Design Patterns

Design at different scales

- Design of individual classes is "easy"
  - Identify the class's operations
  - Specify, implement, and test them
- Design of whole application is hard
  - "Software architecture"
  - Very application-specific
- In between: design of coordinated groups of classes
  - Some commonly occurring **design patterns**

Design patterns

- Identify a standard programming goal
  - "Want to be able to change GUI without affecting main code"
- Describe a way of designing a few interrelated classes to achieve the goal
  - "Have a subject class separate from an observer class, with the following operations…"
- Point out trade-offs
  - "Good when simple protocol between subject and observer, not so good otherwise"

Benefits of design patterns

- Pass on "standard wisdom" from experienced to novice designers
  - What are some good solutions
  - What are their strengths and weaknesses
  - What to look for to decide which pattern to choose
- Give names to standard patterns, to ease communication

A pattern: Observer

- Motivation: achieve loose coupling between data and its external views
  - data = "**subject**" (aka model, publisher)
  - view = "**observer**" (aka subscriber)
- Any number of observers of model
- Observers notified when model changes, without model having to know (statically) about the observers

Subject participant

```
abstract class Subject {
   private List<Observer> observers;
   void Attach(Observer o) {
     add o to observers; }
   void Notify() {
     foreach o in observers: o.Update(); }
}
class SomeConcreteSubject extends Subject {
  store data, invoke Notify() when changed
}
```

Observer participant

```
abstract class Observer {
  private Subject subject;
  void Update();
}
class SomeConcObserver extends Observer {
  display view of subject;
  update view when Update called;
}
```


Applicability

- When an abstraction has two aspects, one dependent on the other
- When a change in one object requires changing others, but you don't know how others need to be changed
- When an object should be able to notify other objects without knowing who those objects are

Example: DB system

- What are the subjects & observers in the GUI interface to a generic DB system?
  - When might there be multiple observer classes of a single subject class?
  - What advantages to separating subject from observer classes?

Benefits: modularity & reuse

- Encapsulate subjects and observers separately
  - Allow changing implementation, design decisions independently
- Reuse subjects and observers independently
- Add new observers independently of subjects

Liabilities

- Unexpected updates, spurious updates
  - Generally, coordinating when & in what order to do updates of observers if there is a sequence of updates to subject
- Limited Update() protocol
  - No information about what part of subject changed
- Avoiding Update() in the middle of an "atomic" change to subject

A pattern: Template Method

- Motivation: OO design!
  - describe the skeleton or default behavior of an algorithm in one method of a superclass
  - let subclasses instantiate/refine its behavior for their specific context

Participants

```
abstract class Skeleton {
  TemplateMethod(…) {
      … SubMethod1(…) … SubMethod2(…) … }
  SubMethod1(…) { … } // or abstract
  SubMethod2(…) { … } // ditto
}
class Refinement extends Skeleton {
  SubMethod1(…) { … }  // refinement code
  SubMethod2(…) {    … }  // ditto
}
```


Applicability

- To implement a generic algorithm once, and leave the parts that can vary to subclasses
- To factor common behavior into a shared place, while still allowing some differences
- To provide a framework that controls subclass extensions, via "hooks"

A pattern: Adapter

- Motivation: to make two different libraries, with different assumptions, work together
  - Their interfaces aren't compatible
    - E.g., if subjects and views were written separately by third-parties, and then we wished to combine them
- Can adapt classes statically, or objects dynamically

Participants (class adapter)

- Client: sends Target.Request()
- interface Target: defines Request()
- class Adaptee: defines OtherRequest()
  - Something different than Client wants
- class Adapter implements Target
  extends Adaptee {

```
  Request() { this.otherRequest(); }
}
```


Applicability (class adapter)

- Want to use a class, and its interface isn't what you want
- Can make a subclass of every bad interface, and change all instantiations from the original class to the new class

Benefits (class adapter)

- Allows reuse & integration of independently developed libraries

Liabilities (class adapter)

- Potential explosion in # of adapter subclasses (one per adaptee class)
- Need to modify all creations of adaptee classes to be adapter classes instead

Participants (object adapter)

- Client, Target, Adaptee: as before
- class Adapter implements Target {

```
    private Adaptee adaptee;
    Request() { adaptee.otherRequest(); }
  }
```


Applicability (object adapter)

- As with class adapter, but where impractical to make subclasses of every class to be adapted, and/or have to adapt instances of the original class after they've been created

Benefits (object adapter)

- Class adapter benefits, plus additional flexibility in handling instances of Adaptee classes directly

Liabilities (object adapter)

- Extra object creations, forwarding of messages
  - No such overhead with class adapter
- Object identity etc. can be tricky to get right
  - Ditto
- Cannot override methods of adaptee as easily as with class adapter
- Two-way peer-to-peer adaptation?

Summary (so far)

- Design patterns identify good programming techniques
- Most are known widely at a general level, but:
  - Point out subtleties & choices in how they're fleshed out
  - Point out pro's and con's
  - Point out implementation trade-offs
- Give all this a concise name