O'REILLY ONJava.com The Independent Source for Enterprise Java
Published on **ONJava.com** (http://www.onjava.com/)
See this if you're having trouble printing code examples


# A Generic MVC Model in Java

by Arjan Vermeij
07/07/2004

Model-View-Controller (MVC) is a widely used design pattern, especially popular in graphical user interface (GUI) programming. JDK 1.5 introduces parameterized types, or *generics*. Combining the two allows for a generic implementation of the MVC design pattern, freeing the programmer from writing code that handles the registration and notification of listeners, as well as from writing getter and setter methods for the properties of models. This article shows how this can be accomplished.

## Model-View-Controller

The ideas behind MVC are quite old, developed during the 1970s. The invention of MVC is attributed to Trygve Reenskaug, who was working at Xerox PARC at the time.

The idea is to break up an application into three parts:

- **A *model* that holds part of the current state of the application.** In this article the state of a model is represented by a set of *properties*. A property is implemented as a getter and a setter method, as in the JavaBeans concept.
- **A *view* that is, for whatever reason, interested in knowing the current state of the application.** A view registers itself with a model as a *listener* on the model. Whenever the state of a model changes, the model notifies its registered listeners. In GUI programming, a view often does nothing more than display the state of the model through some graphical component.
- **A *controller* that changes the properties of the model.** When and how a controller changes a property is at the discretion of the controller. In GUI programming, a controller is often something that takes input from the user; for example, a button or a text field.

To illustrate, let's introduce a simple example. The example uses two simple models, a view, and three controllers. It is implemented using JFC/Swing, and is shown in Figure 1.

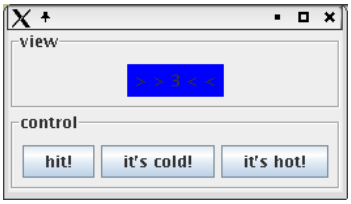


*Figure 1. A simple MVC demo*

This example is not meant to be a useful application at all; its only purpose is to show how generics can help to reduce the effort involved in writing applications based on the MVC design pattern.

The example is about a hit count and a temperature. The upper half of the window shows the view. The number shows the hit count, and the color symbolizes the temperature. The bottom half of the window shows the controllers. Whenever the user clicks on the "hit!" button, the hit count is incremented. Clicking on the "it's cold!" button makes the color in the view turn blue, while clicking "it's hot!" makes the color turn red.

The example uses two models, one for the hit count and one for the temperature. Let's have a look at the code for the handling of the temperature. Here is the code for the interface implemented by the view, the listener on the `TemperatureModel`.

```
public interface TemperatureModelListener
{
  void temperatureModelChanged
     (TemperatureModel temperatureModel);
}
```

And here is the code for the `TemperatureModel` itself.

```
import java.util.*;

public class TemperatureModel
{
  public enum Temperature {cold, hot};

  private final List <TemperatureModelListener>
                           temperatureModelListeners;
  private Temperature temperature;

  public TemperatureModel ()
  {
    this.temperatureModelListeners
       = new ArrayList <TemperatureModelListener> ();
    this.temperature = Temperature.cold;
  }

  public void addTemperatureModelListener
     (final TemperatureModelListener temperatureModelListener)
  {
    if (! this.temperatureModelListeners.contains
             (temperatureModelListener)) {
      this.temperatureModelListeners.add
         (temperatureModelListener);
      notifyTemperatureModelListener
         (temperatureModelListener);
    }
  }

  public void removeTemperatureModelListener
     (final TemperatureModelListener temperatureModelListener)
  {
```

```
    this.temperatureModelListeners.remove
       (temperatureModelListener);
  }

  public void setTemperature (final Temperature temperature)
  {
    this.temperature = temperature;
    notifyTemperatureModelListeners ();
  }

  public Temperature getTemperature ()
  {
    return this.temperature;
  }

  private void notifyTemperatureModelListeners ()
  {
    for (final TemperatureModelListener temperatureModelListener
            : this.temperatureModelListeners) {
      notifyTemperatureModelListener (temperatureModelListener);
    }
  }

  private void notifyTemperatureModelListener
     (final TemperatureModelListener temperatureModelListener)
  {
    temperatureModelListener.temperatureModelChanged (this);
  }
}
```

The controllers set the value of the temperature property. The view registers itself as a temperature listener with the temperature model. You can download the complete *mvc-traditional.zip* code example from the Resources section at the bottom of this article.

The code uses some features introduced in JDK 1.5. First there is the declaration of the `Temperature` type:

```
enum Temperature {cold, hot}
```

This is an enumeration type with values `cold` and `hot`.

Then there are declarations, such as:

```
... List <...> ...
```

This is actually an instantiation of a generic type. See the next section for a brief explanation on generics.

Finally, the `notifyTemperatureModelListeners` method contains an unfamiliar-looking `for` loop of the form:

```
for (type variable : collection) { ... }
```

This `for` loop simply iterates over all the values in the collection.

As is immediately clear from the listing above, the `TemperatureModel` class is quite big, considering the simplicity of the features it supports. The `HitModel` class is about the same size. The `TemperatureModel` and `HitModel` classes are, as you might expect, quite similar -- both have methods to add and remove listeners, and both have methods to get and set the value of a property. It is tedious to write and maintain code that is so similar. Perhaps having only two models might not be a problem, but a serious GUI application can have lots of models.

The question is then, how we can use generics to make our life easier? Let's first explain what generics are.

## New in JDK 1.5: Generics

JDK 1.5 introduces parameterized types, or generics. One of their main advantages is increased type safety by eliminating the need for class casts.

The collection classes in the `java.util` package have been retrofitted with generic interfaces. Thus, instead of writing:

```
final List stringList = new ArrayList ();

stringList.add ("hello");
// Ok, but may cause a runtime exception later on:
stringList.add (new Integer (1));
```

we can instead write:

```
final List <String> stringList = new ArrayList <String> ();

stringList.add ("hello");
// Not ok, compile time error:
stringList.add (new Integer (1));
```

For more information on generics in Java, see William Grosso's article "Generics and Method Objects" or the tutorial "Generics in the Java Programming Language" (PDF), by Gilad Bracha. You may want to consult JSR-014, which is the original proposal.

## A Generic Model

As stated before, the `TemperatureModel` and `HitModel` classes are quite similar. Let us first try to factor out the code that handles the listeners of a model.

Both the `TemperatureModelListener` and `HitModelListener` interfaces each have one method, both of which do the same thing: notifying a listener about a change in a model. The names of these methods differ, but apart from that the only other difference is the type of their formal parameter: `TemperatureModel` and `HitModel`, respectively. That is exactly what generics are all about, so here is the code for a generic listener.

```
public interface ModelListener <M>
{
  void modelChanged (M model);
}
```

Although I do not like abbreviations, I use the generic formal type name `M` here. Using one-letter names for generic formal types is the convention, as explained in "Generics in the Java Programming Language" (PDF).

We then rewrite our listeners as shown below:

```
public interface TemperatureModelListener
extends ModelListener <TemperatureModel> { }
```

```
public interface HitModelListener
extends ModelListener <HitModel> { }
```

Both the `TemperatureModel` and `HitModel` classes manage listeners. Again, the only significant difference is the type of the listeners. So we might be tempted to write:

```
public class Model <L> ...
```

This, however, does not work, because the model makes a call to the listener's `modelChanged ()` method. We have to make sure that a model is instantiated not with just any type, but with something that is a `ModelListener`. In generic speak, the formal type must be *bound* by `ModelListener`.

```
import java.util.*;

/**
* A generic MVC model.
*/
public class Model <L extends ModelListener>
{
  private final List <L> listeners;

  public Model ()
  {
    this.listeners = new ArrayList <L> ();
  }

  public void addModelListener (final L listener)
  {
    if (! this.listeners.contains (listener)) {
      this.listeners.add (listener);
      notifyModelListener (listener);
    }
  }

  public void removeModelListener (final L listener)
  {
    this.listeners.remove (listener);
  }

  protected void notifyModelListeners ()
  {
    for (final L listener : this.listeners) {
      notifyModelListener (listener);
    }
  }

  protected void notifyModelListener (final L listener)
  {
    listener.modelChanged (this);
  }
}
```

Being able to subclass this `Model` considerably reduces the size of our models. Shown below is the revised `TemperatureModel`.

```
public class TemperatureModel
extends Model <TemperatureModelListener>
{
  public enum Temperature {cold, hot};

  private Temperature temperature;

  public TemperatureModel ()
  {
    this.temperature = Temperature.cold;
  }

  public void setTemperature
     (final Temperature temperature)
  {
    this.temperature = temperature;
    notifyModelListeners ();
  }

  public Temperature getTemperature ()
  {
    return this.temperature;
  }
}
```

Some minor changes to the code of the client of our models are needed. For example, instead of calling `hitModel.addHitModelListener()`, we now call `hitModel.addModelListener()`. The complete *mvc-generic-model.zip* example is available in the Resources section below.

We managed to factor out the listener-handling code. This is an improvement, as it allows us to remove four methods from each model. But can we do better still? In fact, we can, because our two models still have quite a few things in common. Let's look at a generic property.

## Generic Properties

The `TemperatureModel` has a `temperature` attribute, and the `HitModel` has a `number` attribute. Apart from their types, the attributes are quite similar. Again, this makes for a good candidate for a generic solution. Our first attempt might look something like:

```
public class Property <T> ...
```

where `T` stands for the type of the property. However, the setter methods make a call to the model's `notifyModelListeners ()` method. A property needs a reference to the model of which it is a property. Thus, we might come up with something like the following:

```
public class Property <Model, T> ...
```

where `Model` stands for the model of which the property is a property. But there is a more elegant solution: make the `Property` class a non-static inner class of `Model`. Here is the code:

```
protected class Property <T>
{
  private T value;

  public Property (final T initialValue)
  {
    this.value = initialValue;
  }
```

```
  public void setValue (final T value)
  {
    this.value = value;
    notifyModelListeners ();
  }

  public T getValue ()
  {
    return this.value;
  }
}
```

This further reduces the size of our models. Below is the final version of our `TemperatureModel`:

```
public class TemperatureModel extends
        Model <TemperatureModelListener>
{
  public enum Temperature {cold, hot};

  public final Property <Temperature> temperature;

  public TemperatureModel ()
  {
    this.temperature
       = new Property <Temperature> (Temperature.cold);
  }
}
```

At first glance, the declaration of the `temperature` property seems to violate one of the most basic rules of class design: make instance variables private. In this case, however, this is not an issue, because the `temperature` variable is declared `final`, and the `temperature` variable itself does not have any public instance variables. Non-static inner classes model composition. They can be seen as dynamic extension of object state. If the object itself is public, then in general, there is no problem in its dynamic extensions being public, too. For more information on inner classes, see the excellent "Nested Classes, Part 1," by Robert Simmons, Jr.

Client code involving properties needs some minor modifications: e.g., instead of calling `temperatureModel.getTemperature()`, we now make a call to `temperatureModel.temperature.getValue()`. This example is available as *mvc-general-property.zip* in the Resources section below.

## Constrained Properties

The `Property` class is very simple. It allows a client to get and set the value, nothing more. Sometimes, properties are a little more complicated. What if, for example, the model imposes a constraint on the value of a property?

Constraints on the value of a property can be easily implemented by adding a custom setter method to an anonymous subclass of the instantiation of the `Property` class. For example, the code below shows how to limit the hit count of our `HitModel` to a maximum of 10.

```
public class HitModel extends Model <HitModelListener>
{
  public final Property <Integer> number;

  public HitModel ()
  {
    this.number
       = new Property <Integer> (0)
         {
           public void setValue (final Integer value)
           {
             super.setValue (Math.min (10, value));
           }
         };
  }
}
```

## Conclusion

At the start of this article, the `TemperatureModel` class had one constructor, four public methods, two private methods, and two private instance variables. By making the `TemperatureModel` class a subclass of the generic `Model` class, we managed to reduce that to one constructor and one public instance variable. That's progress!

## Resources

- mvc-traditional.zip example
- mvc-generic-model.zip example
- mvc-generic-property.zip example

*Arjan Vermeij is a senior software engineer and Principal Scientific Assistant at the Nato Undersea Research Center*

---

Return to ONJava.com.