Rahul Rajagopalan
6.170

Code Design Critique

I used Model-View-Controller as the overarching architectural pattern when designing the game. The state of the game board and the rules of the game are encapsulated in an immutable Board class, which uses a closure to hide its internal representation. A GameModel object keeps track of the current Board, and provides an Observable interface. The GameView is connected to the GameModel only through the Observer pattern; the GameModel notifies it when it changes and provides it with a reference to the current Board. Because Boards are immutable, the view can only read it; it can not directly change the representation of the model. Players also act as observers to the Board; this reflects a symmetry in how humans and AI respond to moves, and allows the same model code to support any possible combination of human and AI players.

The view of the system is implemented as a HTML document displayed in a browser, styled by CSS. The associated JavaScript classes programatically generate the elements to be displayed (as it does not make sense to display the elements if JavaScript is disabled). The JavaScript does not style the HTML elements directly; it only adds classes and ids to them. The style is then added through an external CSS file.

The buttons and other input elements in the view have on-click events that are attached in JavaScript, and delegate to functions on controller objects. The controllers then map the user inputs to methods on the model. By using a controller, the view never directly calls the model, and the model will be able to operate unchanged even if the view was changed or removed.

While the Observer pattern is a standard idiom for connecting models and views, I consider it to be the weak link in the architecture. It was necessary to include calls to notifyObservers() in all of the model's mutators (code duplication), and tracing/debugging the cascade of events that occur whenever the model updates its state was relatively tedious. Unfortunately, it's difficult to model a system that changes over time, responds to user input, and communicates with other objects in a clean, functional way, and Observer is one of the more structured ways to do it; this implementation also minimized representation exposure by only passing around immutable objects.

I used the Dependency Injection pattern to connect all the objects together. That is, high-level objects in the system do not instantiate the other objects they communicate with; those other objects are instead provided in the constructor. A function in gameFactory.js handles the details of creating all the objects and connecting them. This makes it possible to program entirely towards interfaces (calling the new operator directly ties us to an implementation), and makes it easier to swap out objects.

While developing this application, the effects of modular design appeared in multiple ways. First, when designing the visual interface, I was able to add CSS directly to the stylesheet, only concerning myself with the classes of the HTML elements. It was not necessary to inspect the JavaScript code that created the elements. It would even be possible for someone who knows CSS but not JavaScript to change the visual design of this implementation of Othello.

Also, due to the decoupled design, swapping out implementations of components of the system was made easier. For example, the board was originally represented in HTML using a <table> element, but a LA pointed out that while the board was similar to a table in structure, it couldn't necessarily be considered tabular data. I replaced the BoardTableView class with a BoardDivView class that uses s instead BoardTableView is gone from the current application, but it can be

recovered from the git history log. When making this change, the vast majority of the diff was the new BoardTableView class; it was only necessary to change a few lines of instantiation logic in GameFactory plus a method rename for semantics in GameController, and the model code was completely untouched.

Appendix: Coding Conventions

JavaScript is a unique language in that there are multiple accepted conventions for certain tasks. I have annotated my code with type information that is read by the Google Closure Compiler. The intent is to use the compiler to detect possible sources of errors before the code runs (e.g. calling functions with the wrong type of input, misspelling variable names, and so on). The index.html file links to uncompiled JavaScript though, because of complications associated with running compiled code that depends on libraries that do not conform to the compiler's standards. The type annotations also serve as useful documentation for humans reading the code.

In addition, there are two ways of attaching methods to objects in use. The first declares the methods in the constructor, like so:

```
MyObject = function(arg1, arg2) {
  this.myFunction = function() { … }
};
```

Another way is to use prototypes, e.g.:

```
MyObject = function(arg1, arg2) { … }
MyObject.prototype.myFunction = function() { … }
```

The first way allows you to use closures to hide aspects of the object's representation, but has a significant performance penalty – the functions are duplicated every time an object is created from the constructor. (benchmarks can be found at http://jsperf.com/prototype-vs-closures/2) The second way avoids the penalty, and allows you to use the this keyword to explicitly distinguish object properties from instance variables, but does not allow you to privatize variables. In general, I have tried to get the benefits of both approaches by using the first way when necessary to enforce representation invariants, and the second way otherwise. This also minimizes the amount of code that has direct access to private variables, making it less likely that the representation will be exposed accidentally.