COMP 401

Prasun Dewan[1]

22. Delegation

Delegation is an alternative to inheritance for reusing code among multiple classes. Inheritance uses the IS-A relationship for re-use; delegation uses the HAS-A reference relationship to do the same. Inheritance and delegation have the same kind of relationship that, say, Aspirin and Tylenol, have. Both are alternatives for fixing a problem – Tylenol and Aspirin a headache, and Inheritance and Delegation code duplication - with one more appropriate than the other in some situations.

In this chapter, we will study the nature of delegation, see how we can convert an inheriting class to a delegating one, compare the advantages and disadvantages of the two approaches, and identify scenarios in which they should be used.

Constructs Gradually Falling Out of Favor

Inheritance is a very attractive concept. It distinguishes an object-oriented language from an object-based one. An object-based language supports only objects and classes, while an object-oriented supports objects, classes, and inheritance. Object-oriented languages have had much more impact than object-based language because programming language inheritance corresponds to the real-life concept of inheritance, which made it, for, many a "natural" solution for the code duplication problem.

The point of this chapter is that it is easy to go overboard in the use of inheritance -- the history of object-oriented libraries is replete with examples of its overuse. In particular, the Java 1.0 libraries were mostly inheritance based. After facing real problems with this approach, the Java designers decided to convert many of the inheritance-based libraries to delegation-based ones in Java 1.1.

The history of computer science is also full of examples of constructs that have been embraced enthusiastically for a while and then fallen out of grace. If you have used Basic or FORTRAN, you know about the Go To statement, which allows a program to transfer jump to some arbitrary labeled location in the code. It tremendously increases the flexibility of a programmers, but also makes the code less structured, and hence, difficult to understand. Way back in 1968, Dijkstra, a very smart and far sighted computer scientist, wrote the seminal article "Go To Statement Considered Harmful." This article did not have an immediate impact on computer science. However, gradually, people started moving from

.

Go To statements to more structured constructs such as loops. For example, Pascal, a language designed by another very smart computer scientist, Niklaus Wirth, supported Go To statements, even though Wirth recommended against its use. Java, a much more modern language, has taken the stop of not even supporting Go To statements.

After rallying against Go To statements, computer scientists started attacking un-encapsulated data types, which are essentially objects with public variables. Every textbook and professor will tell you never to create such variables. However, language designers have not been brave enough to remove such variables from languages.

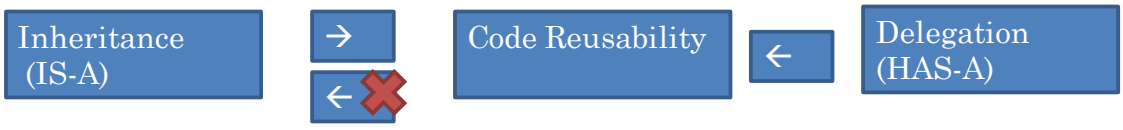
Another popular practice that is now frowned on is using classes to type variables. In languages with interfaces, as we have seen several times so far, using interfaces to type variables leads to more reusable code. Unfortunately, as we have seen in the design of Java libraries, the default, for many programmers today, is to still use classes as types, and convert to interface types when they face reusability problems.

In all of these examples, the tension between practice and recommendations from purists arises because in the short run, the recommendations increase programmer effort, though in the long term, they reduce software costs.

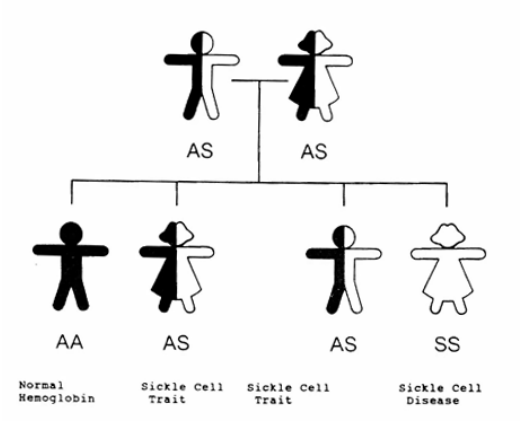
Inheritance is another construct that can, in the short term, reduce programmer effort, but in the long term, result in brittle code that must be re-factored to accommodate changes. However, the argument against inheritance is much more subtle. The recommendation here is not ban it from programs, but us it only when it is appropriate to do so. Thus, this chapter could be titled "Indiscriminate use of inheritance considered harmful."

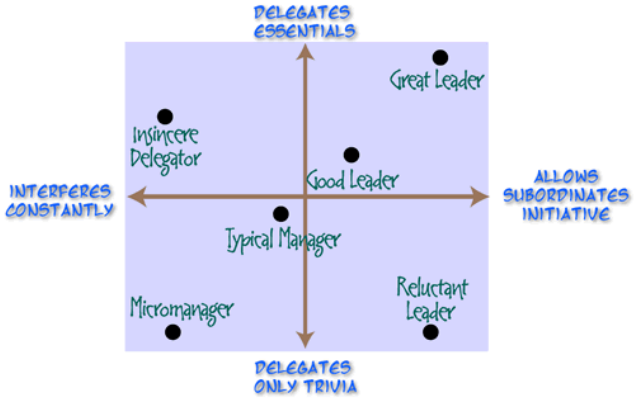
Inheritance vs. Delegation

The big reason, as mentioned, before, for using inheritance is sharing of code among multiple classes. However, as shown in the figure, while inheritance implies code reusability, code reusability does not require inheritance. Delegation is an alternative approach to allow multiple classes to share code. In the case of inheritance, a reusing class has an IS-A relationship with a reused class. Thus, it inherits code from the reused class. In the case of delegation, the reused class HAS-A reference to the reused class. This reference allows it to delegate tasks to the reused class.



Just as computer inheritance corresponds to the real-world concept of inheritance, computer delegation corresponds to the real-world concept of a worker delegating a task to another. The figure shows real-world examples of these two concepts.





Based on these examples, inheritance and delegation seem like unrelated concepts –like apples and oranges. In fact, they are closely related, as illustrated by the following example: A basketball player can inherit the skills to get a basket in a certain situation, or can pass to someone, that is, delegate to, someone who has the skills. Similarly, a class can inherit some functionality or delegate to a class that has the functionality, possibly through inheritance.

This is all very abstract, so let us take an example to better understand and compare these two concepts.

Inheriting vs. Delegating Observable Counter

In the chapter on MVC, we started with a simple, unobservable, counter.

```
public class ACounter implements Counter {
        int counter = 0;
        public void add (int amount) {
                counter += amount;
        }
        public int getValue() {
                return counter;
        }
}
```

We then presented an observable version of it.

```
public class AnObservableCounter implements ObservableCounter {
        int counter = 0;
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                counter += amount;
                notifyObservers();
        }
        public int getValue() {
                return counter;
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        public void removeObserver(CounterObserver observer) {
                observers.removeElement(observer);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

The observable counter class duplicates code in the regular counter class. Let us use this simple example to compare the inheritance and delegation approach to removing code duplication.

By now, we know enough about inheritance to easily create the observable counter class as a subclass of the regular counter class.

```
public class AnObservableCounter extends ACounter implements
ObservableCounter {
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                super.add(amount);
                notifyObservers();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        public void removeObserver(CounterObserver observer) {
                observers.removeElement(observer);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

As we see above, the reusing class, the observable counter, has an IS-A relationship with the re-used class, the regular counter. As mentioned above, in the delegation approach, the reusing class HAS-A reference to the reused class. In our example, this means that the reusing class has an instance variable that stores a reference to an instance of the reused class (or any other class implementing the Counter reference). The following code shows how this reference is used for code reuse.

```
public class AnObservableCounter implements ObservableCounter {
        Counter counter ;
        public AnObservableCounter (Counter theCounter) {
                counter = theCounter;
        }
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                counter.add(amount);
                notifyObservers();
        }
        public int getValue() {return counter.getValue();}
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        public void removeObserver(CounterObserver observer) {
                observers.removeElement(observer);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

The delegating class implements the same interface as the inheriting class. It implements the getValue() method by simply forwarding the call to the reused class. It implements the add() method by forwarding or delegating the call to the reused class, and then notifying the observers. Thus, the behavior of this class is identical to the behavior of the inheriting and non-inheriting versions of the observable counter.

Using the HAS-A relationship to reuse code is called delegation as calls to reused methods are delegated to the reused class.

Inheritance vs. Delegation in General

Let us go beyond this example and compare in an abstract manner the two approaches. As we will see later, delegation allows the reused class to have an arbitrary interface – not one that is a subtype of the interface of the re-used class. In such situations, the inheritance and delegation approaches cannot be compared in that they are not alternatives. Therefore in our abstract comparison, we will assume that the interface of the re-used class is indeed a subtype of the interface of the reused class.

Let us assume that we would like to create two classes, S and U. S contains some code that is shared or reused by U and other classes. The interface implemented by U is an extension of the interface implemented by S. In the inheritance-based solution, U IS-A S. In the delegation-based solution, U HAS-A S.

Inheritance vs. Delegation using OOM

The figure below illustrates the difference between the inheritance and delegation approaches (when they are alternatives) using the standard (Object Modeling Technique) conventions for depicting the IS-A and HAS-A relationships: arrows with triangles and diamonds, respectively. It also uses the non-standard convention of showing the implements relationship as arrows with squares. Because Microsoft Office does not make it convenient to create such arrows, we will not use these conventions in all examples, relying instead on labeled arrows to differentiate between different relationships.

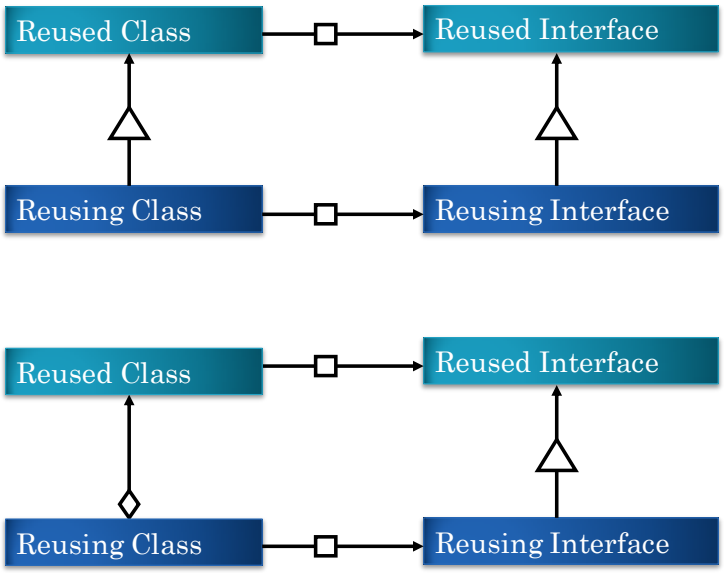


Figure 1 Delegation as an alternative to inheritance

In both cases, a class is reused by extending its interface. The difference is in the relationship between the reusing and reused class: IS-A, in case of inheritance and HAS-A in case of delegation.

Let us look below at some of the consequences of this difference.

Single vs. Multiple Composed Instances

In inheritance, a single instance, of U (the reusing class), is created for the members (variables and methods) defined in U and S (the reused class). In delegation, two instances, of U and S, called the delegator and delegate, are created, for the members of U and S, as shown in the figure below.

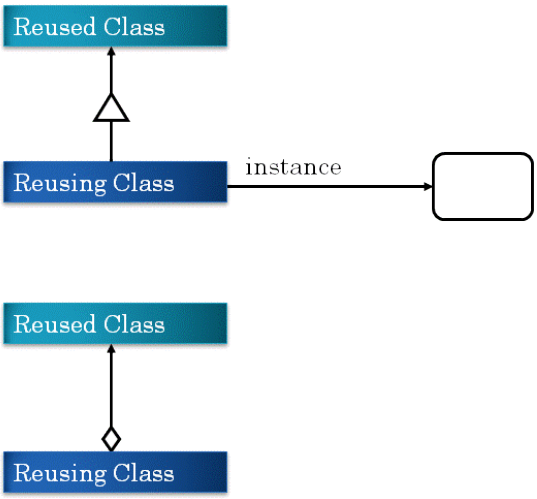


Figure 2 Single vs. Multiple Instances in Inheritance and Delegation

The square box with rounded corners is the standard notation for instances. In the figure it shows that the delegate is a physical component of the delegator. The delegate must be bound to the delegator through a constructor, initialization method, or some other form of initialization, while this step is not necessary in the inheritance approach.

This is illustrated by the constructor of the delegating counter:

```
public AnObservableCounter (Counter theCounter) {

        counter = the Counter;

}
```

The inheriting class defined no constructor as it did not need an equivalent step.

Automatically reused methods vs. delegating stubs

In the inheritance approach, if U does not override some method, m. of S, then in the delegation approach U must defined a "stub" for method m that does nothing else but call method m of U, as shown in the figure below. The method getValue() in the delegating proxy is an example of such a stub:

```
public int getValue() {
        return counter.getValue();
}
```

This is shown abstractly in the figure.

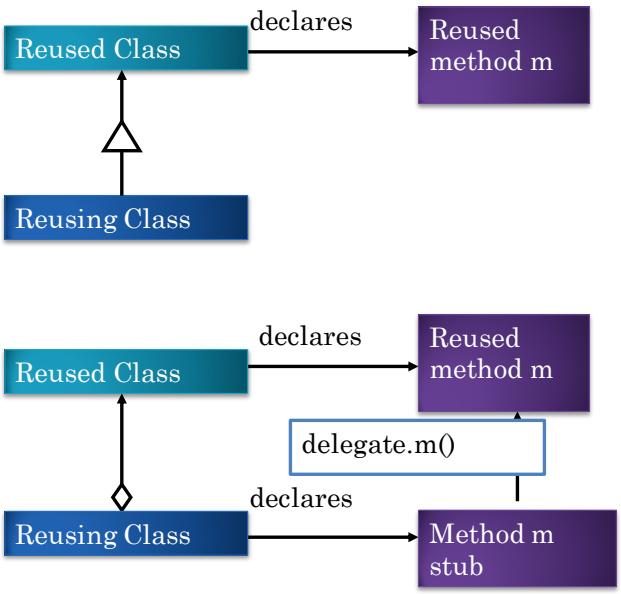


Figure 3 Delegation requires stubs for methods not overridden in the inheritance approach

The task of defining stubs can be very tedious, especially when a large number of methods of the reused class are used directly.

Non Reused Methods

Methods that are completerly overridden or, more generally, not reused, in the inheritance approach require an equivalent amount of work in the delegation approach. In both cases the new method is implemented in the reusing class.

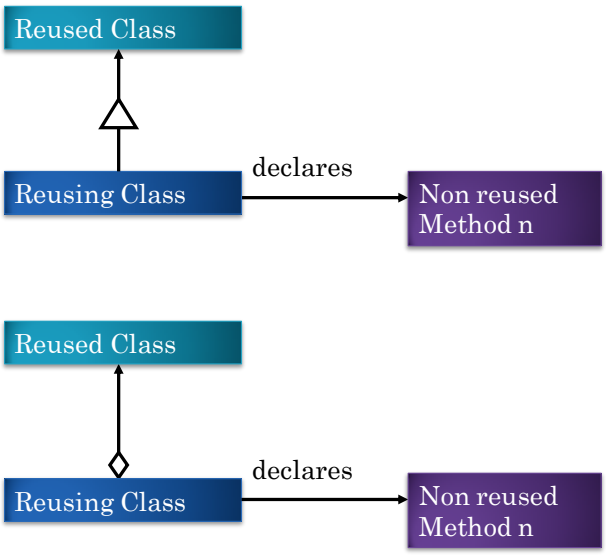


Figure 4 Non reused methods in the two approaches are handled similarly

Super vs. delegate calls

Sometimes an overriding or some other method in U must call some method in S. In the inheritance approach, super is used as the target of the call while in the delegation approach the delegate is the target, as shown in the figure below.

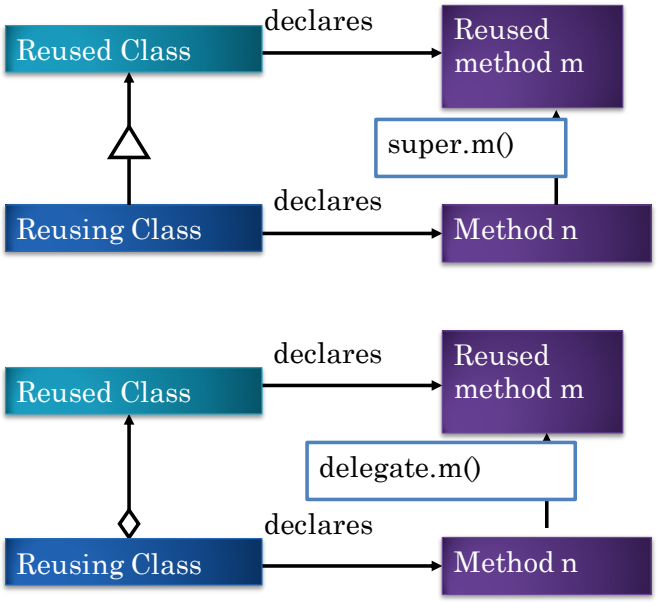


Figure 5 Super vs. delegate calls

This is illustrated by comparing the two implementations of the overridden method, add() in the two classes.

In the inheritance approach, super is the target of the overridden add() method of the reused class:

```
public void add (int amount) {
        super.add(amount);
        notifyObsrevers();
}
```

In the delegation approach, on the other hand, the delegate is the target:

```
public void add (int amount) {
        counter.add(amount);
        notifyObservers();
}
```

Instead of super the reference to the delegate is used, as shown in the figure below.

Callbacks

Sometimes the reused class, S, must call methods in the reusing class, U, not because there is a symbiotic relationship in which each class reuses the other, but because these calls are needed to reuse the service provided by S. These calls in the reverse direction are called *callbacks*. Typically they allow U to customize the service provided by S. The figure below shows the difference between calls and callbacks.

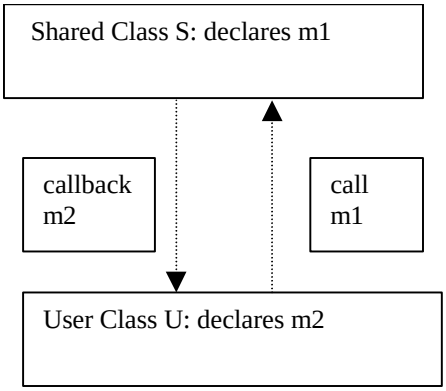


Figure 6 Calls vs. Callbacks

If two classes invoke methods in each other, is it correct to call one the re-used class and another the re-using class? To understand why it is so, consider the re-used class as a provider of some service and the re-using class as a client of this service. Consider now the analogy with a flight reservation organization. You make a call to the organization to make a reservation, but leave a callback number so that the organization can inform you about flight changes or check with you about seat preferences. It is the organization that is providing the service, not you, as the callback you get is part of the service provided. Similarly, a callback from a re-used class to provide some service is part of the service provided by that class to the reusing class.

Abstract methods are examples of callbacks. Consider the abstract history class we saw earlier.

```
public abstract  class ACollection {
        abstract boolean isFull();
        abstract void uncheckedAddElement(Object element)
        public void addElement(Object element) {
                if (!isFull())
                        uncheckedAddElement(element);
                else
                        handleError();
        }
        void handleError () {
                System.out.println("Adding item to a full history");
        };
}
```

The following was an example of a concrete class implemented these methods:

```
public class AStringHistory implements StringHistory {
        public final int MAX_SIZE = 50;
        String[] contents = new String[MAX_SIZE];
        int size = 0;
        public int size() { return size;}
        public String elementAt (int index) { return contents[index]; }
        boolean isFull() { return size == MAX_SIZE; }
        void uncheckedAddElement(String element) {
                contents[size] = element;
                size++;
          }
      }
}
```

Here the abstract class is a shared class used by the concrete subclass. The method addElement() in the shared class makes calls to the abstract methods defined by it. When this method is invoked on an instance of the concrete class, the implementations of the abstract methods in the concrete class are used. Thus, these implementations are examples of callbacks.

In the delegation approach, the following concrete delegate class replaces the abstract class:

```
public class ACollection  implements DelegateCollection {
        StringHistory stringHistory;
        public ACollection (StringHistory theStringHistory) {
                stringHistory = theStringHistory;
        }
        public void addElement(Object element) {
                if (!stringHistory.isFull())
                        stringHistory.uncheckedAddElement(element);
                else
                        handleError();
        }
        void handleError () {
                System.out.println("Adding item to a full history");
        };
}
```

Callbacks invoked on delagator

Delegate class passed reference to delegator

Delegate class must implement interface

Its constructor receives an extra argument referencing the delegator, which is used in the callbacks. These callbacks are implemented by the following class:

```
public class AStringHistory implements StringHistory {
        Collection delegate = new ACollection(this);
        public final int MAX_SIZE = 50;
        String[] contents = new String[MAX_SIZE];
        int size = 0;
        public int size() { return size;}
        public String elementAt (int index) {  return contents[index]; }
        public boolean isFull() { return size == MAX_SIZE; }
        public void uncheckedAddElement(String element) {
                contents[size] = element;
                size++;
            }
      }
      public void addElement(String newVal) {
            delegate.addElemen*newVal);
            }
}
```

Since the delegate class is not a superclass of the delegator class, the headers of the callbacks cannot be defined as abstract methods. Since we wanted to use an interface to type the delegator, we are forced to make them public. If we declared the delegator class in the same package as the delegate class used it to type its instances, then we could have given them reduced access such as default or protected access. Thus, the delegation approach forces us to make a compromise (if we want an interface to type the delegator.)

The following figure explains the differences between the inheritance and delegation approaches to supporting callbacks.

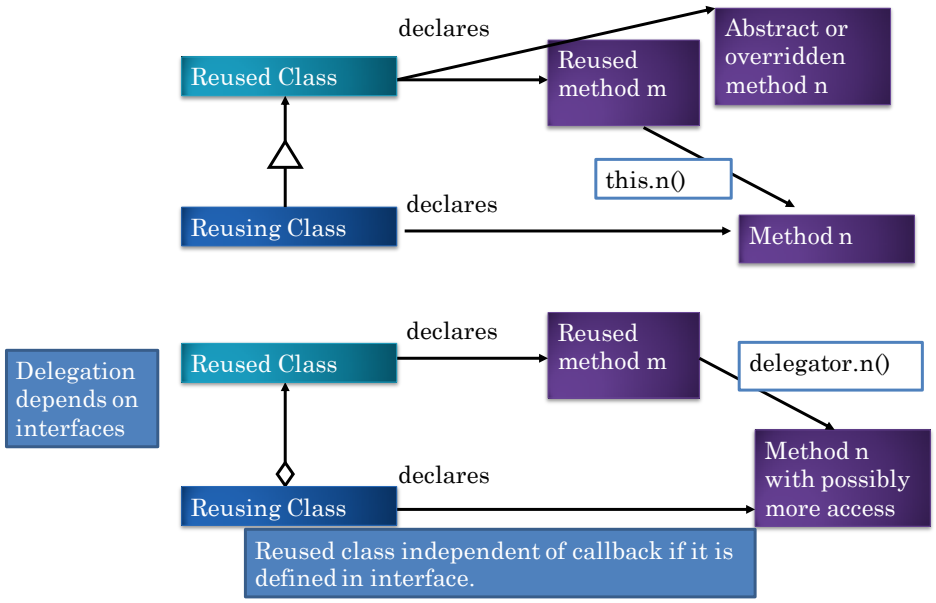


Figure 7 Callbacks as overridding vs. delegator methods

As shown in the figure above, in the inheritance approach, callbacks are methods that override concrete or implement abstract methods inherited from the superclass. In either case, calls are made implicitly or explicitly on the **this** variable. Dynamic dispatch of calls to these methods ensure that it is the implementations in the inheriting class that are used. In the delegation approach, on the other hand, these calls are made directly by using a reference to the delegator. The shared class is passed a reference in a constructor or init method as shown in the reference below.

Accessing instance variables

A similar issue arises when the reusing class, U accesses instance variables of the shared class S. Consider the string database class from the inheritance chapter:

```
package collections;
public class AStringDatabase extends AStringHistory implements
StringDatabase {
        public void deleteElement (String element) {
                shiftUp(indexOf(element));
        }
        public int indexOf (String element) {
                int index = 0;
                while ((index < size) && !element.equals(contents[index]))
                        index++;
                return index;
        }
        void shiftUp (int startIndex) {…}
        public boolean member(String element) {
                return indexOf (element) < size;
        }
        public void clear() {size = 0;}
}
```

It inherits from the string history class, and accesses the instance variables, contents and size, from the inherited class. A delegation-based equivalent is given below:

```
package collections;
public class AStringDatabase implements StringDatabase {
        AStringHistory stringHistory = new AStringHistory();
        public void deleteElement (String element) {
                shiftUp(indexOf(element));
        }
        public int indexOf (String element) {
                int index = 0;
                while ((index < stringHistory.size) &&
!element.equals(stringHistory.contents[index]))
                        index++;
                return index;
        }
        void shiftUp (int startIndex) {…}
        public boolean member(String element) {
                return indexOf (element) < stringHistory.size;
        }
        public void clear() {stringHistory.size = 0;}
        //forwarding method stubs
        …
}
```

Class as type and accessing non public variables in same package

Like other delegating classes, it defines stubs for forwarding methods to the delegate. This class types the delegate using its class rather than its interface. As it is declared in the same package as the delegate class, using the class name as a type allows it to refer to the non-private variables, contents and size, of the delegate. To refer to them, it prefixes the variable name with a reference to the delegate:

```
stringHistory.size
```

An alternative would have to type the delegate using its interface and declare public method in the interface to set and get the size and contents variables. As mentioned before, both are unfortunate compromises required by the delegation approach. However, such compromises are also made in the inheritance approach. Typing the delegate using its class is no worse that inheriting from the reused class – in both cases the reusing class is bound to a specific reused class rather than its interface.

Advantages of Inheritance

We can summarize the advantages of inheritance over delegation.

- There is no need to instantiate multiple classes.
- There is no need to compose the delegator and delegate.
- There is no need to define stub methods that forward calls – the more the reused methods the more the effort involved in defining the stubs.
- Callbacks do not have to be public.

Thus, the delegation-based approach is more painful because of the need to define stubs and compose instances. The equivalent steps in the inheritance solution are automated for us by the language. For instance, when a shared method, m, is invoked in an instance of U, the language automatically does the forwarding of this invocation request to S. It is reassuring to know that inheritance offers benefits over delegation, because otherwise our study of inheritance would have been in vain!

However, in several situations, delegation offers advantages over inheritance, discussed below.

Substituting Reused Class

Suppose we wish to change the reused class, from S, to another class, T implementing the same interface. In the delegation approach, all we have to do is compose an instance of the previous reusing class, U with an instance if T. In inheritance, since the "composition" occurs at compile time, we must define another reusing class, V, to use T. This is shown in the figure below.

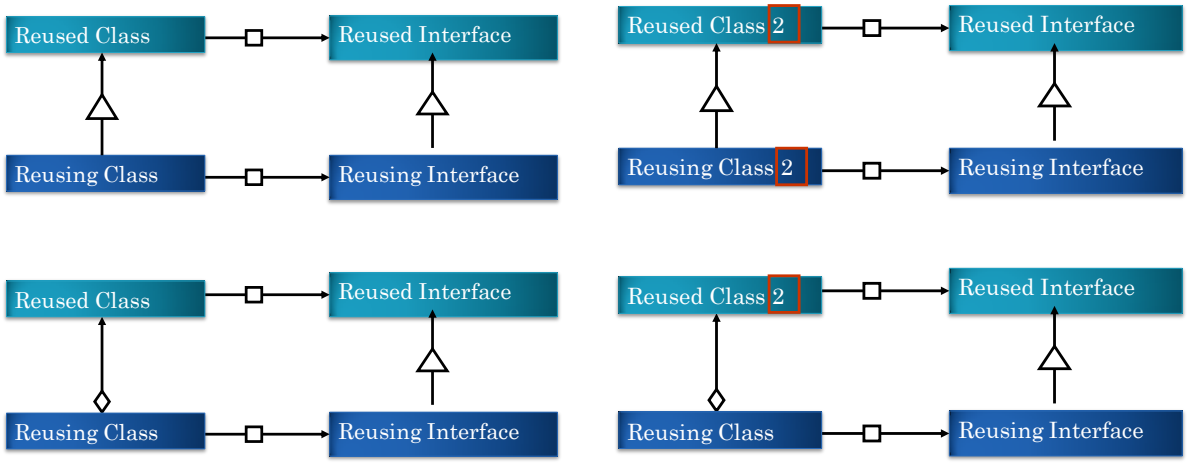


Figure 8 Substitution of reused class does not require new reusing class in delegation

To illustrate, consider we wish build another implementation of the counter interface that resets the counter if it is too large.

```
public class AResettingCounter implements Counter {
        int counter = 0;
        public void add (int amount) {
                if ((counter  - amount) >= Integer.MAX_INT)
                        counter = 0;
                else
                        counter += amount;
        }
        public int getValue() {
                return counter;
        }
}
```

To create an observable version of it, in the the inheritance approach, we must create a new class that is identical to the previous except its extend declaration names the new rather than the old counter class:

```
public class AnObservablerResettingCounter extends AResettingCounter
implements ObservableCounter {
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                super.add(amount);
                notifyObservers();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        public void removeObserver(CounterObserver observer) {
                observers.removeElement(observer);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

In the delegaton approach, we keep the same counter observable class, as its constructor can accept an instance of any class that implements the counter interface:

```
public AnObservableCounter (Counter theCounter) {
        counter = theCounter;
}
```

Multiple Concurrent Reuse

Suppose the instance variables and methods of the shared class, S, must be shared simultaneously by two reusing classes, U and V. In the delegation approach, we can compose an instance of S with both an instance of U and an instance of V. In the inheritance approach, on the other hand, we must make V a subclass of U, or vice versa. This is shown in the figure below.

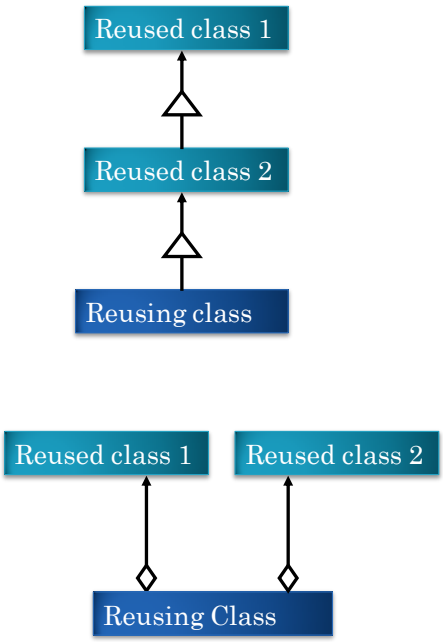


Figure 9 Concurrent reuse of shared class

This means that, in the inheritance approach, for every possible combination of classes wishing to reuse S concurrently, we must create such an inheritance chain. Moreover, it is not possible to dynamically add or remove a user of S. Since the composition is done dynamically in the delegation approach, we can create and change it at runtime. Furthermore it is not clear if V should extend U or vice versa, which means that the IS-A relationship between the two classes is not fundamental.

This is illustrated by considering the binding of counter views and controllers to a counter model. In the MVC chapter, we used delegation to do the binding. For instance to create a user interface to a counter that supported console based input and both console and message based views simultaneously, we used delegation to create the composition, as shown in the following figure.

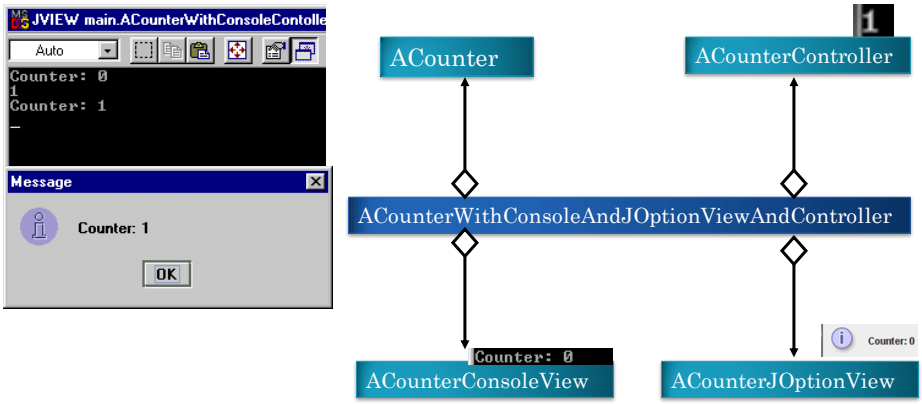


Figure 10 Sharing combination defined dynamically using delegation

The figure below shows an inheritance chain supporting the same user interface.

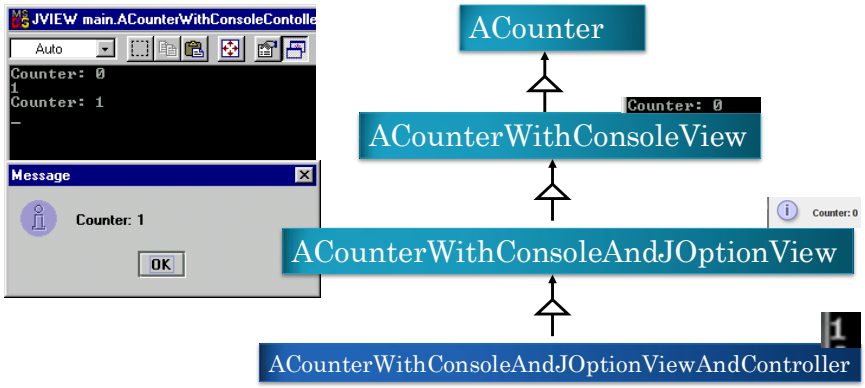


Figure 11 Inheritance chain supporting the same combination

The inheritance chain shows another technique to separate the model, view and controller into different classes.The delegation approach was given before, so we will not consider its delatiled implementation again. Let us consider the implementation of the inheritance approach to illustrate the difference concretely.

The console view does not need to be notified about calls to the add() method. It simply overrides the add method, calls the overridden method, and then displays the new counter.

```
public class ACounterWithConsoleView  extends ACounter {
        void appendToConsole(int counterValue) {
                System.out.println("Counter: " + counterValue);
        }
        public void add (int amount) {
                super.add(amount);
                appendToConsole(getValue());
        }
}
```

The message view is implemented similarly, overriding the add from the console view rather than the model.

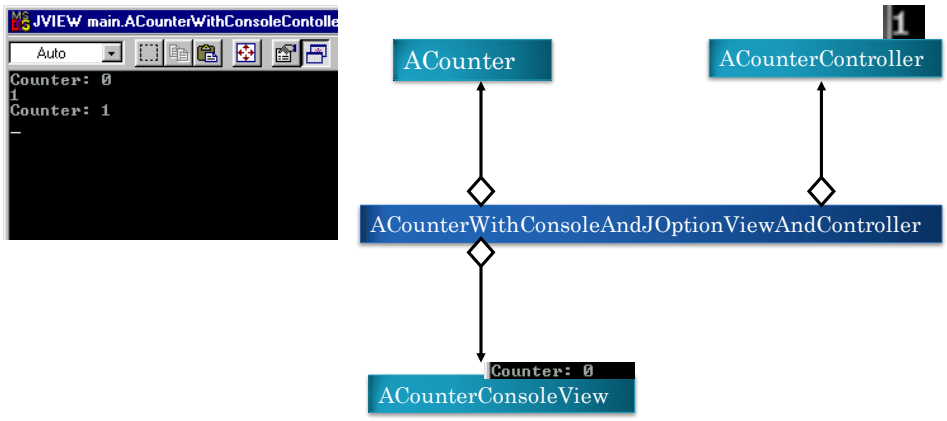
```
import javax.swing.JOptionPane;
public class ACounterWithConsoleAndJOptionView  extends
ACounterWithConsoleView {
        void displayMessage(int counterValue) {
                JOptionPane.showMessageDialog(null, "Counter: " +
counterValue);
        }
        public void add (int amount) {
                super.add(amount);
                displayMessage(getValue());
        }
}
```

The controller is a subclass of the message view that calls the add() method implemented by it, as shown below:

```
public class ACounterWithConsoleAndJOptionViewAndController extends
ACounterWithConsoleAndJOptionView  implements
CounterWithConsoleViewAndController {
        public void processInput() {
                while (true) {
                        int nextInput = Console.readInt();
                        if (nextInput == 0) return;
                        add(nextInput);
                }
        }
}
```

The decision to make the class implementing the class implementing the controller a sublcass of the class implementingthe message view , and the latter a subclass of the class implementing the console view, is arbitrary. Neither of these two relationships is fundamental.

This example shows the advantages of the inheritance approach. There is no need to notify views. Instantiating the controller automatically composes the configuration. Overrridding the write method is a sbustitute for notification. Thus, if there in only one combination of views and controllers, the inheritance approach is much easier to use. The problem arises when more than combination is needed. Each of these combinations must be anticipated. Moreover, these combinations result in code duplication. Assume that we wish to subtract the message view from the combination above. The delegation approach simply allows us to just remove the HAS-A link to the view.



The inheritance approach requries us to create a new controller. This is shown in the figure below.

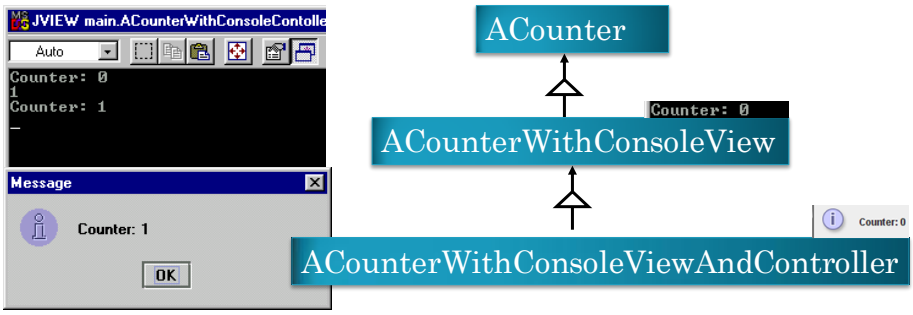


Figure 12 Changing the composition requires new controller in the inheritance approach

The new controller is a duplicate of the previous one except that it inherits from a different class:

```
public class ACounterWithConsoleViewAndController extends
ACounterWithConsoleView  implements CounterWithConsoleViewAndController
{
        public void processInput() {
                while (true) {
                        int nextInput = Console.readInt();
                        if (nextInput == 0) return;
                        add(nextInput);
                }
public class ACounterWithConsoleAndJOptionViewAndController extends
ACounterWithConsoleAndJOptionView i
CounterWithConsoleViewAndController {
        public void processInput() {
                while (true) {
                        int nextInput =
                        if (nextInput == 0) return;
                        add(nextInput);
                }
        }
}
```

Only difference between it and previous inheriting controller but must create a new class!

Distribution

Another advantage of delegation arises from the fact that a different instance is created for both the reused and reusing class. In a distributed environment, these can be placed on two different locations. For example, a view can be on the local client machine and the model can be on a remote server machine. The inheritance approach does not offer this flexibility as one instance is created that combines the instance members declared in the reused and reusing class. This is shown in the figure below.

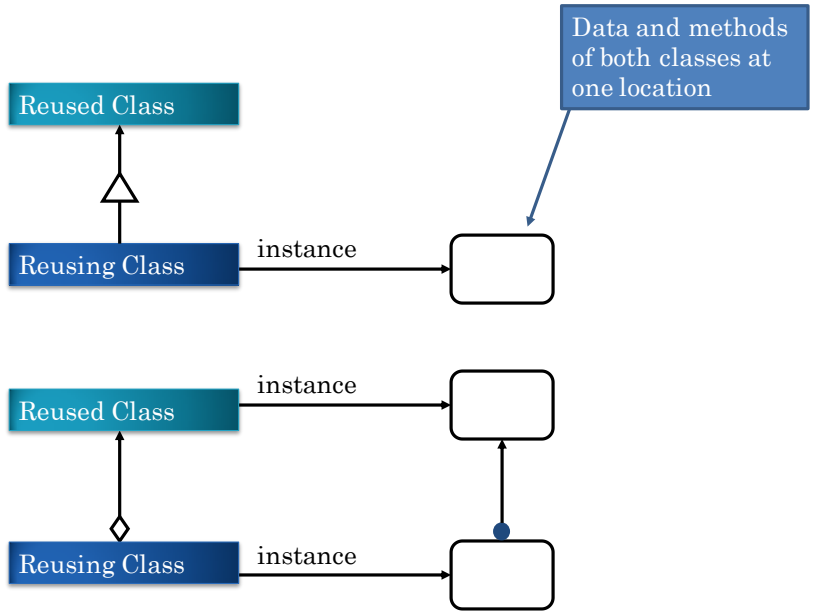


Figure 13 Delegate and delegator can be in two different locations

Reusing multiple classes

Sometimes a class must reuse more than one class. In the inheritance approach, this means making the reusing class a subclass of multiple reused classes, while in the delegation approach this means making the reusing class have references to more than one delegate. This is shown in the figure below.

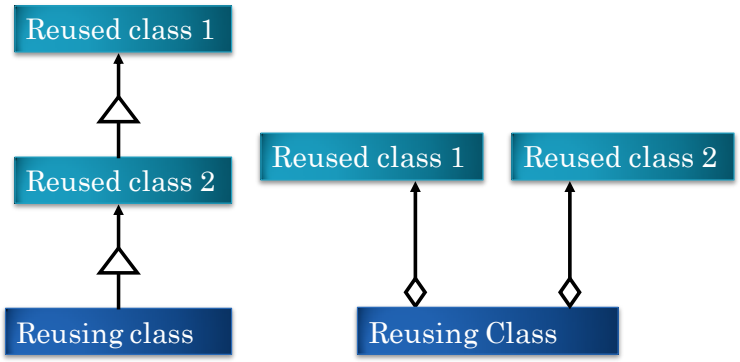


Figure 14 Superclass chain vs. multiple delegates

In languages such as Java that do not support multiple inheritance, one must create a linear chain of subclasses. This means, it is not possible to reuse a class low in the hierarchy without also inheriting from classes higher in the hierarchy. The delegation approach does not have this problem.

Advantages of Delegation

To summarize, delegation offers several advantages over inheritance:

- The reused class can be changed without changing reusing class.
- Multiple reusing classes can share reused class variables simultaneously.
- Variables and methods of reusing and reused class can be on separate computers.
- It works in single inheritance languages

Making the tradeoff

Since both delegation and inheritance have their pros and cons, we must look carefully at the situation before making the choice. When the benefits of delegation do not apply, inheritance should be the alternative as it easier to use. Consider the delegating string database. It is bound to a single string history class, and there is no obvious advantage to separating the instance methods of the history and database class on separate computers. There is the disadvantage that a large number of forwarding methods must be defined. Thus, in this case, inheritance is the preferred approach. In case of the counter model, views, and controller, we need the ability to compose the objects in multiple way and possibly put the model on a server and the view and controller on a client machine. Thus, the delegation approach should be used in this case.

Sometimes it is clear that the inheritance approach is wrong.

When the best name we can choose for a reusing class has the form:

SwithU

where S is the reused class S and U the reusing class, then we know that S should have a reference to U rather than extend S. Thus, from the name ACounterWithConsoleView, we know that the best relationship between the two classes is HAS-A rather than IS-A.

Similarly, when the IS-A relationship between two classes can be reversed, then we know it is the wrong relationship between the two classes. For example, when we create the multiple view counter by making the console view class an extension of the message view, or the reverse, then we know that IS-A is the wrong relationship between the two.

Sometimes the reusing class does not need all of the public methods of defined by the shared class. In this situation IS-A is clearly the wrong relationship between the two. Consider using a vector to create a string history. If we make string history an extension of the ArrayList or Vector class provided by Java, defining new methods that add and retrieve strings instead of arbitrary objects, then we also inherit the methods to add and remove arbitrary objects. We can override these methods to do nothing, but that is a lot of work and more important deceive a user of the class who looks at the public methods to understand its behavior. The fundamental problem here is that we do not want a string history to be used wherever a vector can be.

This problem can arise even if the reusing class needs all the methods of the reused class. Consider the figure below

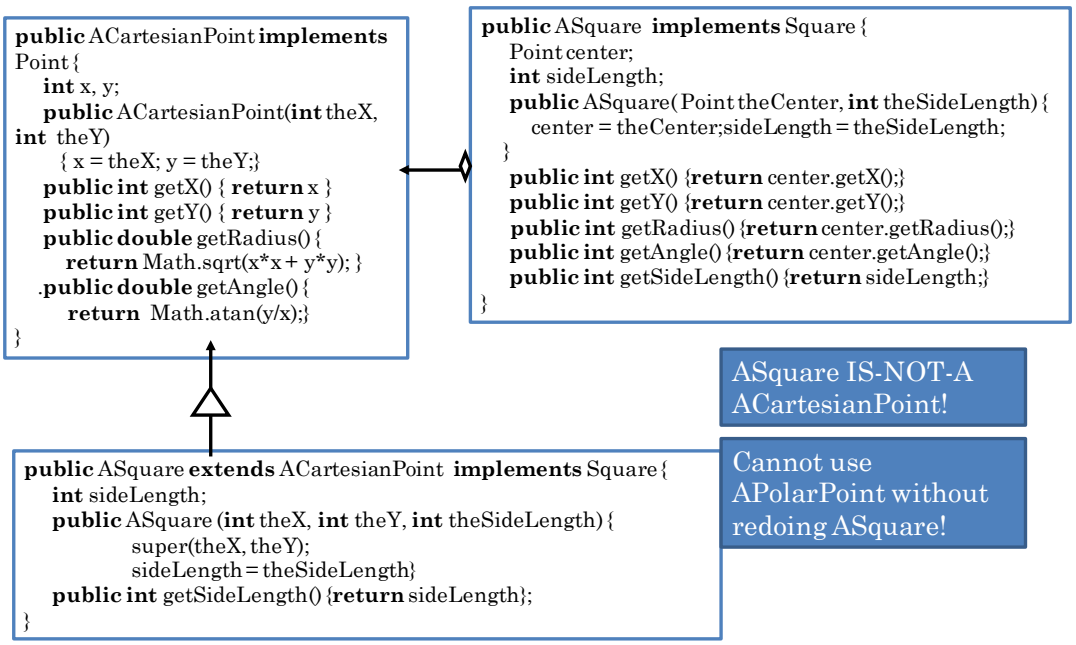


Figure 15 ASquare IS-A ACartesianPoint vs. HAS-A ACartesianPoint

It shows two ways of using the code in a class defining a point is used to create one defining a square. In either case the point class is used to represent the center of the square. In the inheritance case, the square class inherits from the point class while in the delegation approach it has a reference to the point class. In both cases, the square class defines an additional variable and property to represent the length of each side of the square.

In the inheritance approach, the square class inherits all of the methods of the point class. Yet it is wrong as we don't want to use a square wherever we can use a point, that is, ASquare IS NOT APoint. The practical disadvantage of this approach is that the inheriting class is obscure (it is not clear that the point class represents the center) and we must create a separate square class to reuse a point class that stores the Polar rather than Cartesian representation of a point.

Making the tradeoff in everyday software

Authors of objects we use every day have had to wrestle with the choice between delegation and inheritance. In Java 1.0, a user of a widget class by inheriting from it, overriding the method called in the widget that was supposed to process some user action. The overriding method used to call super in the widget class to process the action and then added application-specific functionality such as incrementing a counter when an increment button is pressed. This approach was, in fact, borrowed from the Smalltalk user-interface toolkit. In later versions of Java, the delegation approach is used. A widget user has a reference to an instance of the widget class and is notified (via the observer pattern) after the widget

processes the user action. In response to the notification, it performs application-specific functionality.

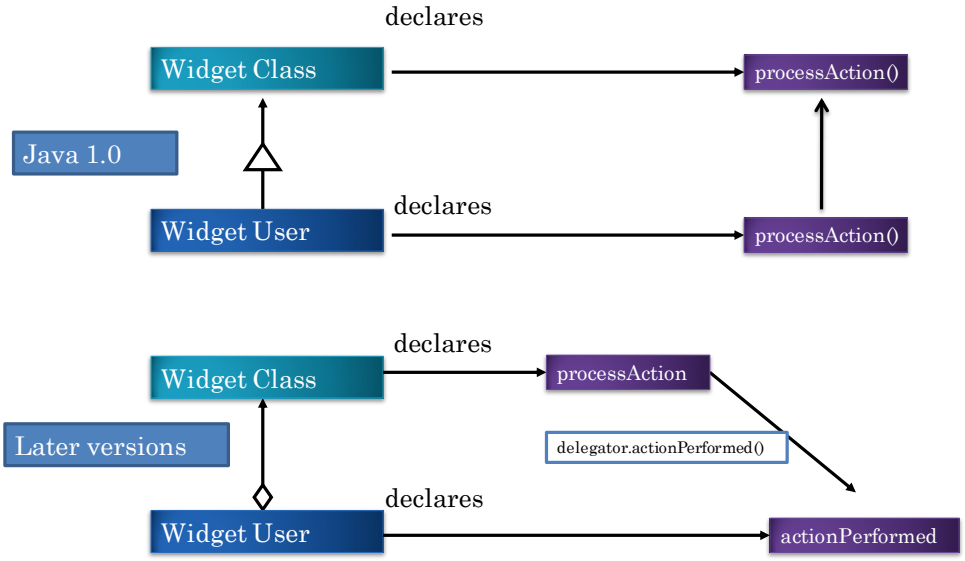
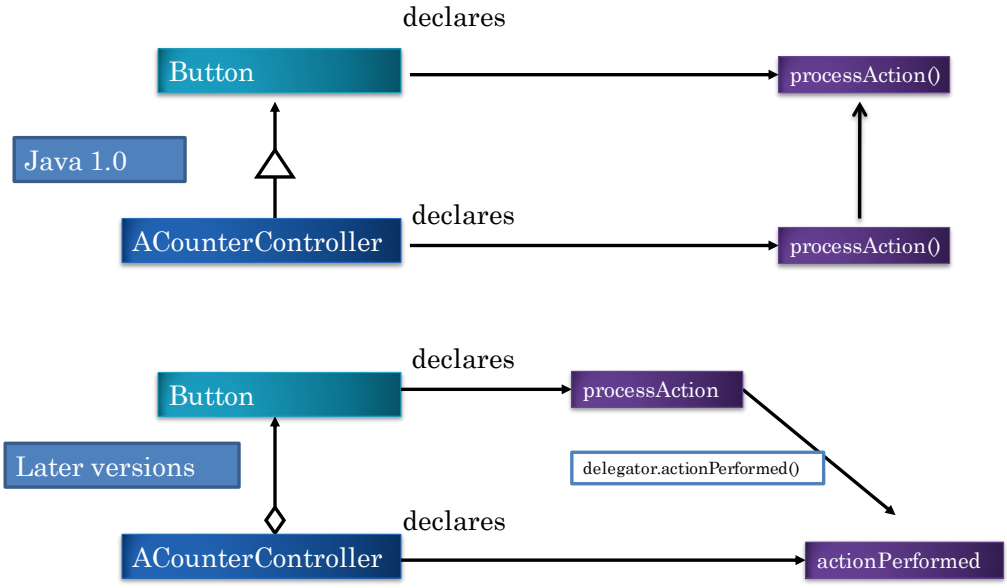


Figure 16 Inheriting a vs. delegating to a widget class

For example, in the inheritance approach, a counter controller that wishes to use AWT Button would be a subclass of Button! In the delegation approach, it would have a reference to Button.



The main reason for changing to the delegation approach was to overcome the problem of single-inheritance in Java. With the inheritance approach, it was not possible for a class that extended a non-widget class to use a widget or for a class to use two different widget classes. The other advantages of delegation also apply such as the ability to transparently reuse another widget class implementing the same functionality.

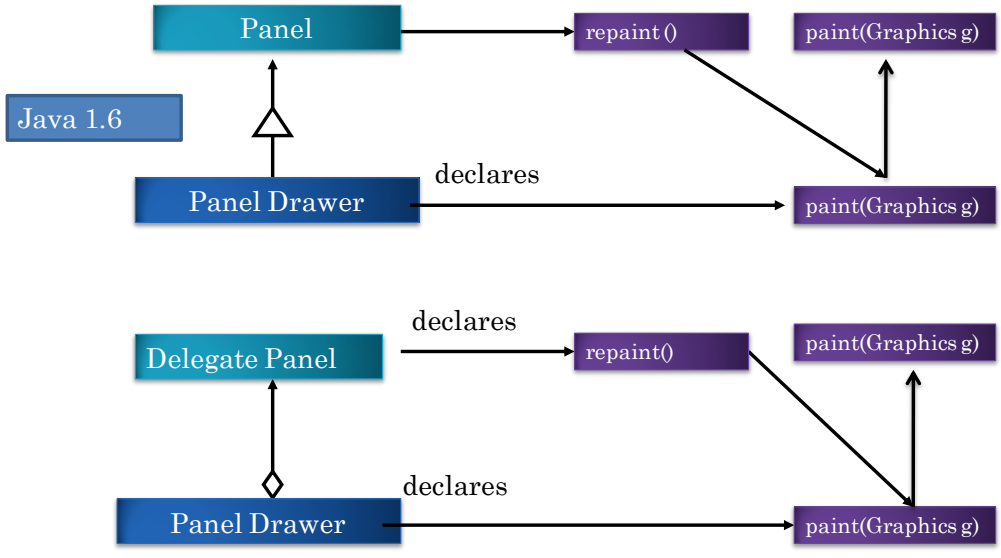
A more interesting reason is that programmers may want to easily switch between competing implementation of a widget. This was not possible with the inheritance approach, as illustrated by the example above. In the inheritance approach, we would have to create a separate console controller that inherits from JButton that repeats the code. In the delegation approach, a single console controller could have links to either class. Unfortunately, Button and JButton do not have a common interface for invoking button functionality, but in theory they could.

Java provides a special class, Observable, that implements the observer pattern. It was intended to be the superclass of any observable class. However, such an object could not be a subclass of another class. For example, it was not possible for AStringSet, an extension of AStringDatabase, to use this class. While this class is still exists in Java, Java now also provides another class, PropertyChangeSupport, which provides a delegation-based implementation of the obervable pattern.

Similarly, when you study threads, you will see it is possible to inherit from a Thread class or have a reference to the class and implement a Runnable interface defining threads. The Runnable version did not exist in version 1.0.

In general, the fashion today is to use delegation rather than inheritance. The discussion above had given the reasons, techniques, and drawbacks for doing so.

As it turns out, inheritance continues to be used, even in everyday software. For example, in Java AWT/Swing, if we want to create an application class that draws on a panel, we must make the class a subclass of an AWT Panel, and override its paint() method. The reason is that whenever a panel is to be repainted, AWT calls the paint(Graphics g) method in the panel, where Graphics is a class that defines methods to paint in the panel. To perform custom painiting, our application class must become a subclass of Panel and override the paint(Graphics g) method. After creating the custom graphics, the subclass method must call the inherited method to finish the painting task. The class that does the painting has no panel behavior, yet it is a subclass. More practically, it is not possible to create a class that paints either in a Panel or one of its subclasses such as Applet.



IBM's SWT toolkit fixes this problem by having using the HAS-A relationship between a panel drawer and an panel.

In summary, even experienced programmers such as designers of Java libraries make the mistake of using the inheritance relationship indiscriminately. Therefore, whenever you are tempted to use the IS-A relationship for code reuse, to through the pros and cons presented here to see if it is appropriate to do so. Otherwise, like the Java library developers, you might have to release new versions of code simply to replace inheritance with delegation.