Foundation Patterns

Dwight Deugo
deugo@scs.carleton.ca
School of Computer Science
Carleton University

Introduction

Although not often mentioned, many patterns depend on one important distinction. This is the distinction between an object's class and its type. Stated another way, patterns rely on interface inheritance rather than on implementation inheritance. Nevertheless, on examining their structures, pattern descriptions usually describe implementation inheritance. This is not surprising, since languages like Smalltalk and C++ do not explicitly support the notion of a type or a subtype within the language. Java is the exception to this, directly supporting interfaces and their inheritance, bringing attention to these well deserving topics.

By making this implicit distinction in their patterns, pattern writers give evidence that there are other patterns, fundamental to many if not all patterns, assumed and undocumented or still waiting to be mined. These patterns live within other patterns. Moreover, I believe they live at the foundations of good object-oriented principles, concepts and object-oriented application development. From the location at which I find these patterns, I give them the name foundation patterns. Foundation patterns are patterns either used by or specialized by other patterns. The capture of these patterns benefits the understanding of all those involved with object-oriented application development and with patterns.

I begin the project with a discussion on two such foundation patterns: **Delegation** and **Substitution**. Other examples of foundation patterns include the **Loop Patterns** [Astrachan and Wallingford, 1998]. The use of delegation will be familiar to many. However, I often find that developers combine delegation and substitution under the single heading of delegation, due to substitution incorporating delegation. In this paper, I separate the two patterns into their rightful positions, making clear what each pattern's role is in object-oriented design.

Delegation Pattern

Intent

Delegation allows objects to share behavior without using inheritance and without duplicating code. Delegation decreases the number of responsibilities of an object and their corresponding implementation sizes. Consequently, delegation minimizes the amount of code a developer is responsible for and the number of bugs he creates. Moreover, successful delegation increases the collaboration between class, thereby increasing the flexibility of the immediate application and the reuse of classes in it and subsequent applications.

Motivation

There are several well-known heuristics that you should follow when designing and developing object-oriented software.

> **One heuristic is to have many objects, each having a small number of responsibilities, rather than a few objects that do everything. Another is to keep the implementation sizes of the responsibilities of an object**

An object that has too many fundamental responsibilities (10, +/- 2), or whose resulting methods are large (more than 10 statements), is usually difficult to understand because it is too complex [Wirfs-Brock, 1990]. Well-designed methods are usually small [Barry, 1989]. Objects with large numbers of responsibilities are problematic for developers to create, maintain, extend and debug, or for others to use. From a developer's point of view, the more responsibilities of an object, the greater the amount of time required to implement them. Moreover, both the original and future developers will need longer to understand the implementation before they can maintain or extend the object. Also, the more lines of code implemented, the greater the probability of introducing a bug. Finally, from a user of an object's point of view, an object with many responsibilities always require more time to understand than one with few.

Consider the situation of implementing an object representing a deck of cards called CardDeck. Two methods required by CardDeck are *shuffle()* and *draw()*. Draw returns the Card positioned at the top of the CardDeck and shuffle places all 52 Cards back into the CardDeck and arranges them randomly. You could use an array of integers of length 52 to represent the internal structure of the CardDeck along with a marker indicating which position in the array represents the top, initially set to 0. Draw returns the corresponding Card represented by the number found in the array at the marker position and increments the marker by one, returning null if the CardDeck is empty. Shuffle resets the marker to the top of the deck and reinitializes the array with random numbers between 1 and 52, using each number in the range only once.

One approach you could take to develop the shuffle method that fills in the array is to grab Donald Knuth's 'The Art of Computer Programming' Volume 2, down from your shelf and implement the suggested stream of pseudorandom numbers using a 48-bit seed which is modified using a linear congruential formula in a loop. However, taking this approach you would find that the shuffle method is rather long and complicated.

Knowing good object-oriented practices, you would refactors *shuffle()* into several smaller methods that *shuffle()* could invoke. However, looking at the methods for CardDeck , you now finds ones such as *setSeed(int), next()* and *nextInRange(int,int)*. You pass the *setSeed(int)* method a seed integer which is required by the pseudorandom algorithm implemented in the method *next()*. Next returns a floating point number in the range 0.0-1.0, used by the method *nextInRange(int,int)*. NextInRange returns an integer in the range 1-52, used to fill in the array of integers representing the Cards.

In hindsight, none of these behaviors, even implemented as private methods, seems as if they belong in CardDeck and might confuse developers either using or changing CardDeck. The problem is that CardDeck has captured two abstractions! The first abstraction is a deck of cards. The second abstraction is a random number generator.

A better solution is to separate the abstractions into two classes and have the first one delegate to the second, shown in figure 1. The RandomNumberGenerator has the behaviors *setSeed(int), next()* and *nextInRange(int,int)* and the internal integer seed variable supporting them. The CardDeck still has the behaviors *draw()* and *shuffle()*, the internal integer marker and the Card array supporting them. However, now *shuffle()* uses the services of the RandomNumberGenerator. Each class supports a smaller interface than the one original, and their behaviors and internal structures are focused on the specific abstractions, making them both easier to use and understand. The earlier refactoring exercise has also helped to keep the implementation sizes of these behaviors at reasonable complexity levels and sizes.

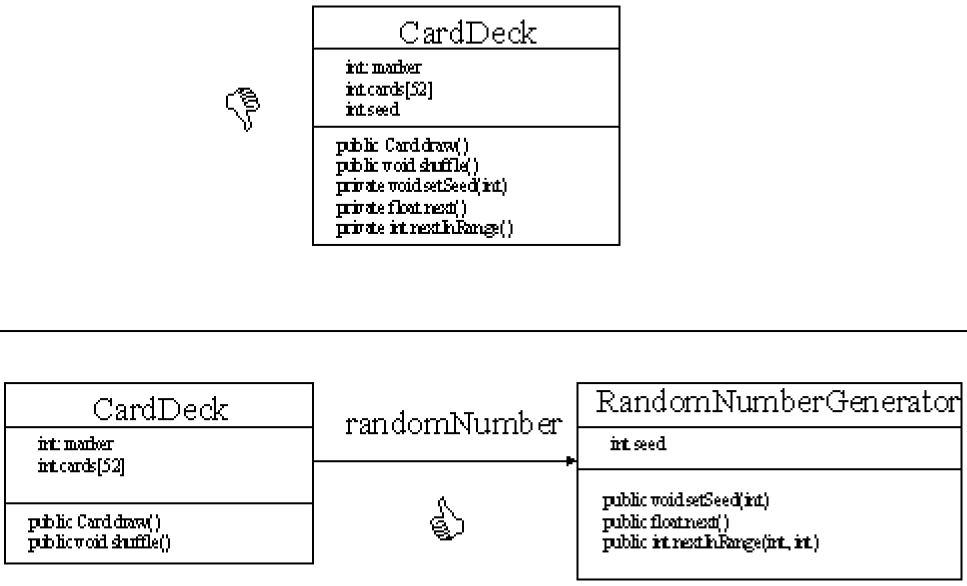

Figure 1 *Refactoring CardDeck*

Another heuristic is not to implement identical behaviors in two or more places. This leads to an obvious maintenance problem: if a developer changes one implementation, he must also do the same for all others or inadvertently introduce bugs into the application.

Objects often have similar responsibilities. How many times have you implemented an object that needed to sort a list of objects, required a random number or needed to maintain and process information about a person's address, client or company? Without good browsing skills or access to a good IDE that supports searching, and the ability to copy and paste, developers often choose one of two techniques for developing new objects and their responsibilities. They develop every behavior from scratch, or they copy the implementation from a previous object to the new one. In either case, the result is the same. The same behavior or code segment is located in two or more places which causes an update synchronization problem. In the latter case, provided sufficient commenting, updating one implementation may draw the developer's attention to the fact that another update is required. However, in the former case no such forwarding exists. To make things worse, multiple developers have spent time doing the same work, which is not something their managers would appreciate.

Consider the case where, in addition to needing a deck of cards, you also needed a Die. The Die has the behavior role, returning a number between 1 and 6. If both the and CardDeck supported their own abstractions and that of a random number generator, there is a problem. If you change the random number generator algorithm in either, there is a good chance it needs to change both. Not only does you duplicated effort in implementing the Die and CardDeck, you also increases the maintenance effort. Effectively, every change needs to be done twice.

As before, the solution is to separate the abstractions into three classes: Die, CardDeck and RandomNumberGenerator, and have Die and CardDeck delegate the responsibility of generating random numbers to the RandomNumberGenerator, as shown in figure 2. To change the algorithm for generating random numbers because, for example, a new one is faster or 'more' random, you need only alter the appropriate RandomNumberGenerator methods. Die and CardDeck automatically incorporate these changes as a result of delegating to the same object.

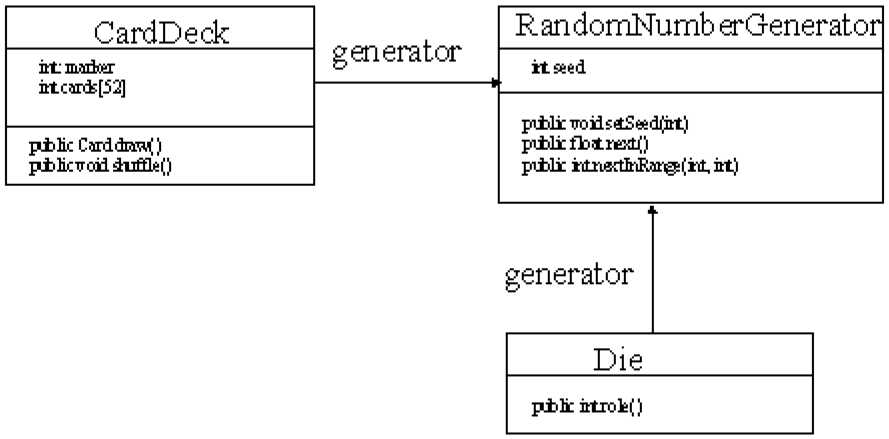

Figure 2 *Reusing RandomNumberGenerator*

All of these heuristics fall under the general one of not reinventing the wheel. If some other object knows how to do something, use it! Delegate responsibility.

You must always remember that you are not the first one to develop objects. There is a long line of developers before you that have implemented objects with similar, if not identical, responsibilities. I don't know how many times I have seen a 'new' implementation of a Sorter, a RandomNumberGenerator, a Company, a Profile, a Broker or a Name class. The reality of object development is that many good objects, frameworks, and **patterns** already exist. It may sound strange, but I would rather have members of my development team use other developer's objects than write their own. Every line of code they don't write, is one less line containing a potential bug for the team to fix. The conflict is that you need to develop new objects, which involves coding, but if you write less code, you create fewer bugs. To resolve the conflict you can have your objects delegate their responsibilities to other developer's objects. Let the bugs show up in their code. If your objects work well, you look good. If your object doesn't work because of a problem with an object you delegated responsibility to, it is someone else's problem. You still look good. In addition, using other people's classes encourages them to produce better quality code. Strong identifiable ownership for code, usually brings with it quality. No one ever wants to be responsible for inferior code. And, if bugs do occur and are fixed, more developers benefit.

Consider the previous solution of having the Die and CardDeck objects delegating the responsibility for generating random numbers to the RandomNumberGenerator. However, this time, I search the existing classes in the standard Java packages and find that a random number generator class already exists, called **java.util.Random**. Rather than having to write, maintain, update and debug my own RandomNumberGenerator class, I remove it completely and have Die and CardDeck delegate to java.util.Random. The two objects Dice and CardDeck still work the same, but I am now responsible for one less class.

The general delegation solution has many objects delegating partial or full responsibilities for one of their services to many other objects. This solution does not advocate that every object delegate to every other object. Rather, is suggests that there are those types objects, call them Subcontractors, that are created with the purpose of servicing many others. And, there are other types of objects, call them ServiceProviders, that service few but use many of the services of others.

As a manager, you struggle to find developers that you can trust to help you with your work. As a developer, you should struggle to find objects that you can trust to help you with your development. The search is worth the effort. The reward is not having to write and maintain as much code.

Applicability

Use the Delegation pattern when:

- **An object contains many methods supporting more than one abstraction.** Break the object into multiple objects, each containing a single abstraction and have one delegate to the other.
- **You want to factor out and share common behavior between different classes.** Don't put the same implemented behavior in two or more classes, it causes an update problem. Put the behavior in one class and have the others delegate responsibility to it.
- **A class already exists that can service a request.** Never implement classes that already exist. Delegate.
- **You want to be responsible for less code.** If you can, rely on other people's code. You will get the same job done and not generate any new bugs.
- **You want to share implementation and the inheritance mechanism will not provide it.** Since inheritance is a code sharing mechanism, it is possible to write a behavior once in a superclass and have all of its subclasses inherit the behavior. However, often classes that need the behavior are not related by inheritance. Rather they are associated by aggregation or by reference. In these situations, you can not use inheritance to share the behavior, it must be delegated
- Y**ou want to build flexible, adaptable, non-brittle classes.**

Structure

Shown in figure 3.

Participants

- ServiceProvider (CardDeck)
  - ⇒ identifies one or more service responsibilities for client objects to request.
  - ⇒ knows which subcontractors to use to handle its service requests.
- Subcontractor (RandNumberGenerator)
  - ⇒ identifies and implements one or more subservice behaviors that are potentially requested by many different ServiceProviders.

Collaborations

1. A Client requests a service from a ServicePovider.
2. The ServiceProvider delegates the responsibility to one or more Subcontractors, collates their results and returns the final result to the Client.

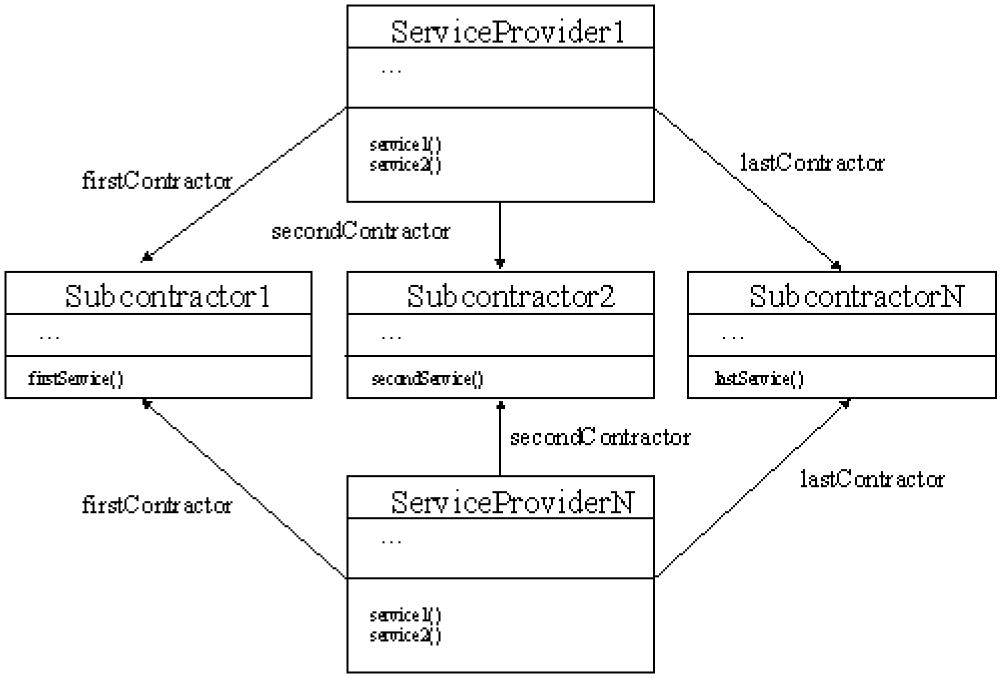

Figure 3 *Delegation Structure*

Consequences

The Delegation pattern has the following advantages:

- **It makes objects easier to understand, develop and maintain.** Delegation forces the separation of an object containing multiple abstractions into multiple objects containing single abstractions. The refactoring of behavior into two or more objects results in a decreases in the number of methods found in any one object compared to the original. Also, refactoring accounts for a decrease in method sizes, as methods now contain mostly messages sent to other objects, rather than elaborate computations. These decreases make objects not only easier to understand, but also to develop and maintain.
- **The size of an application using delegation is less than one not using it.** With different ServiceProviders delegating to common Subcontractors, not only do you achieve better reuse, there is a decreases in the overall amount of code. This is a result of not replicating common code in each ServiceProvider. Rather, the common code is located in only one location – one object. The more ServiceProviders that use common Subcontractors and the more combinations of delegated services, the greater the overall decrease in the amount of code.
- **It decreases the potential for bugs.** I argue that as the amount of code decreases, so does the potential for bugs. In addition, because common behavior is factored into one place, if it is in error, it only needs one correction for the overall impact of the fix to be felt everywhere. If the behavior was replicated in several places, finding an error in one place requires the diligence of the developer to ensure all other cases have also been fixed, which is not always guaranteed.
- **Less documentation is required.** Objects that are easy to understand and use require less documentation. This saves development time for both past and future developers. Past developers do not need to write as much documentation and explain their objects to others in the future, and future developers do not need to read as much documentation and ask past developers so many questions.

- **It enables the use of the Substitution pattern.** Without delegation, it is impossible to support dynamic substitution of different subcontractors, leading to static, brittle objects and applications.

The Delegation pattern has a few disadvantages:

- **It increases the number of messages between objects in the system.** Many people express difficulty in understanding and following the computation resulting from message sends between collaborating objects to satisfy a service. I believe this has more do to with experience than a real issue in complexity. However, messages do have a cost: they cost time to process. Therefore, the misuse of delegation can lead to spaghetti code that is difficult to understand and has poor performance, out-weighing the advantages mentioned above. Typically though, noticeable performance problems are an indication of an overuse of delegation. Wise use of delegation rarely causes performance problems and often makes performance problems easier to locate and fix.
- **In some situations, it may increase the number of objects in the system.** As individual objects are broken into smaller ones, there is an obvious increase in the number of objects in the system. Objects need to be managed, version numbers need to be assigned, and often compatibility and synchronization issues arise. The greater the number of objects, the greater the time spent on these tasks and issues.
- **Taken to an extreme, delegation results in objects that do not represent an abstraction!** Taking a single object with twenty behaviors and breaking into to twenty collaborating objects each with one behavior is delegation overload! Instead of one object with multiple abstractions, you now have twenty objects with no abstractions. A single behavior does not usually make for a good object. Twenty behaviors for an object are too many. No more than ten responsibilities is a good target for a well-designed object. However, the 10 by 10 rule (ten responsibilities and methods of no longer than 10 statements) should be considered as a measuring stick. There is always balance to strike.

Implementation

- **Subcontractors as Singletons**. In **rare** situations, a Subcontractor does not keep track of any new information, but only provides computational services on existing data. In this case, it can be implemented as a singleton and associated with ServiceProviders by reference or temporary association. This minimizes the number of Subcontractors instantiated, improving both space and time efficiency.
- **Not every ServiceProvider requires all Subcontractors**. Different ServiceProviders may need the services of different Subcontractors. There are no restrictions placed on ServiceProviders on who they delegate responsibilities to.

Sample Code

Here is a partial Java implementation of two ServiceProviders and two Subcontractors. ServiceProviderA requires the services of both Subcontractors, while ServiceProviderB requires only SubcontractorA. Also, both ServiceProviders require that they be given the objects to delegate responsibility to when being constructed.

```
public class ServiceProviderA extends Object {
    private SubcontractorA  firstSubcontractor;
    private SubcontractorB secondSubcontractor;
public ServiceProviderA (SubcontractorA  firstHelper, SubcontractorB secondHelper) {
            firstSubcontractor = firstHelper;
            secondSubcontractor = secondHelper;}

    public String serviceRequest() {
            return firstSubcontractor.subservice1() +
                    secondSubcontractor.subservice2();}
}

public class ServiceProviderB extends Object {
    private SubcontractorA  firstSubcontractor;

    public ServiceProviderB (SubcontractorA  firstHelper) {
            firstSubcontractor = firstHelper;}

    public String differentServiceRequest() {
            return firstSubcontractor.subservice1();}
}
```

The following two Subcontractors provide the subservices.

```
    public SubcontractorA extends Object {

            public String subservice1() {
                    …;}
    }

    public SubcontractorB extends Object {

            public String subservice2() {
                    …;}
    }
```

Related Patterns

Delegation is a pervasive pattern. Many patterns use delegation, which is a reason for it being identified as a foundation pattern. I mention the following patterns as good examples of patterns incorporating the delegation pattern: Whole-Part [Buschmann, 1996a], Proxy[Gamma, 1995a] and Master-Slave [Buschmann, 1996b].

Known Uses

A good example of a pattern using delegation, is the Master-Slave pattern. In the Master-Slave pattern, a Master distributes work involved with one of its services to Slave components, computing its final result from the results of the Slaves. A Master could implement its Slaves' responsibilities as its own, but would then contain multiple abstractions. Instead, the Master-Slave pattern separates the abstractions into a Master and a Slave, enabling not only three variants for fault tolerance, parallel computation and computational accuracy, but also the potential of different Masters sharing the same Slaves.

Substitution Pattern

Intent

Substitution enables an object to change its implementation using composition rather than by significant coding or relying on inheritance. It also weakens the dependency between collaborating objects to one based on type rather than on class. Consequently, substitution permits objects to alter (or compose) their implementations on-the-fly at run-time, rather than strictly at compile-time. This ability increases the range of behaviors an object can exhibit without increasing its implementation size and decreasing the number of classes in an application overall.

Motivation

In addition to the heuristics noted in the discussion of the Delegation pattern, developers should consider several the following observations when designing and developing object-oriented software.

Developers do not always need to handle unfolding and modified requirements by force-fitting their solutions and changes into the inheritance hierarchy.

Particular domains require the specialization of one type of class many times. For most developers, this means creating subclasses of the original and implementing the specialized behaviors in them. However, when many specialized classes are required, it can be very difficult to determine the complete set, even more difficult to determine how to implement the specialized behaviors, and very complex to figure out how to arrange the behaviors and classes in an inheritance hierarchy to get the best level of reuse. Of course, as new conditions arise and requirements change other new specialized classes are required, forcing you to experience these difficulties again.

Developers must not subject users to unwanted recompilations and redeployments of an application. Instead, developers need to consider allowing users to configure their own objects at run-time, rather than forcing them to accept only those create at development time.

By using inheritance to share implementation, you must provide a different class for each specialization. This forces you to compile every time a new class is added or when an existing class is modified. In addition, anytime you want to share behaviors between classes, you must recompile the class the behaviors are added to and, even worse, compile those classes who have had their inheritance relationships changed to use the shared behavior. This means that changes can only occur at compile-time. However, many changes are often a result of situations that arise after an application has been deployed, where compiling may not be an option.

Take for example a bank account. Most banks have many different types: saving, personal, checking, stock, mutual fund, register retirement saving, personal self directed, register retirement self-directed saving, and more! In addition, some accounts share interest calculations, others don't. Some have service charges, other's don't. All have different combinations of features, such as free web access, and require different minimum balance to enable free features.

You could decide not to use inheritance or delegation when implementing the `Accounts`. Taking this approach results in approximately the same number of classes as with inheritance.

However, since there is no longer any code sharing through inheritance, different Accounts will duplicate behavior, which creates an update problem.

The short-term solution is to use the Delegation pattern. By abstracting common behavior into Subcontractor objects, and having the Accounts delegate to them, at least you solve the update problem. However, you still have the original problem. If you want to develop a new type of Account or change an existing one, you either have to directly add or change the behavior of it, have it delegate to a different Subcontractor, or change the behavior in the Subcontractor. Therefore, whether using inheritance or delegation to share behavior, you must compile a class every time you need to change its behavior, either from the point of view of the ServiceProvider or the Subcontractor.

In the banking industry, new types of Accounts come and go at the frequency of changes in the weather. In order to keep pace, banking applications can not go through redevelopment and recompilation every time a new Account or change to an existing Account is required. The solution to this problem is to configure Accounts - assembled them by composition - rather than to developed them from scratch. Delegation is part of the solution, it enables you to build an Account using different objects, but to complete the solution you need the ability to perform substitution for those objects at run-time.

Similar to Delegation, you develop Subcontractor objects for your ServiceProvider object – Account - to delegate responsibilities, as shown in figure 4. However, rather than one Subcontractor, you develop multiple Subcontractors having the same interface – objects that implement identical methods, at least in signature but not necessarily in computation. Since these Subcontractors have identical interfaces, an Account can delegate to anyone of them, provided it relies on only the interface, not on the class of objects. And, since the Subcontractors are different classes that can vary their behaviors, when substituted for one another, the result is a variation in the behavior of the delegating object. In the banking application, you can repeat the exercise, developing different Subcontractors supporting interest, service charges and other related abstraction interfaces. Having Subcontractors support similar interfaces and being able to be substituted for one another, you can assemble different combinations to represent an Account, creating a wide range of behaviors representing different accounts from a single Account class.

This approach has many benefits. The first is that at either at development-time, or more importantly, at run-time you can vary the composition and resulting behavior of an Account. The second is that you only need one Account class. Rather than specializing different Accounts, you represent different Accounts as different assemblies of Subcontractors. Third, you have reduced the number of classes need to represent Accounts. In the original approach, let's say you had twenty different bank accounts. You would need twenty different Account classes. Using delegation and substitution, you can conceivably cut this number in half. One class for Account, four classes implementing an Interest interface and five classes implementing the ServiceCharge interface. The one Account can contain any one Interest subcontractor and any one ServiceCharge contractor, which gives you twenty different combinations, each representing a different Account. Would you rather be responsible for twenty classes or ten? The answer is obvious!

Applicability

Use the Substitution pattern when:

- **You need to develop many similar types of objects that do not fit into a well-structured hierarchy and their enumeration is difficult to determine a priori, making their implementation problematic at best.** It is often difficult to predict all variations of specialized objects. And, although they vary in behavior, in some cases arranging them into a hierarchy results in code duplication in different branches. Rather than subclass, use delegation and substitution to assemble the object from others to generate the correct behaviors.

- **You need to dynamically change or configure an object's behavior at run-time.** You can change or share behaviors between classes by altering one class' composition or by using inheritance**.** Since inheritance relationships can change only at compile-time**,** by using substitution and changing an object's composition, who and what an object delegates can be changed at run-time.

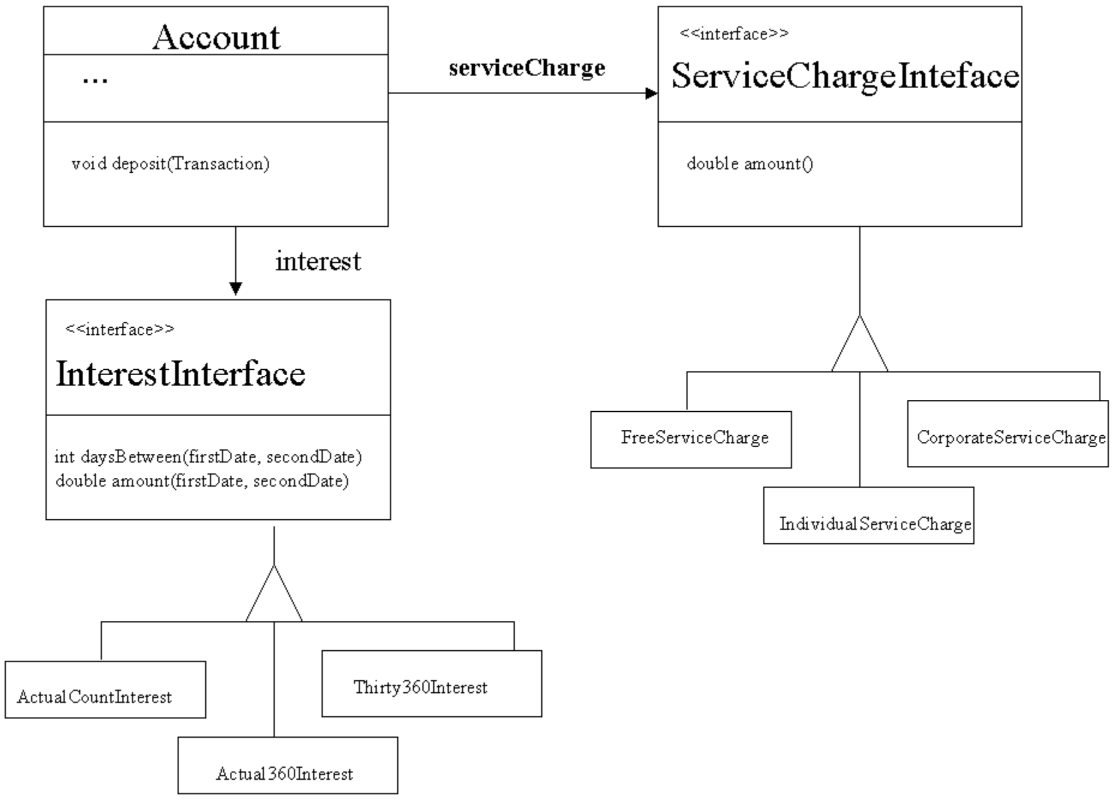

Figure 4. *Account Class Diagram*

- **You want to support many similar objects in as maintainable a fashion as possible.** Combined with delegation, substitution allows one object to have different behaviors resulting from different combinations of others used in its composition. The benefit to the developer is that the overall design is easier to understand as there are fewer classes needed to provide the same range of behaviors had the original object been implemented once for each specialization. Fewer classes also imply a decrease in maintenance costs.
- **You want to loosen the dependency between objects.** The weaker or fewer dependencies between objects, the easier it is to use them in other applications. The reason is that when you use an object in a future application, you must also include the objects it depends on though inheritance or by association. If the dependency is on interface, rather than on class, only one object supporting the interface must be used in the future application, not all. The decision is yours to make. It is not forced upon you.
- **You want to build flexible, adaptable, non-brittle classes**

Structure

Shown in figure 5.

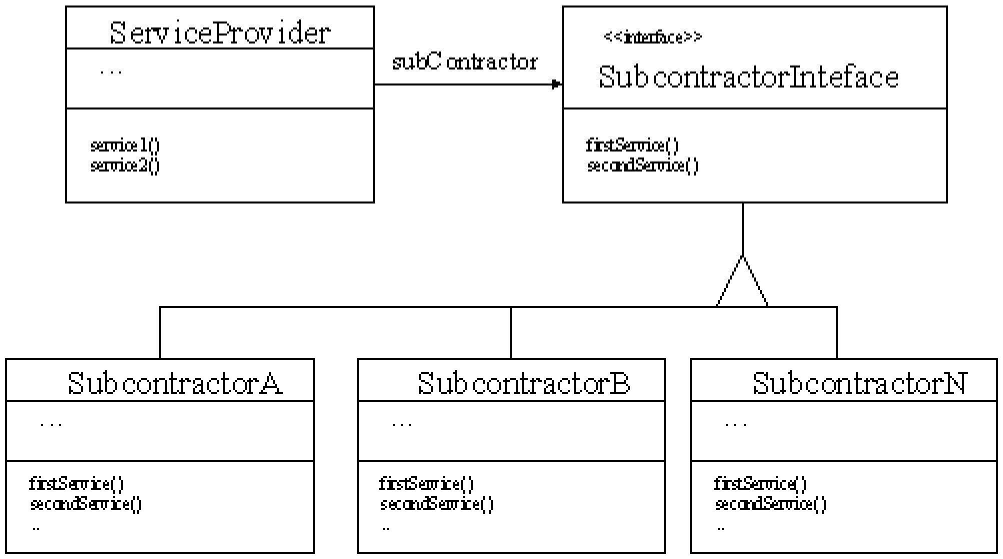

Figure 5. *Substitution Structure*

Participants

- ServiceProvider (Account)
  ⇒ identifies one or more service behaviors for Cient to request.
  ⇒ is assigned one or more Subcontractor to help handle service requests.
- Subcontractor Interface (InterestInterface, ServiceChargenterface)
  ⇒ identifies subservice behaviors that are requested by many different ServiceProviders.
- Subcontractors (FreeServiceCharge, CorporateServiceCharge, IndividualServiceCharge)
  ⇒ implements one or more subservice behaviors identified in an interface that are requested by many different ServiceProviders.
  ⇒ assigned to and invoked by the ServiceProvider.

Collaborations

1. Before a ServiceProvider requires the help of any Subcontractor, it is provided with one that implements the required interface.
2. The ServiceProvider delegates responsibility to the Subcontractors – independent of their classes - collates the results and returns the final result to the Client.

Consequences

The Substitution pattern has several advantages.

- **It prevents bloated interfaces and keeps them more static.** It is always difficult to predict beforehand new ways of servicing a request and on which parameters will play a role in the service. Taking an approach that considers each variation as a different request continually increases a ServiceProvider's interface, leading to the bloated interface. Using substitution, the ServiceProvider's interface remains static, but still enables the ServiceProvider to assemble the desired behavior.

- **It allows you to derive objects by composition rather than by implementation.** Rather than increase the number of behaviors an object has to handle different service request conditions, a better technique is to look at the potential causes of those variations, configure and have the ServiceProvider delegate the responsibility for the service to the appropriate Subcontractors. This allows you to vary the Subcontractors instead of the ServiceProvider in order to generate the desired behavior. New conditions may lead to the production of new Subcontractors, but the ServiceProvider needs only to vary its associations with Subcontractors, not its implementation, to service the new requests.
- **It allows you to dynamically alter an object's behavior.** Without substitution, you can only compose an object of specific classes of objects, which limits the possible combinations. With substitution, an object can be composed of many different classes of objects, provided these objects implement the appropriate interfaces. In addition, since different types of objects can be used for the same purpose, they can be changed at anytime. In many domains, especially the financial industry, domain objects need to respond differently to service requests almost daily. It is impractical for you to model such a domain or develop applications for it by forcing a recompilation with every change in service. Rather you should build applications that permit the assembly of objects on-the-fly to provide the appropriate behavior. This is only possible when objects use delegation combined with substitution. The combination provides the potential for dynamically changing or configuring ServiceProviders with Subcontractors.

The Substitution pattern has several disadvantages.

- **Object using Substitution can be difficult to understand.** As mentioned by Gamma [Gamma, 1995, p 21] 'highly parameterized software is harder to understand than more static software'.
- **Due to substitution relying on delegation, it has delegation's main disadvantage**: an increase in the number of messages sent between objects in the system.
- **When using substitution at run-time, you must provide a configuration mechanism.** Therefore, the flexibility substitution gives you does not come free. However, the cost of the alternative (creating new classes by coding) may be prohibitively expensive in an environment where you must provide new variations quickly. You must take care to avoid creating a configuration mechanism that does not become unwieldy or too complicated.

Implementation

- **Substitution relies on interface not class**. Many existing patterns use substitution and have their structures use abstract classes to describe the interfaces of the corresponding Subcontractors. The abstract Subcontractors are subclassed to provide specific subservice implementations. It is important to recognize that this is an implementation detail, not a structural detail. A ServiceProvider does **not have to** rely on its Subcontractor being a specific class or subclass. Rather, it relies on a Subcontractor supporting a specific subservice interface.
- **In Java**, interfaces are first class entities. In Java, you can define an interface and have different objects implement it. This is an important feature since it identifies immediately to developers which objects can be substituted for one another. Since we don't want ServiceProviders to rely on the subclasses of a Subcontractor, only the type of service

they provide, it is important that we define interfaces in Java for ServiceProviders to use and for Subcontractors to implement.

- **In an untyped language like Smalltalk,** interfaces are usually in the minds of the developers. In one sense, Smalltalk is the ultimate language for substitution, any object can be substituted for any other. However, there is no guarantee that the substituted object will respond to the same messages supported by the original. The developer must verify this fact. He can gain some confidence by ensuring that the substituting objects inherits from a common superclass and program to its interface, but this check is not automatic. This is one case where I like strong typing. In Java, it would detect this form of error at compile-time. Whereas in Smalltalk, the error would only show up at development-time if the testing was good.

Sample Code

Here is a partial Java implementation of one ServiceProvider and four Subcontractors. In this example, the ServiceProvider handles its serviceRequest by delegating to two Subcontractors. The ServiceProvider is not dependent on the specific classes of Subcontractorsbut, rather, on two Subcontractor interfaces: FirstSubcontractorInterface and SecondSubcontractorInterface. When the ServiceProvider is constructed, it requires the objects that implement these interfaces in order to delegate responsibilities later.

```
public class ServiceProvider extends Object {
    private FirstSubcontractorInterface  firstSubcontractor;
    private SecondSubcontractorInteface secondSubcontractor;

    public ServiceProvider (FirstSubcontractorInterface firstHelper,
    SecondSubcontractorInterface secondHelper) {
            firstSubcontractor = firstHelper;
            secondSubcontractor = secondHelper;}

    public void serviceRequest() {
            firstSubcontractor.subservice1();
            secondSubcontractor.subservice2();}
}
```

The following interfaces represent the two different subservices that concrete Subcontractors must provide.

```
public interface FirstSubcontractorInterface {
    void subservice1();
}

public interface SecondSubcontractorInterface {
    void subservice2();
}
```

The difference with the delegation pattern code is that multiple classes now implement the Subcontractor interfaces. The following two Subcontractors provide the subservice noted in the interface FirstSubcontractorInterface. Therefore, they can be used interchangeably by the ServiceProvider, although they are not subclasses of one another.

```
public SubcontractorA extends Object implements    FirstSubcontractorInterface {

      public void subservice1() {
                  …;}
}

public SubcontractorB extends Object implements    FirstSubcontractorInterface {

      public void subservice1() {
                  …;}
}
```

Moreover, the following two Subcontractors provide the subservice noted in the interface SecondSubcontractorInterface. Therefore, they too can be used interchangeably by the ServiceProvider. However, in this case the two Subcontractors are arranged as subclasses. This is only an implementation detail, of which has no effect on how the ServiceProvider uses them.

```
public SubcontractorL extends Object implements SecondSubcontractorInterface {

      void subservice2() {
                  …;}
}

public SubcontractorM extends ContractorL implements SecondSubcontractorInterface {

      void subservice2() {
                  …;}
}
```

Related Patterns

Substitution is a pervasive pattern. Many patterns use substitution, which is a reason for it being identified as a foundation pattern. I mention the following patterns as good examples of patterns using the substitution pattern: Strategy [Gamma, 1995], Bridge[Gamma, 1995] and Acyclic Visitor[Martin, 1998].

Known Uses

A good example of a pattern using substitution, although many of the GoF patterns make use of it, is the Strategy pattern. In the Strategy pattern, a Context object delegates a responsibility to a Strategy object, which implements a specific algorithm. All Strategy classes implement the same interface, enabling the Context to swap one Strategy for another at run-time, which allows the Context to vary the algorithm it uses.

The structure of the Strategy pattern shows different Strategy classes arranged in a hierarchy. This is reasonable approach in order to enable Strategy classes to share code, although it is not a fundamental aspect of the pattern. This arrangement only ensures that every Strategy has a default implementation for the method that the Context expects the Strategy to have. The fundamental aspect of the pattern is that Strategies have the same interface, from the Context's perspective.

Acknowledgements

I want to thank Ken Auer who went beyond the call of duty in shepherding early versions of this paper. His comments not only made the paper better; they made the patterns clearer to me.

References

O. Astrachan and E. Wallingford, (1998). ‘Looping Pattners’, Proceedings of The Fifth Annual Conference on the Pattern Languages of Programs, Group 7, Paper 7.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal (1996a). 'Pattern-Oriented Software Architecture: A System of Patterns', New York: John Wiley & Sons, 225-242.

B. Barry (1989). 'Prototyping a Real-Time Embedded System in Smalltalk', Proceedings of OOPSLA '89, New Orleans, ACM SIGPLAN.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal (1996b). 'Pattern-Oriented Software Architecture: A System of Patterns', New York: John Wiley & Sons, 245-260.

E.Gamma, R. Helm, R. Johnson, and J.Vlissides (1995a). 'Design Patterns: Elements of Reusable Object-Oriented Software', Reading, MA: Addison-Wesley, pp. 207-217.

E.Gamma, R. Helm, R. Johnson, and J.Vlissides (1995b). 'Design Patterns: Elements of Reusable Object-Oriented Software', Reading, MA: Addison-Wesley, pp. 315-323.

E.Gamma, R. Helm, R. Johnson, and J.Vlissides (1995c). 'Design Patterns: Elements of Reusable Object-Oriented Software', Reading, MA: Addison-Wesley, pp. 151-161.

R. C. Martin (1998). In Pattern Languages of Program Design 3, Eds. Robert Martin, Dirk Riehle and Frank Buschmann, Addison-Wesley, Chapter 7.

R. Wirfs-Brock, B. Wilkerson, L. Wiener (1990). 'Designing Object-Oriented Software', Prentice Hall.

W. Zimmer (1995), 'Relationships Between Design Patterns', Pattern Languages of Program Design, Eds. J. O. Coplien, D. C. Schmidt, Vol. 1, Addison-Wesley, Chapter 18.