

Patterns and Delegation

Thanks to Nick Parlante for much of this handout

Patterns

- "Patterns" are about identifying common code idioms and giving them names, so they fall more easily to hand and are easier for others to read. Having a higher-level vocabulary of ready problem solving techniques helps get things done quicker. The patterns essentially provide better building blocks. Also, having the patterns available makes it easier to read and talk about bodies of code.
- The main patterns book is -- *Design Patterns: Elements of Reusable Object-Oriented Software* -- by Erich Gamma et al. Known as the "gang of four, Gof" book due to its four authors.
- IMHO, patterns can suffer a little from silver-bullet syndrome -- I know some engineers who are so focused on using Patterns as much as possible, the use of patterns becomes the goal rather than solving the actual problem at hand. That said, patterns are clearly a valuable technique and you should be familiar with them.

Idioms

- Identify common, useful code design patterns
- Give them a name
- Programmers get accustomed to the pattern and so work with it more easily
- Programmers can communicate with each other with the pattern vocabulary
- The most important patterns...

Factory

- The "factory" is a mechanism for clients to create/get new instances of some implementation without needing to know or depend on the particular implementing class. Rather than the client call `Foo f = new Foo()`, they call `Foo f = Foo.makeFoo()` -- "makeFoo()" is a "factory" method in this case, returning a new instance to the client. The exact class of what is passed back may vary at runtime to be a particular subclass of `Foo`, but the client does not need to know about that. This keeps things simple for the client, but allows us to be more sophisticated about what is passed back to them. For example, when testing, we can pass the client back a special stub of some sort. Since the client just calls `makeFoo()`, they can't tell.

Singleton

- Having exactly one of something -- e.g. the tetris pieces array. This is basically just the idea of having a single, global instance of something. Expose an interface such that the client code (or rather, various bits of client code, distributed across many modules) always interacts with the singleton. The singleton pattern is mocked somewhat as being so simple that it does not qualify as a pattern -- like saying Pluto is not really a planet.
- Suppose we want a singleton `Foo` object to exist. A common implementation is to have a static `getFoo()` method that returns a pointer to the single, static `Foo` object. All clients go through `getFoo()`, so they always get the one `Foo` object. The `getFoo()` method can use a lazy-evaluation strategy, only creating the `Foo` object the first time someone calls `getFoo()`.

Iterator

- Allow some to iterate over the elements in a collection (start, access each element, detect when at end) without knowing the collection implementation. In Java, this is done with the `Iterator` interface, which defines methods `hasNext()` and `next()`.

Adapter

- Start with an object that implements interface `X`. The "Adapter" wraps the `X` object, and translates between it and the rest of the world to make it look like interface `Y`. This is a form of delegation, implementing a different interface from the delegate.
- e.g. `HashMap` supports a `values()` method that returns a `Collection` of all the values in the `HashMap`. In reality, it does not construct an actual `Collection` of all the values. Instead, it builds a thin adapter object that implements the `Collection` interface and has a pointer to the original `HashMap`. When this adapter gets a message like `size()`, it passes it through to the underlying `HashMap`.

Decorator

- Start with an object that implements interface `X`. The "Decorator" wraps the original `X` object and also implements interface `X`, but with some variation. This is also a form of delegation.
- e.g. `ZipStreamReader` layered on `BufferedReader` layered on `FileReader`. This is a delegation, keeping the same interface as the delegate.

Delegation (vs. Inheritance)

- Both Adapter and Decorator are "delegate" strategies. These have the advantage that you can build up layers, where each layer deals with one aspect. This is a very flexible way to put together capabilities. Contrast it to using inheritance to put together capabilities, which is powerful but complicated. This leads to the modern preference of delegation over inheritance. See below.

Observer / Observable

- The `Observer` registers with the `Observable` that the observer would like to be notified when a particular thing happens to the observable.
- e.g. in Swing, "listeners" are used to coordinate components such as buttons with code to run when certain things happen to the component. In that case, the button is the "source", and a "listener" registers with the source to get "notifications". For example, the method `button.addActionListener(listener)` adds a listener to a button, and the button sends the notification `actionPerformed()` to the listener when the button is clicked.

Model View Controller (MVC)

- This is a pattern where the "model" takes responsibility for data storage, and the "view" takes responsibility for displaying the data from the model. The "controller" coordinates the two. We'll have a separate, detailed tour of the MVC pattern.

Subclassing vs. Delegation

Subclass Strategy

- Suppose there is an existing class, `Oracle`, that implements an `answer()` method
- We want a custom version of the `answer()` capability that adds "not" in front of the answer
- Inheritance solution: create our class as a subclass off `Oracle` and override/inherit `answer()`
- Our class `isa Oracle`
- Use overriding to customize

- Problem: introduces complex relationship between our class and `Oracle` -- we respond to all messages that `Oracle` responds to, we must mesh our methods with the methods up in `Oracle` -- can require a detailed understanding of `Oracle`.

Delegation Strategy

- Rather than subclass, just create an `Oracle` object and store a pointer to it in our class

```
class MyClass {
    private Oracle oracle;
    public MyClass() {
        oracle = new Oracle(); // allocate our "owned" Oracle delegate
    }
    // Re-send message to delegate
    public String answer() {
        return "not " + oracle.answer();
    }
}
```

- Rather than **isa** `Oracle`, our class **has** an `Oracle` (or "owns" an `Oracle`)
- When our class gets the `answer()` message, it turns around and sends it to the owned `Oracle` object
- The owned `Oracle` is a "delegate" -- we send messages to it so it can do the work
- We can exhibit the features of a class without the complexity of subclassing.
- Note that, in contrast to inheritance, our class has a simple client relationship with the delegate -- we are not exposed to how it is implemented. Writing client code is a simpler task than writing subclass code.
- Good: simpler than subclassing, may be used if you need to have some other superclass. Exhibit some of the features of a class, but not all of them.
- Bad: Requires writing the trivial sort of one-line "pass-back" methods that re-send messages to the delegate.
- The current thinking in the OOP community is that inheritance is a pretty fragile technique, and should only be used in circumstances that require it. Delegation is simpler and should be preferred in the cases where it is sufficient.
- e.g. the "Prefer Delegation" rule in Josh Bloch's book, *Effective Java*.

Wrapper Strategy

- Can also think of delegation as a "wrapper" strategy (aka "Adapter" pattern).
- e.g. `Arrays.asList(<array>)` returns an object that implements `List` by wrapping the (delegate) array
- e.g. `HashMap.values()` returns an object that appears to be a collection of the values in the map by wrapping the (delegate) map