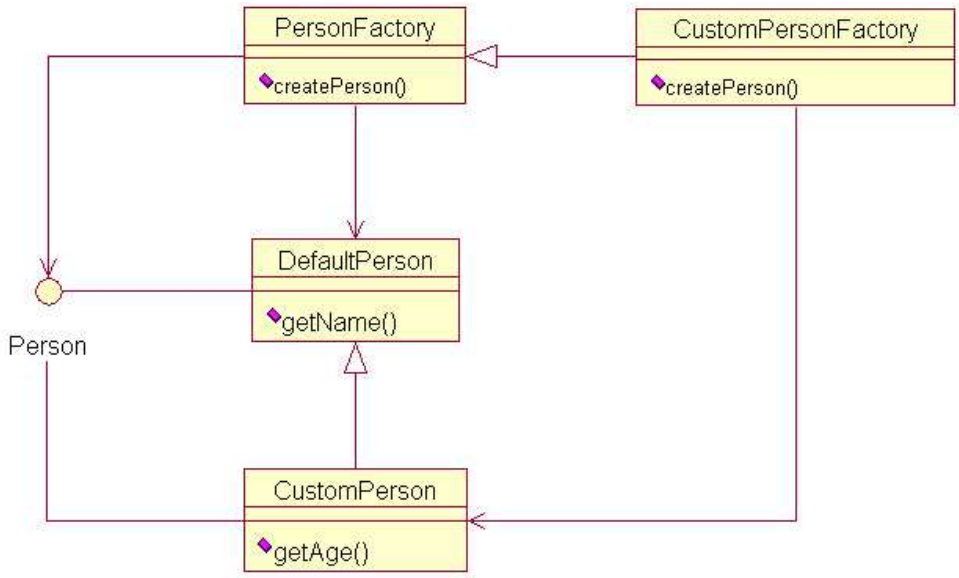

Figure 5: Example Factory Implementation in a Framework with extension

sibility, inversion of control and upgradeability required by such frameworks, it requires:

1. A mechanism to prevent direct instantiation of implementation classes.
2. A mechanism to allow the customer extend the interfaces of the implementation classes.
3. A set of clear documentation on how the upgrade process will operate.

The key issue with complex problems of this kind is that any solution will be a compromise and will have to encompass more than just the software itself. These problems can be addressed in part by design patterns, but design patterns alone cannot provide a complete and robust solution. Developer awareness of the problem and a disciplined approach to managing it are as important as the features of the pattern itself.

5. FACADE

The Facade pattern [10] is used to define a higher-level interface to a subsystem that makes the subsystem easier to use. This pattern is very useful where subsystems provide collections of related features that can be grouped together and presented in a unified way using a Facade class. This kind of abstraction is highly desirable in the volatile world of industrial software and can help insulate clients from changes in the implementation of features they use.

The Facade pattern can also be used to ensure that only those features of the subsystem that the developers intended to publish are available to clients. Here the Facade acts as the public interface to the subsystem, with all other subsystem classes protected so that classes outside the subsystem cannot access them. This is particularly useful where the clients of the subsystem may be developed independently from the subsystem itself, and as such the developers of the subsystem may have limited control over the way the subsystem is used.

In many development languages, such as Java, each class is implemented in a single source file. This means that the source files for a small number of facade classes can become points of contention during the development process. It is this aspect of the Facade pattern that we investigate in more detail in the following section.

5.1 Example Facade Implementation

Returning to the Enterprise Framework in the Social Services domain, it uses facades to provide a high-level interface to each of the subsystems which implement the products it uses (`Person`, `Grant`, etc.). To simplify the discussion, we'll consider only the facade to the `Person` subsystem in the subsequent discussion. The facade to the `Person` subsystem contains 50 methods, these depend on 25 implementation classes, each implementation is in turn constructed by a separate factory. Because the facade is exposed to clients as an EJB, there are also two Transfer Objects (key and result) for each method. The interface and EJB transfer objects depend on 20 "basic type" classes which are used to abstract basic domain values, social security number, money etc. throughout the system. In total then the facade to the `Person` subsystem depends on 170 other classes. See Figure 6.

It should be noted that much larger facades exist in this system, with the largest depending on almost 1000 other classes. It is clear that the use of the facade pattern leads to large numbers of dependencies, which must be considered when using this pattern.

5.2 Facade and Tight Coupling

The Facade pattern tends to generate classes that are tightly coupled to a large number of other classes in the system. In the example above it is clear that the facade is tightly coupled with the interfaces to the implementation classes in the subsystem and to the EJB transfer objects that form part of its interface.

It is well-known that classes with a high level of coupling are significantly more error-prone than classes that have less

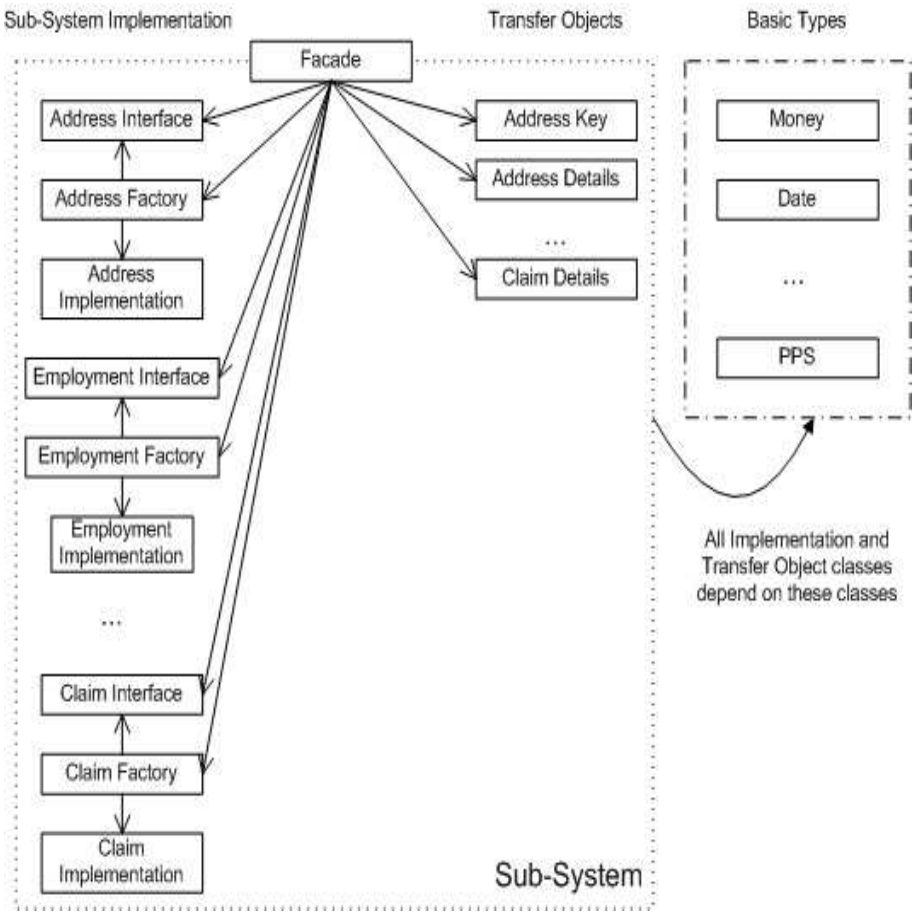

Figure 6: Facade to `Person` subsystem

coupling [1], and this may lead to increased maintenance costs. Assuming that the Facade pattern has been applied appropriately, it may be assumed that the flexibility provided by the pattern outweighs the associated maintenance costs.

However, there are other factors associated with the tight coupling that results from the use of Facade, namely the deleterious effect on build time and the problem of contention over source artifacts. We examine these in the following subsections.

5.2.1 *Build time of the application*

Build time can be a serious issue for industrial software projects. A full build, even on high-powered machines, may take many hours or even days. The work by Laros on compile-time coupling and build time, exemplifies this kind of issue [16]. He considered the case of a C file that included multiple header files, each header file one hundred lines in length. As the number of header files increased so did the build time, until very quickly the build time became unacceptable. See Figure 7.

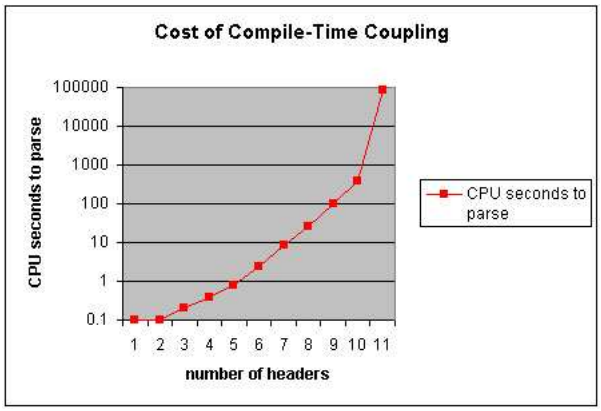

Figure 7: Cost of Compile-Time Coupling, based on data from [16]

Even with languages with runtime linking, like Java, similar problems exist because the dependency checker has to resolve the dependencies between classes. When determining which classes need to be recompiled for a given change, the dependency checker has to navigate these complex dependency trees. At best this will slow the build down, but at worst it will cause the builder to do a full build, because it becomes impossible for the dependency checker to resolve the dependencies fully[4]. This can mean that a small in-

[4]On one project with which the author was involved, he was puzzled by the seemingly unnecessary full builds that were triggered by small updates to the program. An examination

crease in the level of interdependencies can cause a massive increase in average build times.

This increase in build time causes a direct loss in productivity and increases developer frustration. The impact is even greater in the context of Agile Processes, where regular builds are an inherent part of the development process.

5.2.2 Contention over source artifacts

In the authors' experience, contention over source artifacts can cause different problems depending on the source control practices within the organisation. If no concurrent editing of a single artifact is supported, a queue of changes builds up waiting on access to change the key class, in this case the facade. Other changes inside the subsystem cannot be completed, because the facade class needs to be updated. It is important to note that this most often occurs where the interface to the facade class does not need to change, but rather the implementation needs to be updated to match changes to the implementation classes inside the subsystem.

Where concurrent editing of a single artifact is supported, the integration of the changes often proves problematic. At the same time, the sophistication of automated merging included in the source control system can affect how these problems present themselves. The important factor is that any concurrent editing of a single artifact can result in one or more sets of changes being returned to the developers to be manually merged. It is also worth noting that social issues also come into play here, where developers become reluctant to submit changes where they fear a merge conflict may exist. This can mean that advantages of the more flexible source control approach can be lost because of developer behaviour.

5.2.3 Facade Conclusions

The value of the Facade pattern is clear in terms of abstracting access to a set of subsystems in a cohesive way. However, the tight coupling of the Facade classes to the rest of the system has to be borne in mind in applying this pattern. Problems of increased maintenance costs, longer build times, and potential problems with developer contention for Facade source files are all factors that may weigh against using Facade.

6. CONCLUSIONS

We have taken three sample design patterns from the standard GoF text [10] and considered some of the practical problems encountered by the authors in applying these patterns in an industrial context. The problems ranged in nature from details of the runtime environment that hamper pattern implementation (Singleton), to the software upgrade process breaking a pattern's promise (Abstract Factory), to the consequences of the tight source code coupling produced by pattern application (Facade). These problems are not immediately obvious from the pattern description, and yet they must be considered in applying the pattern in a real system.

of the Eclipse Java Builder in use revealed that when the dependency tree reached a certain depth, a full build was automatically triggered. Each time a change was made to a tightly coupled Facade class, like those mentioned in the example above, this "high watermark" was reached and a full build resulted.

The problems described are not insurmountable of course. However, they demonstrate that in deciding to apply a design pattern a developer must take into account more than just the design context into which the pattern fits; issues concerning the low-level runtime environment and the higher-level software architecture, software process and social environment also play a role.

7. ACKNOWLEDGEMENTS

We would like to thank our PLoP shepherd Bob Hanmer, and all our other reviewers, for their insightful comments and suggestions.

8. REFERENCES

[1] V. Basili, L. Briand and W. L. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Transactions on Software Engineering, 1996.

[2] Kent Beck et al. Industrial Experience with Design Patterns, Proceedings of the 18th International Conference on Software Engineering, 1996.

[3] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler and L. G. Votta, A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions, IEEE Transactions on Software Engineering, 2001.

[4] James O. Coplien and Douglas C. Schmidt, Pattern Languages of Program Design, Addison-Wesley, 1995.

[5] Tom DeMarco and Timothy Lister, Peopleware - Productive Projects and Teams, Dorset House, 1999.

[6] Mohamed Fayad and Douglas C. Schmidt, Object-Oriented Application Frameworks, Communications of the ACM, Vol. 40, No. 10, October 1997.

[7] Brian Foote and Joseph Yoder, Big Ball of Mud, Fourth Conference on Patterns Languages of Programs, 1997.

[8] Joshua Fox, When is a Singleton not a Singleton?, JavaWorld, January 2001.

[9] Steven Metsker, Design Patterns Java Workbook, Addison-Wesley, 2002.

[10] Erich Gamma et al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[11] Mark Grand, Patterns in Java vols I and II, Wiley, 1998.

[12] Michael Hahsler, A Quantitative Study of the Application of Design Patterns in Java, Working Papers on Information Processing and Information Management Nr. 01/2003.

[13] Luke Hohmann, Beyond Software Architecture - Creating and Sustaining Winning Solutions, Addison-Wesley, 2003.

[14] Andrew Hunt and David Thomas. The Pragmatic Programmer, Addison-Wesley, 2000.

[15] Joshua Kerievsky, Refactoring to Patterns, Addison-Wesley, 2004.

[16] John Laros, Large-Scale C++ Software Design, Addison-Wesley, 1996.

[17] Sheng Liang and Gilad Bracha, Dynamic Class Loading in the JavaTM Virtual Machine, OOPSLA '98.

[18] Steven Metsker, Design Patterns in C#, Addison-Wesley, 2004.

[19] Bruno R. Preiss, Data Structures and Algorithms with Object-Oriented Design Patterns in C++, Wiley, 1998.

[20] John Vlissides, Pluggable Factory, Part I, C++ Report, November-December 1998.