Game Patterns:
Singletons and Mediators

Singleton Pattern:

The Singleton Pattern is used to limit a class to one instance within an application, or even within an operating system or network. The standard technique for using the Singleton Pattern in Java is to do the following:

```java
public class Singleton {
    // Create one and only one instance of this class.
    private final static Singleton INSTANCE = new Singleton();

    // A private constructor prevents more instances
    // from being made outside the class
    private Singleton() {}

    // Returns the instance of this class created about in
    // the private static value
    public Singleton getInstance() {
        return INSTANCE;
    }
}
```

In general, the Singleton Pattern is used when only one object is needed to accomplish a task across a given system. Some people (see Wikipedia) have claimed the Singleton Pattern is a euphemism for the global variables Java tries so hard to avoid.

Singletons can be of great use in games. For example, a single Universe singleton could be implemented because there is obviously only one universe (unless your game deals with quantum mechanics). The game itself could be a Singleton object, because there is only one game running.

Thinking of uses for the Singleton Pattern alone can be kind of boring, but other design patterns use can use the Singleton Pattern in their implementations. The Factory, Builder, and Prototype are among the patterns Wikipedia listed that use Singletons. The Facade Pattern (which is being covered by someone else in class) can also use the Singleton Pattern.

Another interesting thing about Singletons is that they can use the Lazy Initialization Pattern to create the Singleton instance only when needed. Depending on the type of object the Singleton is modeling, this could be a significant performance gain.

Mediator Pattern:

The Mediator Pattern is used, surprisingly, to mediate communications and interactions between objects. Depending on the size and complexity of the program you are designing, there may be a large number of classes interacting with one another. Communication between classes can become very burdensome as your project grows.

The goal of the Mediator is to be the only class that knows the details of all the other classes in the project. This makes maintaining and adding features easier because only one class need be modified to make the changes system-wide.

Classes send messages to the Mediator when they have new information to present, and the Mediator informs its classes of the changes. This makes synchronizing application events much simpler. An example I found (linked on the Wikipedia page for the Mediator Pattern) involves a GUI application with buttons that enable and disable each other. The button begins by registering itself with
the Mediator, so the Mediator knows who is relying on it. When a button is pressed, it informs the Mediator object, which in turn enables or disables the other buttons. The button-level logic simply consists of telling the Mediator that it was clicked. It is the job of the Mediator to dispatch events based on that message.

For more information, visit: http://www.javacamp.org/designPattern/mediator.html