CSE 331: Design Patterns (2)

Builder (another Creational pattern)

Problem: constructors take arguments all at once, but sometimes we want to provide data piece-by-piece

Solution: create a helper class with setters to add data to it, then actually construct the object

Examples:

- `StringBuilder` in Java: call `append` piece-by-piece, then call `toString`
- a class `C` with 10 fields to initialize; `CBuilder` with 10 setters and a `build()` method
- a method `m()` with 10 arguments; `MArgsBuilder` with 10 setters

```
class C {
  private int x; private int y;  private int z;
  private String name; private boolean flag;
  public C(int x, int y, int z, String name,
           boolean flag) { ... }  }
... new C(0, 1, 2, "james", false) ...
```

```
class CBuilder {
  ... same field decls as C ...
  public C x(int x) { this.x = x; return this}
  ... setters for the other fields ...
  public C build() { return new C(this.x, ...); } }
new CBuilder().x(0).y(1).name("james").z(2).build();
```

Additional benefits: reordering, default values, separating mutation from aliasing...

Structural Patterns: Wrappers, which are thin veneers over an encapsulated class

Pattern	Functionality	Interface
Adapter	same	different
Decorator	different	same
Proxy	same	same

Adapter

Problem: want to pass an instance of class C where an instance of interface I is expected, and C doesn't implement I

Solution: create a wrapper for C that does implement I; can do so via subclassing or delegation

```
interface Sizeable { public int size(); }
class ArrayListSizeable<T> extends ArrayList<T> implements Sizeable {}
public void m(Sizeable s) { ... }
ArrayList<Integer> l = new ArrayList<>();
m(l);  // error: ArrayList does not implement Sizeable
m(new ArrayListSizeable(l))  // works, and has all ArrayList functionality, too
```

In Java, even if a class has exactly the right methods, its instances cannot be passed where some interface is expected. Hence Adapters are quite common in Java codebases that use multiple unrelated libraries.

Decorator

Problem: Several implementations of an interface may want to share some functionality, but not have a common superclass

Solution: create a wrapper class that adds the desired behavior to any other class

```
interface Window {
  public Rectangle bounds();
  public void draw(Screen s);
}
class WindowImpl implements Window { ... }
// adds functionality:
new BorderedWindow(new WindowImpl())
```

```
class BorderedWindow implement Window {
  private Window innerWindow
  public BorderedWindow(Window innerWindow) { ... }
  void draw(Screen s) {
    innerWindow.draw(s);
    innerWindow.bounds().draw(s);
  }  }
```

Proxy

Problem: want to add "unrelated" functionality to an existing object

Solution: create a wrapper class with the unrelated functionality

Examples:

- make an object available remotely over the network
- permit access only if authorized
- avoid object creation unless needed (creation may be expensive)

Subclassing vs Delegation/Composition

All of the structural patterns above can be implemented either by subclassing or by delegation.

Subclassing automatically gives access to all superclass methods, and is usually less code to write. But subclasses are tightly coupled to superclass, and it is harder to remove functionality.

Delegation/Composition is more verbose but more flexible: can easily remove methods, not tied to class hierarchy. You can even wrap/unwrap objects at runtime, and compose multiple wrappers.