latex input: mmd-article-header Title: Design Patterns Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Object Oriented Design, Object Oriented Programming, OOP, Java, Patterns CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2014 Ethan C. Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

Just a list of them at this point

Much of this list is from **An introduction to Object-Oriented Programming** (3rd ed.) by *Timothy Budd*.

1. **Adaptor** --- a *client* object needs a *service* but requires a specific *interface*, a *server* object provides the *functionality*, but *doesn't support* the right interface; instead of rewriting either client or server, create an adapter who **translates the results of the service provider into the language of the client**.
2. **Iterator** --- Reduces info client must know to access elements of a collection, it simply provides the interface (`bool hasMore()`, `Obj nextElem()`)
3. **Factory** --- e.g. the `Iterator iterator()` method of a class returns an `iterator` that knows how to iterate through elements of `this Collection`, but all we care about is that it implements the `Iterator` interface. So the `iterator()` method is a `Factory` that returns an object whose interface we care about, but whose implementation matches what we want to *do*, but we don't care *how* it manages to do so.
4. **Strategy** --- the client always wants to do the same thing (e.g. layout a grid, perform regression, assess a situation) and when it does that it assumes there is a special function to make it happen (e.g. layout(), regress(), assess()) but it wants to use a different *strategy* at different times, so there are a bunch of `Strategy` objects that implement the different strategies but provide the same interface for use.
5. **Singleton** --- Ensures that there will never be more than one instance of the class created.
6. **Composite** --- permit creation of complex objects using simple parts by allowing simple *components* to be nested arbitrarily. Like how `ViewControllers` and `Views` nest inside each other.
7. **Decorator** (aka *filter* or *wrapper*) --- for adding new functionality to objects; an alternative to inheritance. It *has-an* instance of the class it wrapps, but also adds new methods that it can pass to that instance. Or something. Didn't quite get it.
8. **Double-Dispatch** --- say that two *polymorphic variables*, `Shape` & `Device` need to be able to talk to each other. So we use this particular way of writing a

method on each shape to respond to each device.

```
def Printer.display(Shape shape)
  shape.displayOnPrinter(self) # Note "self" is a Printer
end

def Terminal.display(Shape shape)
  shape.displayOnTerminal(self)
end
```

Now we must add

```
def Triangle.displayOnPrinter(Printer p)
  # code #
end

def Triangle.displayOnTerminal(Terminal t)
  # code #
end
```

9. **Flyweight** --- share state in common between similar objects to reduce aggregate required storage. If each instance of a class points to the same data, then don't duplicate it in each object, instead create an instance of type `Class` and each instance of `MyClass` points to the info in the instance of `Class`. These `Class` objects are called *flyweights*.
10. **Proxy** --- similar to the *Adaptor* pattern. E.g. hide details of transmission protocols when communicating over a network to remote objects. Doesn't actually do much work, just changes the interface to make it more uniform (across e.g. protocols).
11. **Facade** --- Similar to the *Proxy* pattern, but where the *actual* work is done by a collection of *interacting* objects, rather than a single object. So the *facade* is the *intermediary* or *focal point* who hands off requests to the appropriate handler. This way we needn't even *remember* which object is meant to handle a particular thing. (This sounds like an excellent idea.)
12. **Observer** --- keep two objects synchronized with each other without direct knowledge of each other, e.g. if one is dynamically created & destroyed during execution (e.g. a view wrt a model). So we create an intermediary `ObserverManager` who communicates with the `model` and the `observers` who maintain lists of who they're observing, and the `view` can add itself to this list.