Martin Lanter

Model-View-Controller (MVC) Pattern in Bomberman

How MVC is implemented in a larger application and potential solutions to common decision problems that arise when implementing MVC.

Abstract

Bomberman is a game written in Java using the Swing library and implementing the Model-View-Controller (MVC) pattern. Much literature can be found in books and online that describes the MVC pattern. They usually contain a small demonstration with one class per model, view and controller respectively. However, they lack a larger demonstration where model and view are compositions of multiple classes. Bomberman's model and view consist of roughly 25 classes each (without AI). This article explains how they all work together.

Table of Contents

1 Introduction

The Model-View-Controller (MVC) pattern is a widely used programming pattern in Swing applications. The basic principle of MVC is a partition of the logic that belongs to the model and the logic that belongs to the view. A controller serves as glue between them. Many tutorials exist in books and online[1] [2] [3] and most of them come with a small demonstration of a minimal MVC implementation. However, real applications are larger and consist of many more classes. The Bomberman client, for instance, consists of roughly 100 classes and 10000 lines of code. Model and view consist of roughly 25 classes each. Applying the MVC pattern to a larger application forces many non-trivial decisions. Some of these are outlined here and possible solution discussed. I have implemented the MVC pattern multiple times in different projects and I would like to share my insights about some common misconceptions and problems that arise when implementing the MVC pattern and which are not covered by small 3-class-demonstrations.

If you plan to program your own Swing application, in particular a game, you might find this article helpful to not make the same mistakes I had to learn from and to skip hours of philosophizing about how to validly implement MVC. This article targets programmer that have a basic understanding of Java, AWT or Swing and the MVC pattern.

2 The Model-View-Controller (MVC) Pattern

The model is a data structure that represents the state of the game. It holds all information about where each obstacle is placed, where bombs, explosions and players are and all their properties such as the speed and direction of a player. A good model design is vital as your whole game bases on it. The model must not depend on anything from the controller or the view and thus must not have any direct references to such objects (transitively).

Bomberman has a class Model that is the root to all information. The model contains lists that contain all the objects on the playing field. There is one class that represents a player, bomb, explosion, item or obstacle respectively. These are model classes as well. Some of these objects have a back-reference to the model. For instance, a player needs to know the width of the playing field which is stored in the model. Since both classes are model classes such cross references are acceptable and in some cases indispensable.

The view displays your state or rather model. It has a reference to the model and is able to retrieve the exact state to render and display on the screen (**Figure 1**: State Query). The view must not depend on the controller and thus must not have any direct references to it (transitively).

Bomberman's bomb model class does not have any idea, how a bomb looks like. It only knows data such as where it is placed and by which player. The view, however, certainly knows how a bomb is supposed to look like. Thus, it paints the image that belongs to the bomb at the current time at the position where the bomb is placed. In Bomberman, the bomb has its own view class BombUI. An object of BombUI has a reference to the Bomb object it graphically represents. Analogously, each model-object such as Explosion, Item, Player and so on has a corresponding UI view-object.

[1] Java SE Application Design with MVC: http://www.oracle.com/technetwork/articles/javase/index-142890.html
[2] Game Architecture: http://www.koonsolo.com/news/model-view-controller-for-games/
[3] MVC – Design (German): http://www.java-forum.org/allgemeines/91829-mvc.html

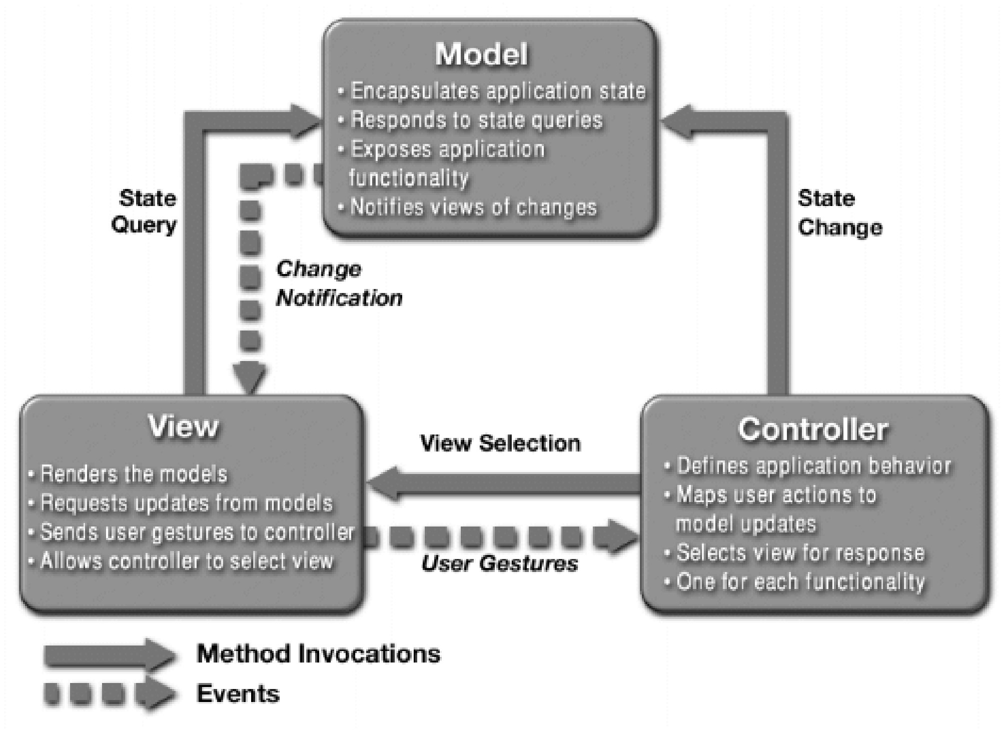
Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Figure 1: MVC diagram from Sun's Model-View-Controller pattern documentation for Swing.

If the model does not have a direct reference to the view, how is it able to notify the view, when the state has changed, e.g., when a player has died. The answer is the observer pattern (**Figure 1**: Change Notification). An observable object contains a list to all objects that have interest into the observable. Interested object are called observers and register at the observable. The observer only exposes a restricted interface to the observable, i.e. it implements the observer interface that only provides update-methods. The observable does not know the exact type of its observers; it only knows how to notify them, if an event occurs.

The controller glues the model and view together. Usually, it first instantiates the model followed by instantiating the view (**Figure 1**: View Selection) with a reference to the model (and the view registers itself at the model as observer). If the user changes something on the view, e.g., clicks a button or enters a text into a text field, the change must be propagates to the model. Again, we use the observer pattern. In Swing, observers are called listeners (**Figure 1**: User Gestures). To listen to a button one has to register an ActionListener containing a method actionPerformed() that is called when the user presses the button. Such a listener is a mini-controller and changes the model according to the occurred event (**Figure 1**: State Change). It is somewhat debatable whether a listener must necessarily be defined in a controller class/package or whether it is in some cases fine to define them in a view class despite being a controller. Strictly speaking, a controller must be in a controller class and your view should provide setter for all the listener the controller is able to register, e.g., the OK-button-listener, cancel-button-listener and all listener to RadioButtons, CheckBoxes, Textfields and other modifiable Swing components. As a consequence, you end up providing numerous methods just for registering a controller that propagates a change to the model and could have been written in the view just as well. If you even nest multiple view panels and add a RadioButton to the innermost panel, each panel has to provide a method to set the listener for that RadioButton and delegate it to the next inner panel. Since this is undesirable, most Swing developers relax the MVC partitioning in this regard and define listener in view classes.

Side note: Controlling objects such as a timer that periodically triggers your game to update its state can be placed inside the controller. You could argue that periodic updates are part of the model world and move it into the model. I think this is fine if you use the java.util.timer. The javax.swing.timer belongs to a swing (view) package though and should not be inside the model. Note that there are classes in the AWT and Swing package that are called model classes like javax.swing.DefaultListModel. My advice is yet not to mix up your model classes with such swing model classes. They are not designed to be low-level model classes since they already define with which view class they must be displayed. What if you later want to display the same objects in a JList on one JPanel and as JComboBox on the other JPanel? Better use an array or collection class in your model.

3 Observer Design

First, let us look at the listeners that listen to user gestures. Swing components provide methods where a controller registers listeners. For each listener type, there is an interface, e.g. ActionListener, MouseListener, MouseMotionListener, MouseWheelListener, ComponentListener and many more. Subclass these interfaces and implement the required method to listen to events from users. Some listeners contain many methods, MouseListener, for instance, has five. If you want to react to mouse clicks alone, you can subclass MouseAdapter and override the method mouseClicked. Adapter classes are abstract classes that have dummy implementations that do nothing for the methods of an interface. If you have multiple components that share an ActionListener, consider using a java.swing.Action[4].

The second observation flow goes from the model to the view. If you have a boolean in your model that is only changed by a CheckBox, you might ask yourself why the model should update the CheckBox after the listener of the CheckBox updates the model. First of all, you do not need to be afraid of an endless loop where listener and model eternally keep updating each other. The smart Swing components only trigger an event if the new value is different from the current value. However, if the user clicks the checkbox the graphics switches automatically without update from the model. Yet, if you have two graphic components displaying the boolean state, clicking one component does not change the graphics of the other. For instance, if you include a JCheckBoxMenuItem to your menu and click on it, the JCheckBox still displays the old value. Therefore you need that possibility to update observers when the boolean changes.

Java comes with a ready-to-use observable class and observer interface in the java.util package. An observer must implement a method update() that takes an Object as parameter. When the state of the observable has set to changed and notifyObservers(arg) is called, the update method is executed with the specified argument arg. Since arg is an Object, the observer needs type checking to cast it to the right type and use it. This results in ugly code. Alternatively, if you only need to tell the observer what kind of event has occurred but no additional object is required for the observer to react, you can use an enumeration (Enum). A Java Enum is a special data type that represents a constant value. Define an Enum with one constant per kind of event and when an event occurs give the corresponding Enum constant as argument to notifyObserver. The observer then can check for the kind of event by using an if-statement and "==" or a switch-construct.

[4] http://docs.oracle.com/javase/tutorial/uiswing/misc/action.html

However, you might prefer to have arguments of a normal class and yet avoid type checking. Generics, introduced in Java 1.5, solve this problem. In the following code snipped A is the generic type of the argument.

```
    public interface Observer<A> {
        public void update(A arg);
    }

    public abstract class Observable<A> {

        private List<Observer<A>> observers = new LinkedList<>();

        public void addObserver(Observer<A> obs) {
            observers.add(obs);
        }

        public void notifyObserver(A arg) {
            for (Observer<A> obs:observers)
                obs.update(arg);
        }
    }
```

To have an observable that notifies observers with an argument of type String, let it extend Observable<String>. Observers that implement Observer<String> can be registered at the observable. If observers register at multiple observables and need to know the source of a notification, you can use an argument type that provides a method to get the source. Note: Any observer that can be notified with an argument of type X can be added to any observable that notifies with an argument of type X. If you want to stronger distinguish between different observers, read the blog[5] from “Jaana” (I don’t know her full name) who explains why you need three generic types for that purpose.

The problem with a generic observer is that they have only one update method. Code to distinguish many different events grows large and confusing. One solution is to make the argument a visitor. The observable calls its observer with the visitor as argument. The observer tells the visitor to visit it (the observer) and the visitor calls the corresponding method of the observer. Whether that is less confusing is disputable. In contrast, it is a pretty easy task to just write your own observer implementation with all the methods and arguments that serve you well. Bomberman’s model contains bombs, explosions, items, players, teams and more, each of which can be added or removed. Thus, the model notifies observers that implement 14 methods whenever any such object is added or removed with the object itself as argument. When the model notifies the view about a new bomb, the view immediately receives the bomb instance and is able to generate a corresponding UI object as indicated in **Figure 2**. As soon as the bomb vanishes again, the view receives it and deletes the corresponding UI. This is crucial for the view to always be able to visually represent the model appropriately with UI objects for each model-object.

My advice is to make each modifiable model object an observable using one of the described observer concepts. In Bomberman, bombs, explosions and items are not modifiable as they are just static objects where only the visual representation changes (of which the view takes care of). The model itself, players and teams, however, are modifiable. Players change their direction or get upgraded when catching an item. Teams gain or lose players. Bomberman’s score panel on the right

[5] http://beyond100classes.blogspot.ch/2012/01/observer-part-iv-generic-observer.html

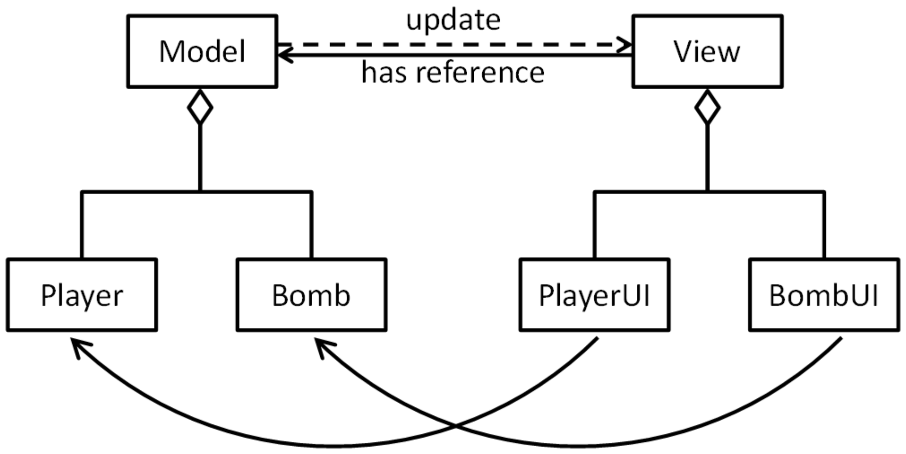

Figure 2: Association of model and view objects in Bomberman. The model contains all players, bombs and sc on. When a new Player is added, the model notifies the view which creates a PlayerUI that references the Player.

always shows the teams and the players. When a team is added to the model, the model notifies the score panel with the new team as argument. The score panel registers its observer at the team and repaints itself. If a team changes, it notifies the score panel which repaints itself. If a team is removed from the model, it notifies the score panel which removes its observer from the team and repaints itself. Using this approach, Bomberman is very extensible. Integrating multiplayer support and propagating changes over the network was very easy and kept the design smooth and clean. No type checking or other magic tricks were required.

Beware of multithreading! With today's computers providing more and more CPUs it is tempting to speed up independent computations by attaching them to multiple threads. This, however, might lead to complicated effects. One thread might add a new object to the model and start notifying observers. Another thread might remove that very object from the view again and start notifying the same observers as well. It is possible for the second thread to notify an observer that the object has been removed before the first thread has even notified it that the object has been added. From the observer's point of view, it looks as if an object is first removed and then added to the model which might lead to confusing results. Adding synchronization leads to more complicated code and is a potential bottleneck that does more harm than good. Without doubt, multithreading is and becomes even more important over time; yet, you must use it cautiously.

4 The Journey of an Image from the Disk to the Screen

The UI of Bomberman basically shows images that are loaded from the disk. Bomberman uses BufferedImages to use an image inside the program. Although each instance of classes BombUI, ItemUI and so on requires its image load from the disk, we do not want to repeatedly read data from the disk anytime such an instance is created. Therefore, Bomberman uses a singleton class ImageCenter that stores all the BufferedImages once loaded.

Images get from the disk to the ImageCenter in three steps. First, a static string leads to a configuration file. Second, the configuration file contains all paths to required images. Third, these images are loaded from ImageCenter. ImageCenter uses a class called GraphicsConfig. GraphicsConfig contains the path to the configuration file and loads all paths which ImageCenter then uses to load the images.

Bomberman's images can't just be displayed the way they are loaded. Since most of the UI objects are animations the corresponding image is a row of all frames of the animation. The bomb, for instance is a sequence of 16 images. Others like the explosion are more complicated. I do not like the idea that the UI class has to cut up such an image appropriately to show the current frame. Instead, Bomberman has a further class that takes a BufferedImage as constructor argument, cuts the image the way it is supposed to and provides methods to access the different frames. The explosion corresponding class, for instance, is ExplosionImage and it provides methods for the top, right, bottom and left end of an explosion plus the horizontal, vertical and center piece. This gives a good abstraction of the actual image structure and the frames that are displayed by the UI.

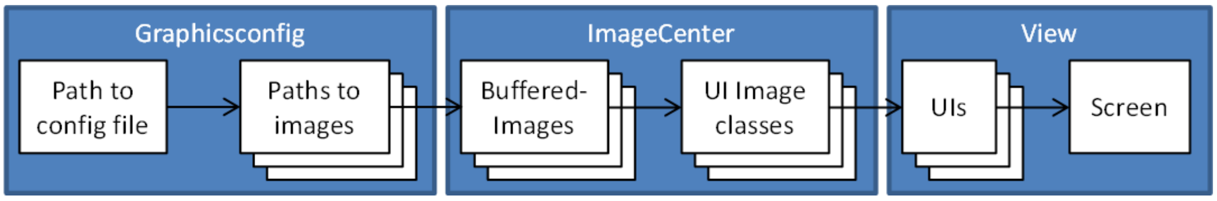

Figure 3: The journey of an image from the disk to the screen. A path leads to the configuration file that leads to all images. Images are loaded as BufferedImage and given to a UI image class that cuts them into pieces. The UI gets the appropriate frame from the UI image and paints it to the screen.

5 Conclusion

This article explains how the MVC pattern has been implemented in Bomberman. Model and view both consist of several classes. To each model class there is a corresponding UI class in the view. Different strategies for observer designs have been given. The controller listens to user gestures and propagates them to the model. The model implements its own observable interface and notifies the view component when it changes. Other modifiable model classes such as Player and Team implement Java's Observer interface. Unmodifiable classes such as bomb an explosion where only the UI changes do not need to implement an observer.

Images should be reused and not repeatedly loaded from the disk. In Bomberman, the ImageCenter keeps all images as BufferedImage. Specific classes cut the images into appropriate pieces and provide methods that the UI classes use to get a particular frame to display on the screen.