Software Engineering 301:
Software Analysis and Design

Design patterns

Agenda

- Basic concepts
- Example patterns
- Common misunderstandings
- Matching against an implementation

Similar problems: similar solutions

Similar solution != similar problem

What's a design pattern?

- "describes a problem which occurs over and over again in our environment,
- "describes the core of the solution to that problem,
- "in such a way that you can use this solution a million times over, without ever doing it the same way twice"

Agenda

- ~~Basic concepts~~
- Example patterns
- Common misunderstandings
- Matching against an implementation

Three patterns

- Three patterns will be of use to us
  - Facade
  - Singleton
  - Observer

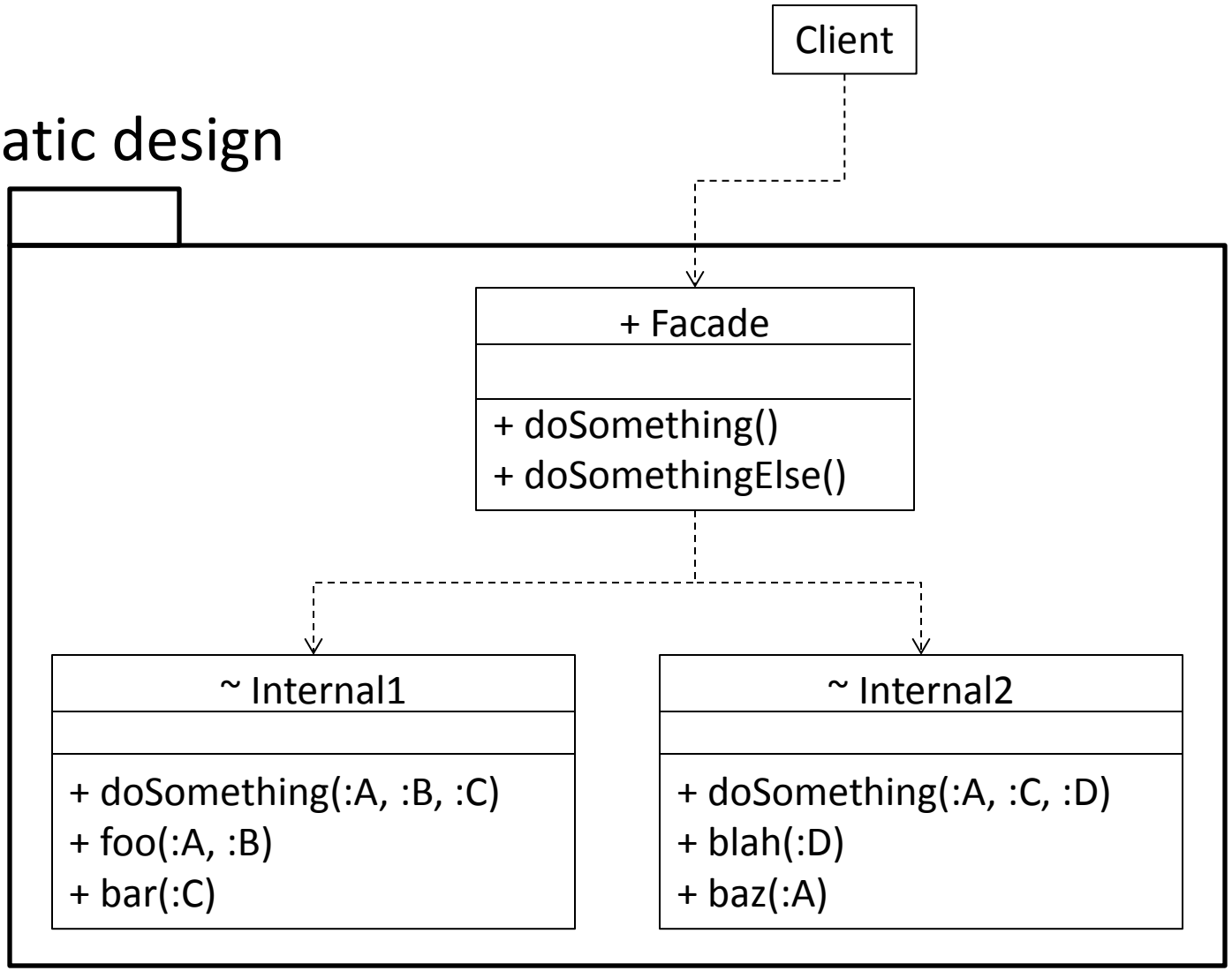
- NOTE: MANY patterns are currently recognized; new ones will surely be found in the future

Facade

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use

Facade

- Static design



Facade

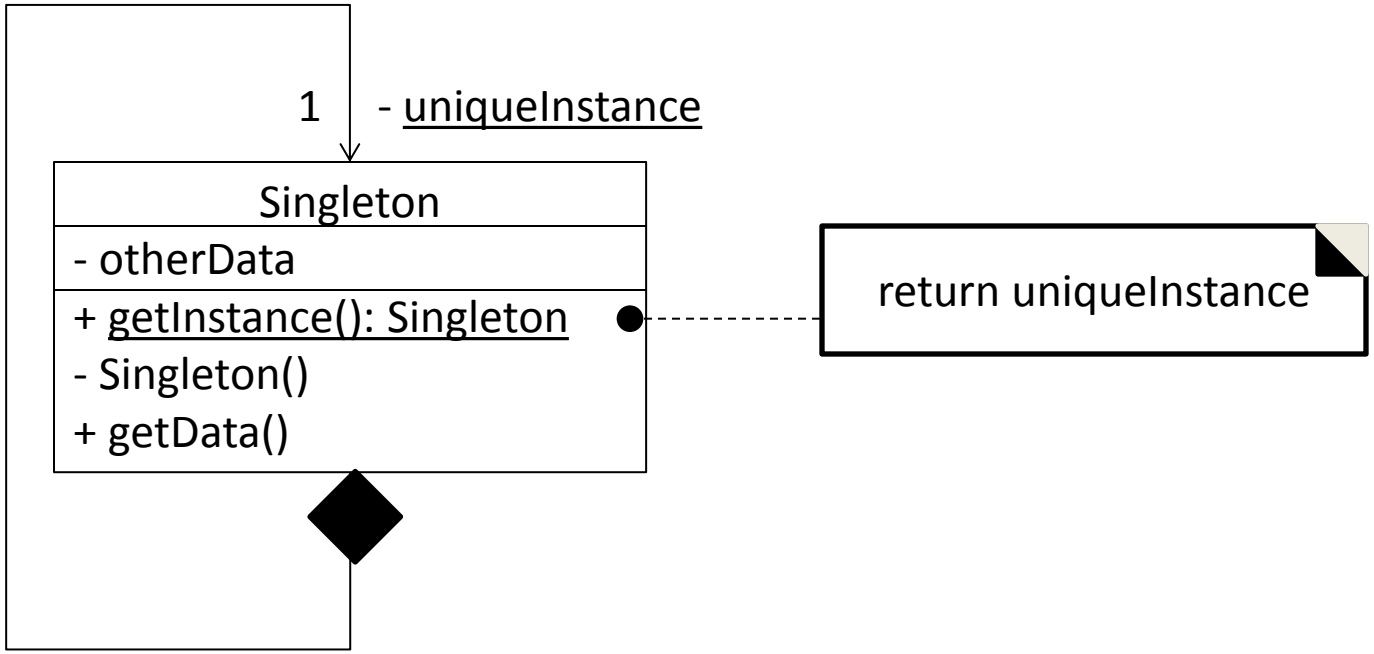
- Consequences
  - Shields clients from internal classes
  - Promotes weak coupling
  - Doesn't actually hide the internal classes
    - Good, when you need the extra control
    - Bad, when you need to change the internal implementation
    - BUT, you can make the internal classes package-protected if you want

Singleton

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it

Singleton

- Static structure



Singleton

- Typical Java implementation:

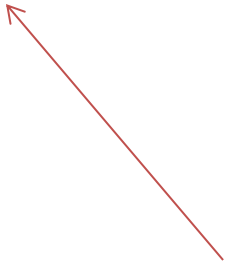
```
public MyClass {
        private static MyClass instance = new MyClass();
        private MyClass() {}
        public static MyClass getInstance() { return instance; }
        public void someOtherMethod() { … }
}
```

Singleton

- Consequences
  - Contolled access
  - Reduced namespace
  - Polymorphism supported
- Issues
  - Subtyping
    - Possible, but must configure uniqueInstance
  - Variable number of instances
    - Store multiple instances, track which are used

Observer

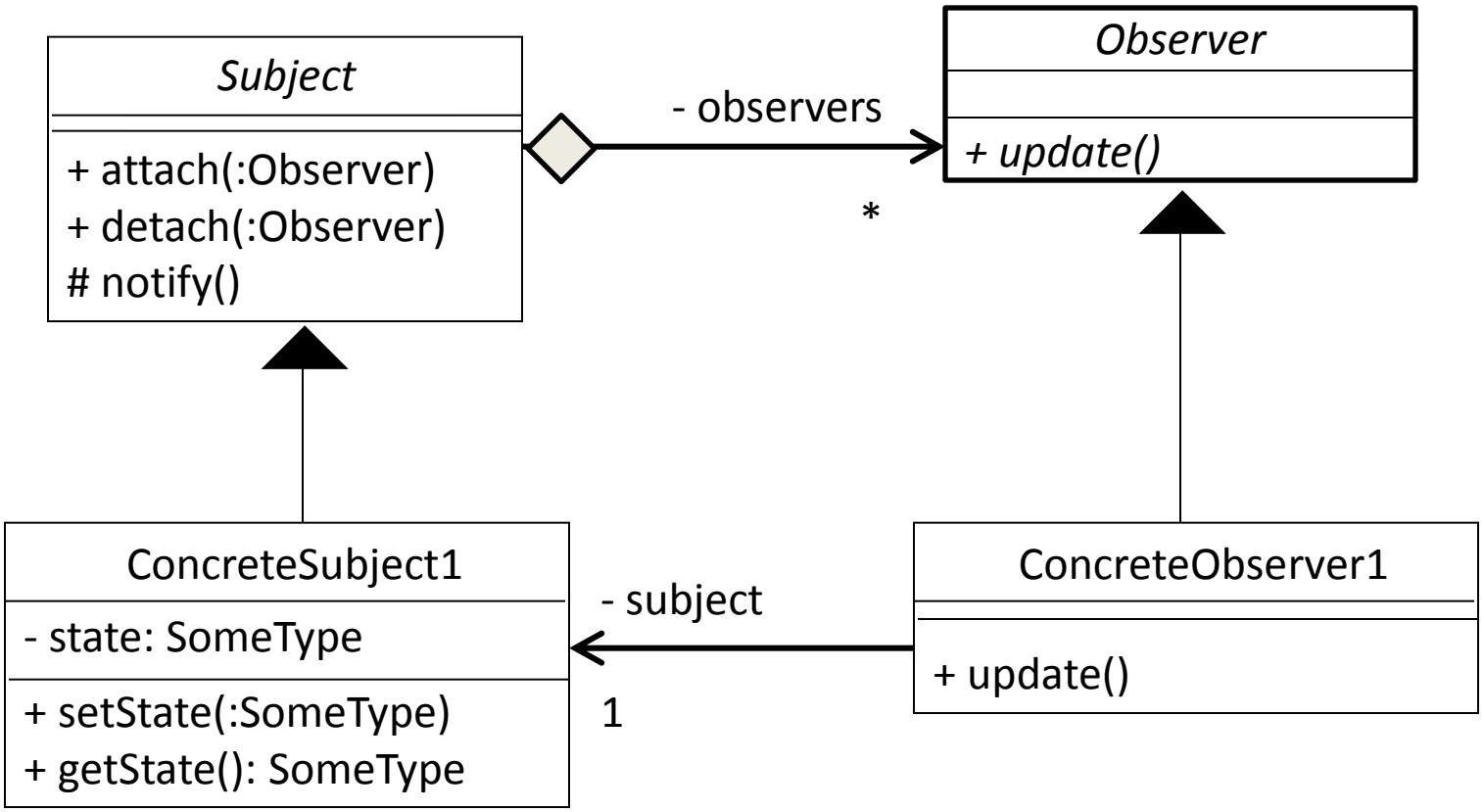
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically



when an event occurs

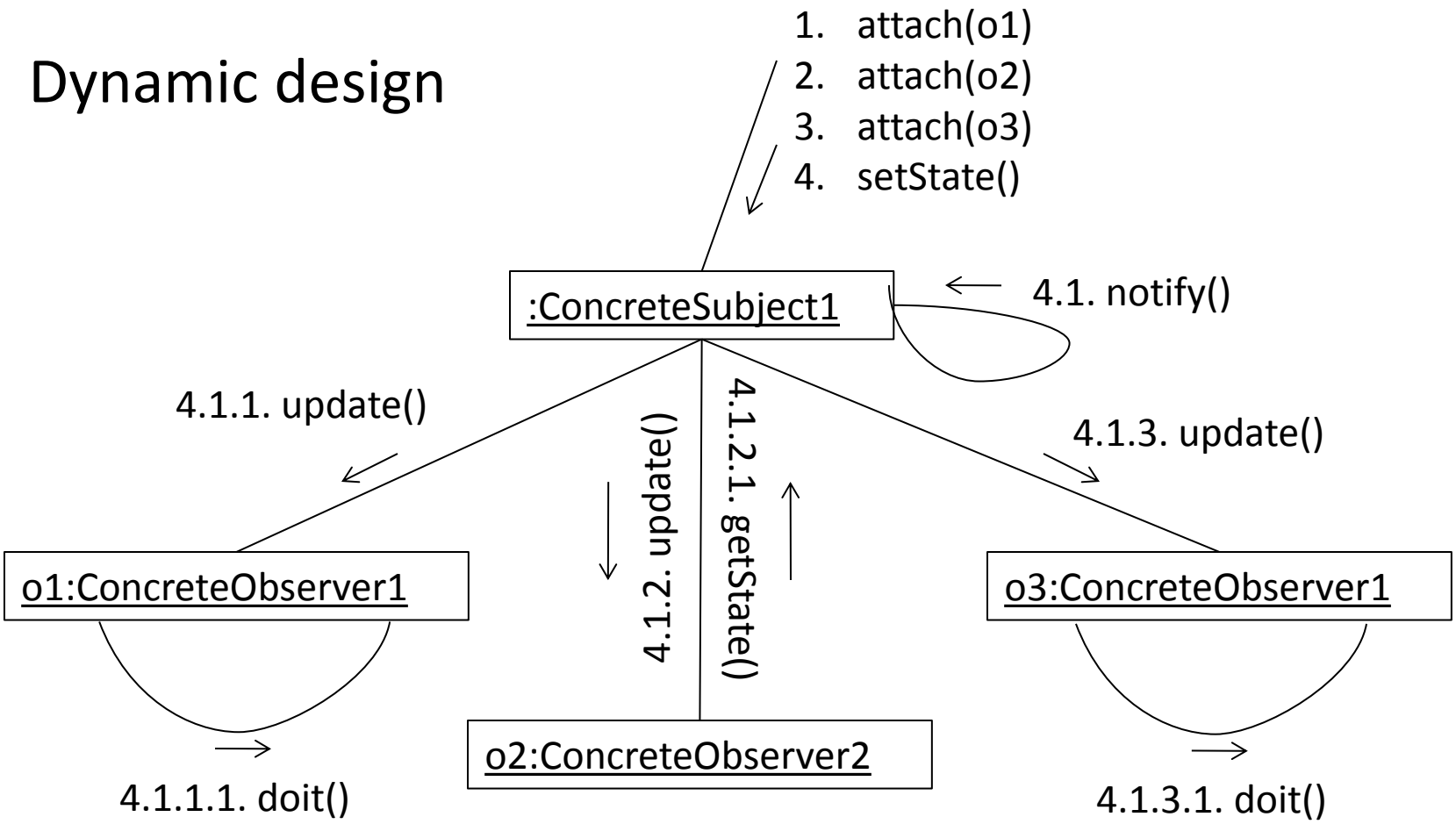
Observer

- Static design



Observer

- Dynamic design



Observer

- Example

```
public class Foo {
   private Vector<Listener> listeners =
      new Vector<Listener>();
   public void register(Listener l) {
      listeners.add(l);
   }
   public int doSomething {
      announceSomething();
      …
   }
   private void announceSomething() {
      for(Listener l: listeners) {
         l.somethingHasHappened(this);
      }
   }
}
```

```
public interface Listener {
   void somethingHasHappened(Foo foo);
   void otherEvent();
}

public class FooWatcher implements Listener {
   public FooWatcher(Foo foo ) {
      foo.register(this);
   }

   public void somethingHasHappened(Foo foo) {
      // ignore it
   }

   public void otherEvent() {
      …
   }
}
```

Observer: consequences

- Abstract coupling between Subject and Observer
  - ConcreteSubject knows very little about each ConcreteObserver:
    - i.e., Observer implements an update() method
  - Rest of a ConcreteObserver can change without affecting ConcreteSubject
  - We can add new ConcreteObserver classes, remove old ones, and modify existing ones
- Support for broadcast communication
  - No need for ConcreteSubject to worry about number of objects interested in its changes
  - Permits the number of objects to change during runtime

Observer: consequences

- Circular dependencies can be problematic
- Event-notification protocol needed
  - What kinds of events should the Subject announce?
  - Should the Subject announce every event that it recognizes to every Observer registered with it?
  - Should the Subject define different kinds of events and allow Observers to register an interest in each kind, independently?

Agenda

- ~~Basic concepts~~
- ~~Example patterns~~
- Common misunderstandings
- Matching against an implementation

Common misunderstandings about design patterns

- A design is never simply a collection of design patterns
- The implementation of a design pattern will not be standardizable
- An implementation may have one class playing multiple roles, or multiple classes playing a single role
- Other names may be used in an implementation

Common misunderstandings about design patterns

- The presence of design patterns does not **imply** "a good design"
- Every design pattern has consequences:
  - depending on the concrete situation, these may be **good or bad**
  - choose wisely: replacing/removing a design pattern is difficult!

Agenda

- ~~Basic concepts~~
- ~~Example patterns~~
- ~~Common misunderstandings~~
- Matching against an implementation

Identifying roles

- Each pattern talks about a set of roles
- Variations in the application of a pattern may:
  - use different names
  - eliminate roles
  - combine roles
  - add extra stuff in, to realize actual functionality

Example

- From Singleton, I gave this example:

```
public MyClass {
        private static MyClass instance = new MyClass();
        private MyClass() {}
        public static MyClass getInstance() { return instance; }
        public void someOtherMethod() { … }
        public int m2(MyClass foo, Bar bar) { … }
}
```

plays role of "Singleton" class

plays role of "uniqueInstance" field

plays role of private constructor

plays role of "getInstance" method

both play role of "getData" method

Example 2

getState, setState, and state have no analogues here

- From Observer

Concrete Subject AND Abstract Subject

```
public class Foo {
   private Vector<Listener> listeners =
      new Vector<Listener>();
   public void register(Listener l) {
      listeners.add(l);
   }
   public int doSomething {
      announceSomething();
      …
   }
   private void announceSomething() {
      for(Listener l: listeners) {
         l.somethingHasHappened(this);
      }
   }
}
```

"attach" method

"notify" method

Abstract Observer

```
public interface Listener {
   void somethingHasHappened(Foo foo);
   void otherEvent();
}
```

both are "update" method (abstract)

ConcreteObserver

```
public class FooWatcher implements Listener {
   public FooWatcher(Foo foo ) {
      foo.register(this);
   }

   public void somethingHasHappened(Foo foo) {
      // ignore it
   }

   public void otherEvent() {
      …
   }
}
```

both are "update" method (concrete)

Next time

- Design principles