The Legacy Bridge Problem

by

Douglas Lyon and Chris Huntley

Abstract

We present a way to automate the reuse of legacy systems without multiple-inheritance, copying source code, accessing existing code or modifying the Java language. Our technique involves the automatic synthesis of *bridge pattern* code. The mechanism automatically generates bridge implementations and bridge interfaces that perform message forwarding and multiple inheritance of types.

These elements can be used to *refactor* any legacy system without reengineering it. We examine the tradeoffs between designs based on manual static delegation, automatic static delegation, dynamic proxy classes, and inheritance. Advantages of our technique include improved performance, type safety, transparency, predictability, flexibility and reliability .

The approach automates the generation of Java source code for both method forwarding and interface declaration. Disambiguation can be automatic, semi-automatic or manual. The bridge class can be evolved, transforming into into an *adapter* that protects client classes from specification change.

Problem Statement

A legacy system is a software system that already exists. Typically, as in our case, a legacy system is in a maintenance mode. Specifications have not been maintained and interfaces are fixed. For example, the FAA has an air traffic control system that was written in the 60's. Altering this type of system could be fraught with difficulty. In fact, a reengineering of this system started in the 70's and continues to this day.

The bridge pattern is a commonly cited object-oriented design pattern that is used to provide separation between the implmentation of a software system and its interface. For example, in RMI (Remote Method Invocation) Java provides a interface to an implementation that can reside on a different computer. It may be implemented in almost any language. In fact, any network layer protocol is uses bridge pattern. For example, a link-layer protocol is used between modems to negotiate error correction and packet size. Another example is the CGI protocal that works in the application layer between a browser and a web-server. The web server and the browser act as a bridge to access an order entry system so you don't have to have a native client.

We are given a large legacy system that is fragile, hard to maintain difficult to reverse engineer unchangable, poorly designed but field tested. Our goal is to provide a bridge between new code and our legacy system. The construction of such a bridge represents a solution to the legacy bridge problem.

Legacy systems are common in today's software houses. Bridges are needed to improve software architecture without changing the legacy code. Bridges enable legacy code reuse. They also provide a stable interface between new code and the legacy code. A bridge also provides a transitional benefit, by allowing new code to be written using modern technology and coding practices, without modifying the legacy system.

We have several criteria when evaluating the trade-offs between various approaches:

1. Performance - The speed of execution.
2. Type safety - This criteria addresses the compile-time check of the parameters. If errors like "message not found" are emitted at run-time, we claim that the system is not "type-safe".
3. Transparency - This criteria addresses the ease-of-use. Several built-in language features provide, for example, the ability to add features to a class (using such techniques as *multiple-inheritance*).

4. Predictability - Knowing, in advance, that something will execute and in how much time it will take to execute.
5. Flexibility - Being able reflect the associations between things in the real-world. For example, some systems hold the relationship between a class and its subclass as constant after compilation. Altering this relationship can typically break a system.
6. Reliability - As we attempt to upgrade legacy systems that were improperly design, they tend to become more fragile. A new feature in one place can cause breakage in several other places. When this occurs, we view the system as unreliable.

Motivation

Any help that can be given when dealing with legacy code is welcome. Legacy code will always be with us, in fact, as soon as new code is written, it can be considered legacy code by some. The problem of building a bridge to a legacy system is typical of a real-world software engineering problem found in industry.

We are further motivated to find a solution that provides a working program during every step of the process enables continuous testing. This is a kind of XP (extreme programming) that treats design as an iterative process. The design evolves by *refactoring* the code.

Refactoring is defined as “changing a system to improve its internal structure without altering its external behavior”. Legacy code often needs refactoring in order to improve its design or readability. Refactoring is a key approach for improving object-oriented software systems [Tichelaar].

Approach

A bridge controls the dependencies between software systems that normally complicates an analysis [Korman]. There are several ways to design a bridge. Some are object-oriented, and some are not. When a pre-arranged protocol is used to isolate one computational layer from another, we have an example of the bridge pattern. In a network, for example, there

are generally several layers. Each layer has a defined responsibility. One layer (e.g., the physical layer) might be responsible for making sure that packets of bits are moved from one point to another. The next layer up might be responsible for routing packets on a network. Each layer has a fixed responsibility and an established protocol for communication. This protocol becomes a specification for exactly how the layer works. Thus, by taking the bridge pattern approach to interfacing to legacy code, we are making use of a protocol (i.e., specification) for the communication between the legacy code and the new code.

The bridge pattern allows for changes in the implementation of the legacy code, but the specification for the communications between the legacy code and the new code must remain fixed. Thus, our approach is to fix the interface to the legacy code so that we can write new code, using sound software engineering design principles.

The object-oriented design literature provides for at least three options for solving the legacy bridge problem. These approaches include inheritance, static delegation, or dynamic delegation (i.e., dynamic proxy classes). We examine the various bridge implementations in the following section, then discuss their trade-offs.

Various Bridge Implementations

This section examines the various implementations of the bridge pattern. In our first section we examine inheritance as an easy, commonly used, but poor technique for implementing the bridge pattern. We then examine the alternative, based in *delegation*. We describe the two types of delegation, dynamic and static. We show how dyanmic delegation is easy to implement, but also represents a poor software engineering approach. We then examing static delegation as a sound software engineering practice. Finally we examine the two kinds of static delegation, manual and automatic. The automatic flavor eases the creation of delegates and interfaces used in the creation of bridges. We then show how automatic delegation makes for a generally superior (and new) approach to building bridges to legacy code.

Inheritance

Inheritance is both a design approach and a programming language feature. Generally, inheritance enables shared behavior. It is generally used to define an unchanging taxonomy for the representation of knowledge about things in the world. For example, a mammal is a kind of animal. A human is a kind of mammal. A whale is another kind of mammal. Because inheritance is used to describe *a kind of* relationships it is said to be an AKO (A Kind Of) hierarchy. Inheritance is sometimes called specialization.

The term *class* has been introduced in order to act as a shortcut for the term *classification*. The term *sub-class* has been introduced as a shortcut for the term sub-classification. The cardinality of the elements in a sub-class is smaller than or equal to the cardinality of the elements in the super-class.

Most languages that implement inheritance have static relationships that describe the taxonomy. Inheritance is very popular because it is transparently able to inherit properties from super-classes. The properties include methods for the manipulation of data, as well as the data itself. Thus, because of its ease-of-use, inheritance is often used by programmers as a way to add features to a class, without any epistemological considerations.

This is generally considered an abuse of the language feature. AKO is just one kind of association between things and is often an inadequate way of modeling associations [Frank]. For example, *roles* in an inheritance structure may change. For example, an insurance company sees the children of clients as *dependents* in its software system. However, after the children grow up they can change from dependents to *customers*. In a static inheritance relationship, role changing is not easy. This is a failure to model dynamic evolution of the world [Kniesel]. Thus, in the example of the role, we delegate to role instances that represent kinds of roles that a person may have. Frank suggests the association of *acts-as* be used for various kinds of roles. For example, a *person acts-as* a *student* [Frank].

Inheritance has been shown to have several disadvantages. For example:

1. Subclasses must inherit only a single implementation from a super class.
2. The topological sorting of the super-classes have been cited as a fruitful source of bugs [Arnold 1996].
3. Inheritance compromises the benefits of encapsulation [Coad].
4. Inheritance hierarchy changes are unsafe [Snyder].
5. Even in a single-inheritance type language like Java, conflicts between multiple parents are not reported. Ambiguity resolution has long been known as a problem with inheritance [Kniesel].
6. Taxonomically organized data has become automatically associated with *object-oriented programming* [Cardelli].

Some have said that multiple inheritance is hard to implement, expensive to run and complicates a programming language [Cardelli]. These conjectures were debunked by Stroustrup [Stro 1987].

The inheritance debate rages on without hard data [Tempero]. Inheritance gives us code reuse but at a cost. The uncertainties that arise from the use of inheritance of implementations have been cited as the rationale for leaving some times of inheritance (namely multiple-inheritance of features) out of Java [Arnold 1998].

Despite these concerns, inheritance remains popular. One reason for this might be that in inheritance, classes transparently inherit operations from their superclasses.

We summarize the imlementation of the bridge pattern using our six criteria:

1. Performance - inheritance is generally a high-performance solution that enables invocation of methods without a large over-head.
2. Type safety - for strongly typed languages (like Java) we can be assured that inheritance is type-safe. The compiler will check the type of all the parameters passed into a method, and flag any possible ambiguous invocations.
3. Transparency - inheritance enables easy addition of features to a class, making it very transparent, and popular, as a language feature.

4. Predictability - here inheritance gets modest marks. There is often a question about which method will be invoked, depending on the order of the base classes being listed. On the other hand, in an unambiguous situation, invocation speed is generally well known.
5. Flexibility - Relationships between a class and its subclass are typically constant after compilation. Altering this relationship can break a subclass. Thus, these relationships are inflexible, once established.
6. Reliability - A long chain of sub-classes constitute an improper design. Sub-classes are very dependent on their super-classes for implementations and data-structures. As we attempt to upgrade alter the super classes to add a new feature we can cause breakage in several subclasses. Thus inheritance does not scale well to large systems and is thus unreliable.

In summary inheritance is a high-performance, type-safe and transparent way to add features to a class. However, it can be unpredictable, inflexibilty and unreliable, particularly when faced with large systems.

Delegation

According to one definition, delegation uses a receiving instance that forwards messages (or invocations) to its delegate(s). This is sometimes called a consultation [Kniesel]. A *proxy* class is used to implement the interface to the legacy code. We call the interface to the legacy code the *bridge interface*. In object-oriented parlance, we say that the proxy class reuses implementations in the legacy code by message forwarding.

There are two basic mechanisms by which message forwarding may be accomplished, *dyanmic delegation* and *static delegation*. In the following sections we detail the difference between these two techniques, and the trade-offs in their use.

Dynamic Delegation

Dynamic delegation (sometimes called dynamic proxies) is a means by which a search is performed for a method to invoke *at run-time*. If the

method is not available, or if the invocation is incorrect, a run-time error occurs. This never happens with inheritance or static delegation.

We summarize the implementation of the bridge pattern using dynamic delegation with our six criteria:

1. Performance - a large over-head is needed to search for methods, which makes dynamic delegation slow.
2. Type safety - poor type-safety make it impossible for the compiler to check the existance of a method before run-time.
3. Transparency - easy addition of features to a class, makes dynamic proxies very transparent,.
4. Predictability - invocation speed is generally unknown, infact, even if methods are findable on one system, the speed of finding them is totally unpredictable as we move between platforms or even implementations.
5. Flexibility - relationships are flexible, and new methods can be added without changing existing classes.
6. Reliability - A long chain of dyanmic proxy invocations constitute an improper design. Dynamic proxy code is hard to follow and, as methods change, the proxies will fail, at run-time. As we attempt to upgrade alter the delegates to add a new feature we can cause breakage in several proxy classes.

In summary dynamic proxies are slow, type-unsafe, unpredictable, inflexibilty and unreliable. In fact, the only good thing about proxy classes is that they are transparent and can leave existing code intact.

Static Delegation

A static delegation makes use of a proxy class that forwards messages to delegates. The proxy class is checked out by the compiler, before the program starts to run. In the following two sections we show two types of static delegation, manual static delegation and automatic static delegation. We show how automatic static delegation lower the cost of generating bridge pattern code.

Manual Static Delegation

The implementation of the bridge pattern using manual static delegation requires that a programmer write the message forwarding code in the proxy class *by hand*. Also, the bridge interface must be written by hand. This is an error-prone, tedious and labor intensive task.

We summarize the implementation of the bridge pattern using manual static delegation with our six criteria:

1. Performance - in-line expansion of code (done by the compiler) can make this a zero-overhead solution.
2. Type safety - good type-safety results from the compiler checking before run-time.
3. Transparency - Adding features to a proxy is error-prone and not transparent. Interfaces and delegates must be upgraded, all by hand.
4. Predictability - invocation speed is well known. With in-line expansion available in modern compilers, the speed is as predictable as any method invocation.
5. Flexibility - relationships are flexible, and new methods can be added without changing existing classes.
6. Reliability - A long chain of static proxy invocations should be reliable, under the bridge pattern. Proxy code is easy to understand.

In summary manual static delegation is fast, type-safe, predictable, flexibilty and reliable. In fact, the only bad thing about it is the cost of doing things manually, which means it is not very transparent to the programmer.

Automatic Static Delegation

Automatic static delegation cures the transparency problem of manual static delegation. We use reflection to automatically generate *static* delegation code, even if the original source code is unavailable. This is a new feature, and has not been described in the literature before, as far as we know.

Synthesizing proxy classes automatically reduces the possibility of introducing errors and should encourage programmers to use delegation

more [Johnson]. In summary, automatic static delegation is fast, type-safe, transparent, predictable, flexibilty and reliable.

Analysis of the Tradeoffs

Automatic static proxy class synthesis dominates the other methods of implementing the bridge pattern. It is able to automate the generation of bridge interfaces, as well as bridge implementations. Its' transparancy is matched only by type unsafe dynamic delegation or the non-scalable inheritance. Here are some advantages to the automatic static bridge pattern:

1. The synthesis does not generate arbitrary code.
2. The interface to the instances remains consistent.
3. The delegation is subject to in-line expansion and is more efficient than multiple inheritance.
4. The mechanism for forwarding is obvious and easy to understand.
5. The proxy is coupled to the delegate in a more controlled manner than dynamic delegation.
6. Classes that use the bridge are presented with a stable interface. For example, a method may become deprecated, but changes need only be seen in the proxy class, not its clients.
7. We can lower the cost of software maintenance and improve reusability of the code.

Problems that remain unsolved by static proxy bridge include:

1. Programmers can write arbitrary code in a forwarding method.
2. There is no straightforward way for the delegate to refer back to the delegating object [Viega].
3. Programmers could limit the forwarding message subset (i.e., make the proxy into a facade).
4. The computational context must still be passed to the delegate [Kniesel].
5. The interface is fragile. If the interface to the delegate changes, the forwarding method in the proxy must change [Kniesel 1998].
6. An additional step, the compilation of generated code with static delegation.

In comparison, dynamic proxy classes generate runtime errors, run slower and need no pre-compilation. We favor compile-time errors over runtime errors, and so find our technique superior in this regard. The trade-off is *pay now* or *pay later*.

Conclusion

We have reviewed different techniques for implementing the bridge pattern to reuse legacy code while encapsulating its' complexity.

Deepening subclasses in order to add features is a fast way to create poor code that is very fragile. Subclasses are useful only if the class theoretic approach is appropriate to the domain, and then only if the taxonomic hierarchy is unlikely to change.

Semi-automatic synthesis of bridge code addresses the time-consuming and error-prone draw-back of manual delegation. It is also easier to understand dynamic delegation code. In brief:

1. Dynamic delegation is more automatic than static delegation.
2. Dynamic delegation is not type-safe, but static delegation is.
3. Automatic static delegation is almost as automatic as dynamic delegation, and just as type safe as static delegation.

The basic issue is that a balance must be struck between code reuse and the fragility that arises from *coupling*, a measure of component interdependency. This balance is obtained by good object-oriented design, which we argue can be had by making good use of the bridge pattern.

In brief, the automatic synthesis of proxy classes changes the way we generate bridges to legacy code. We have found that it changes the way we think about production programming and find it a powerful alternative to inheritance.