Design Patterns

Quan Thanh Tho

Agenda

- Design Patterns
- Structures
- Popular Design Patterns
  - MVC (lightweight)
  - Adapter
  - Observer

Design Patterns

- A **Design Pattern** systematically names, explains, and implements an important recurring design.
- These define well-engineered design solutions that practitioners can apply when crafting their applications

Why Design Patterns

- Good designers do not solve every problem from first principles. They reuse solutions.
- Practitioners do not do a good job of recording experience in software design for others to use. Patterns help solve this problem.

Classic Design Patterns

- Published as a book in 1995
- Design Patterns is essentially a catalog of 23 commonly occurring problems in object-oriented design and a pattern to solve each one.
- The authors are often called the Gang of Four (GoF)

Organization

Behavioral
- Observer
- Adapter
- Chain of Responsibility
- Template Method
- Strategy
- Command
- State

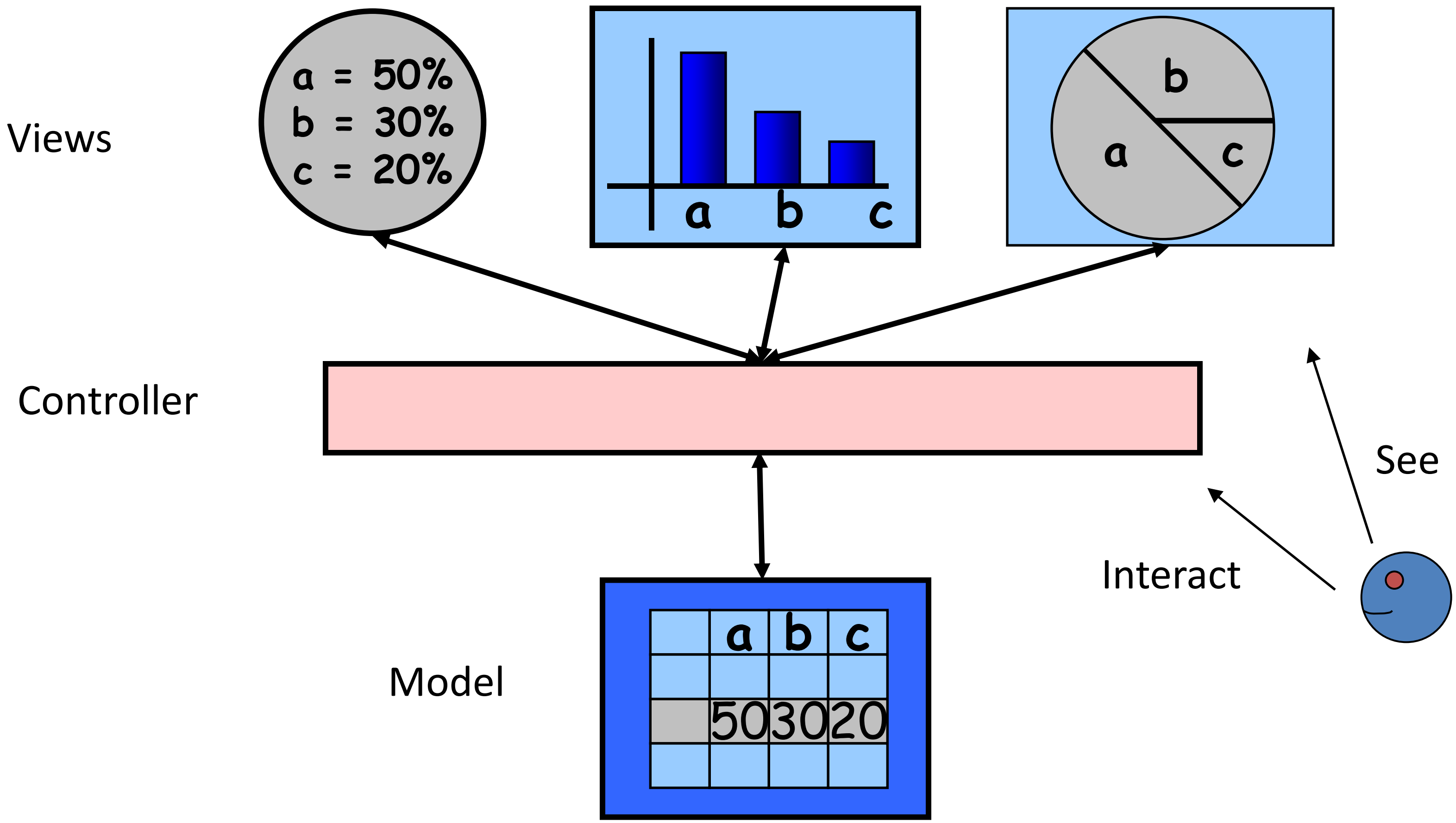
Structural
- Façade
- Composite
- Proxy
- Decorator
- Flyweigth
- Façade

Creational
- Abstract Factory
- Factory Method
- Singleton
- Prototype
- Singleton
- Builder

Popular Design Patterns

- MVC
- Adapter
- Observer

MVC



[Mancoridis picture]

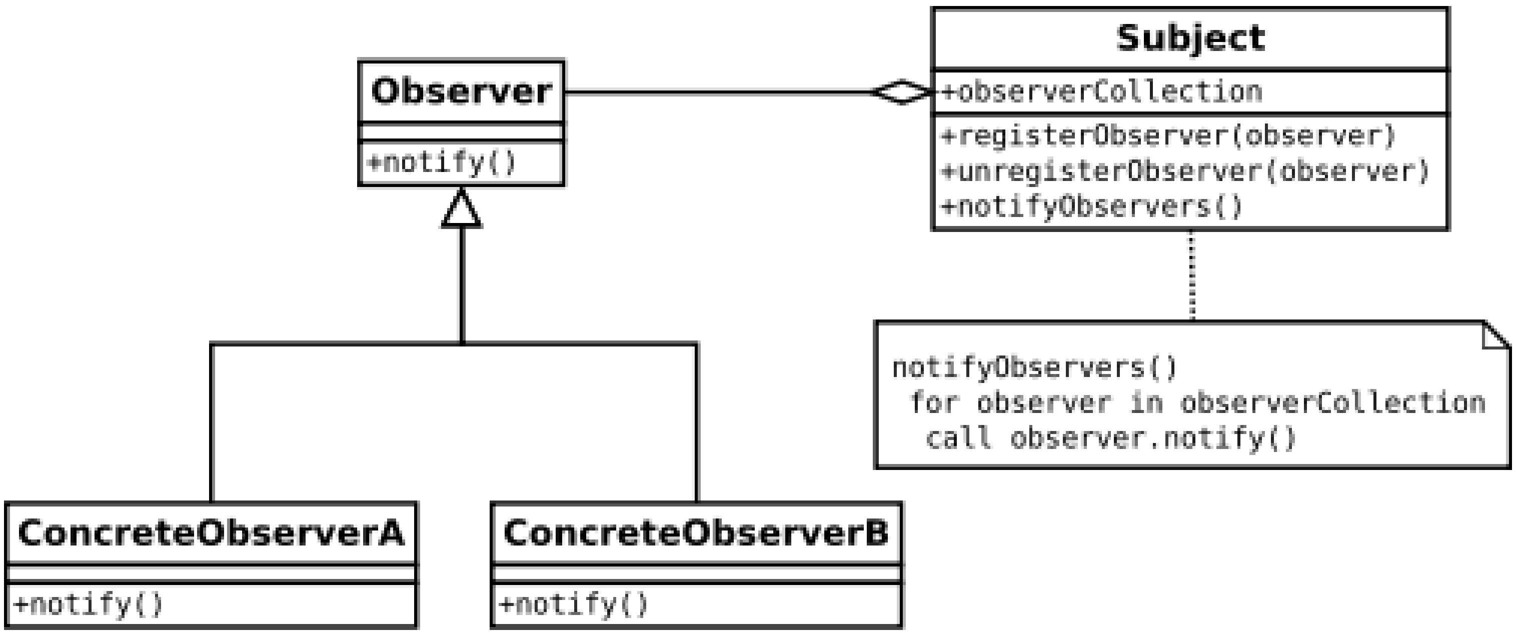
Multiple View Problem

- Need to keep all the views consistent
- If user (or one of users) changes a view, all other views should be updated

Implementing MVC

- Where is list of views (observers) kept?
- How is notification of change transmitted?
- Should a view ask for (or should it be told of) details about changes?

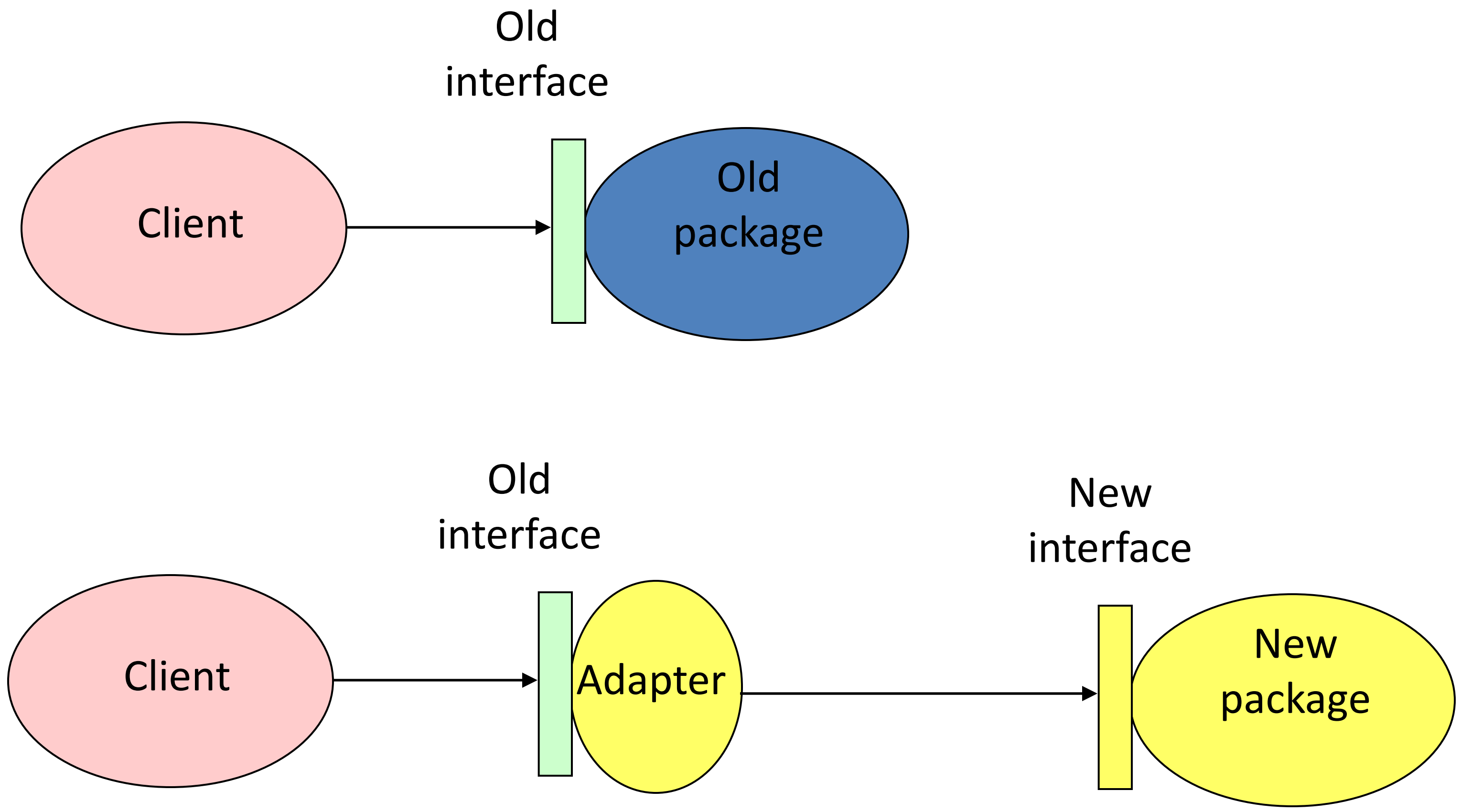
Observer


Observer
+notify()
Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
notifyObservers()
for observer in observerCollection
call observer.notify()
ConcreteObserverA
+notify()
ConcreteObserverB
+notify()

Adapter Pattern

- You have an existing client (application) that uses an old interface to an existing support package.
- You are given a new interface to a new support package
- You need to produce an adapter so that:
  - The client can use the new interface instead of the old one (without changing the client)

Illustration



Sensor Problem

```
class TS7000 {
    native double getTemp();
    ...
    }

double sum = 0.0;
for (int i = 0; i < sensors.length; i++)
    sum += sensors[i].getTemp();
double meanTemp = sum / sensors.length;
```

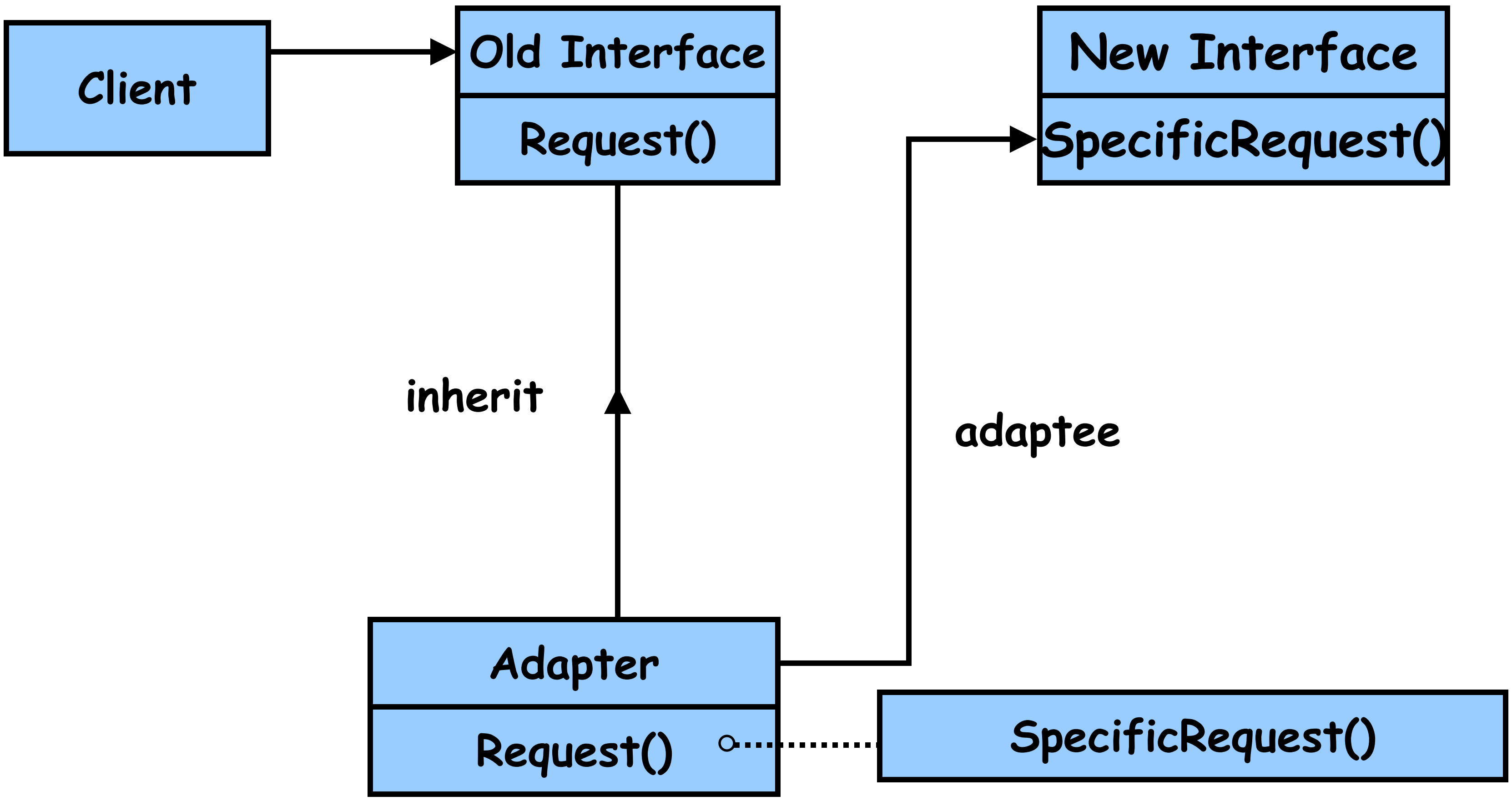
New Sensor Device

```
class SuperTempReader {
    //
    // NOTE: temperature is Celsius tenths of a degree
    //
    native double current_reading();
    ...
    }
```

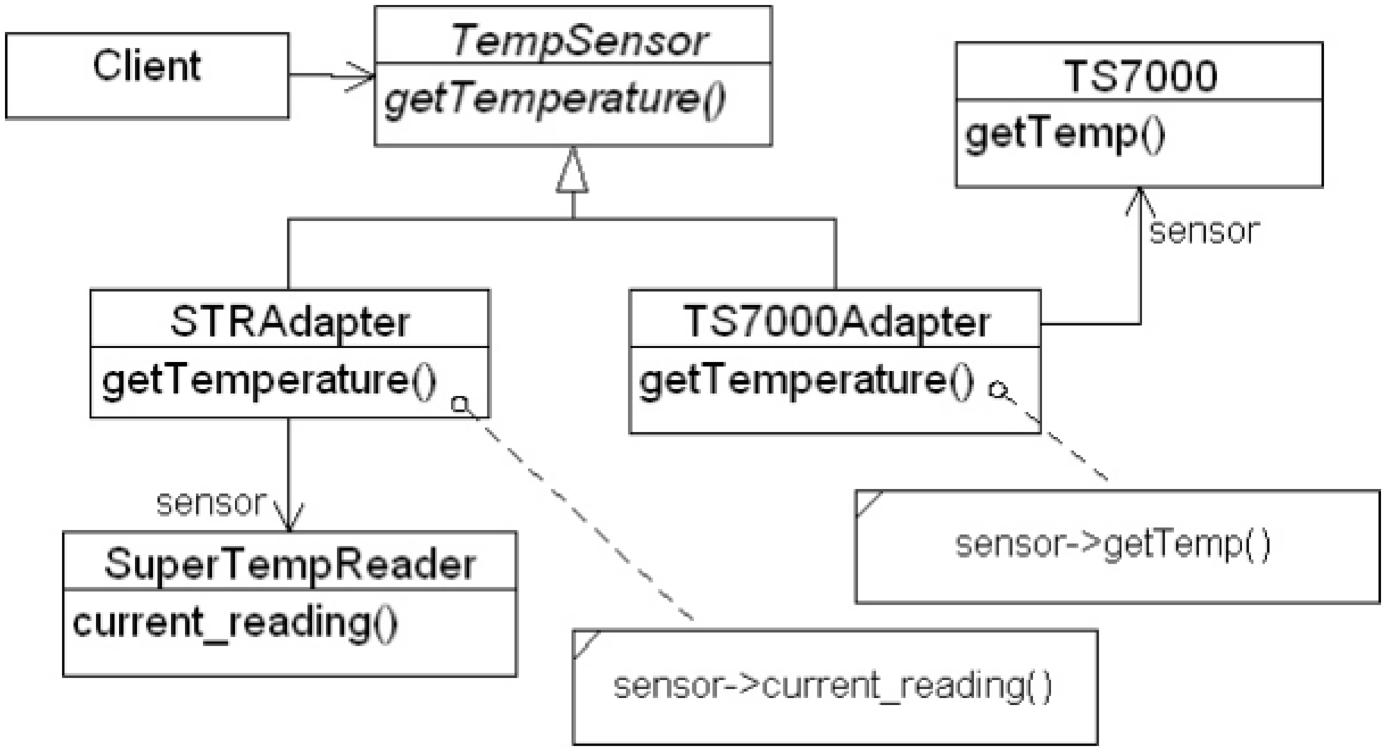
Without Adapter

```
for (int i = 0; i < sensors.length; i++)
{
   if (sensors[i] instanceof TS7000)
      sum += ((TS7000)sensors[i]).getTemp();
   else
      // Must be a SuperTemp!
     sum +=
        ((SuperTempReader)sensors[i]).current_reading() *
   10;
}
```

Adapter Pattern



Applied in the Situation

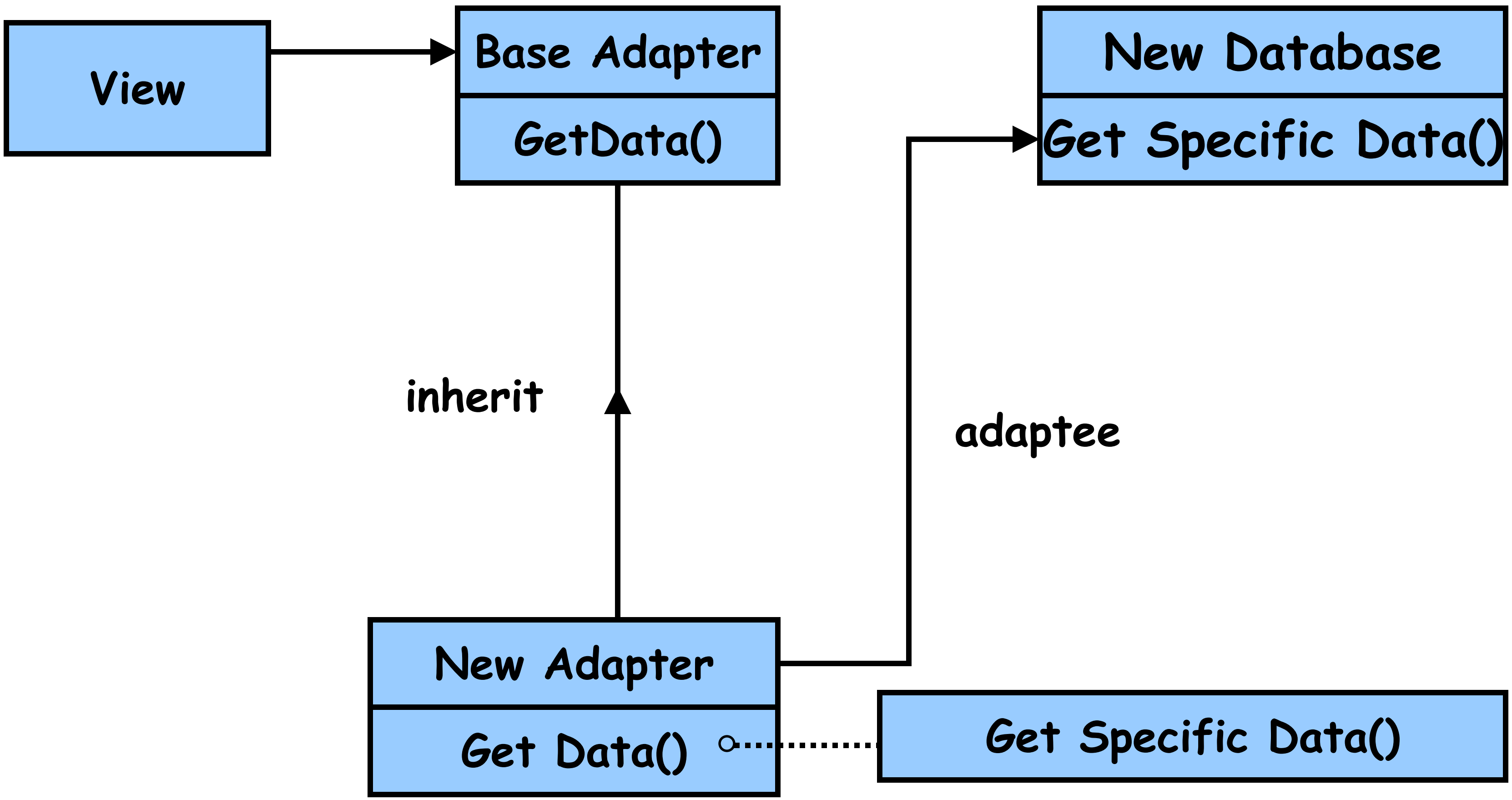


Adapter Implementation

```
abstract class TempSensor
{
   abstract double getTemperature();
}
class STRAdapter extends
     TempSensor
{
   public double getTemperature()
   {
      return sensor.current_reading()
     * 10;
   }
}
```

```
class TS7000Adapter extends
     TempSensor
{
   public double getTemperature()
   {
      return sensor.getTemp();
   }
}
…
…
double sum = 0.0;
for (int i = 0; i < sensors.length; i++)
   sum +=
     sensors[i].getTemperature();
```

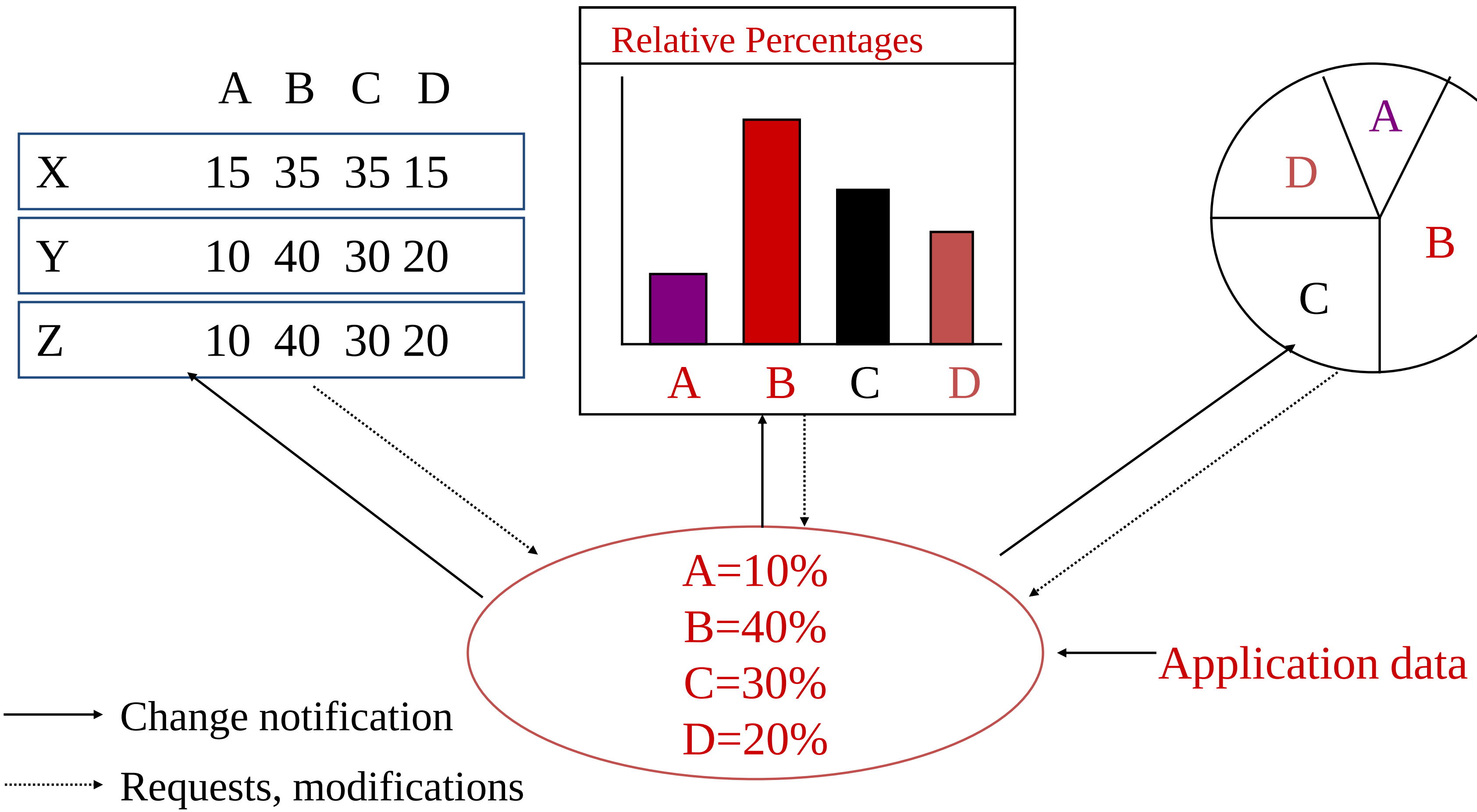
Adapter Implementation in Android



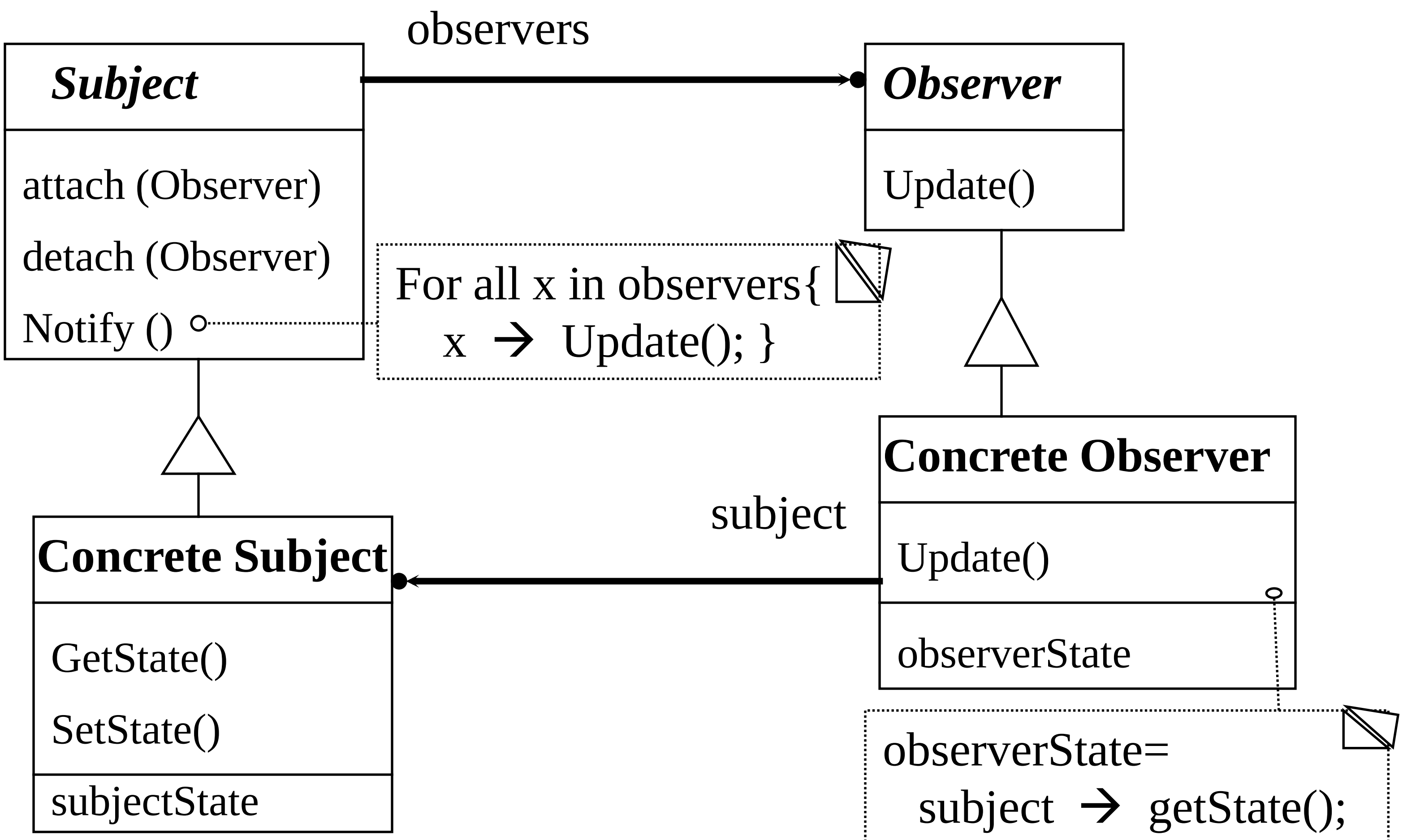
Observer Pattern [1]

- Need to separate presentational aspects with the data, i.e. separate views and data.
- Classes defining application data and presentation can be reused.
- Change in one view automatically reflected in other views. Also, change in the application data is reflected in all views.
- Defines one-to-many dependency amongst objects so that when one object changes its state, all its dependents are notified.

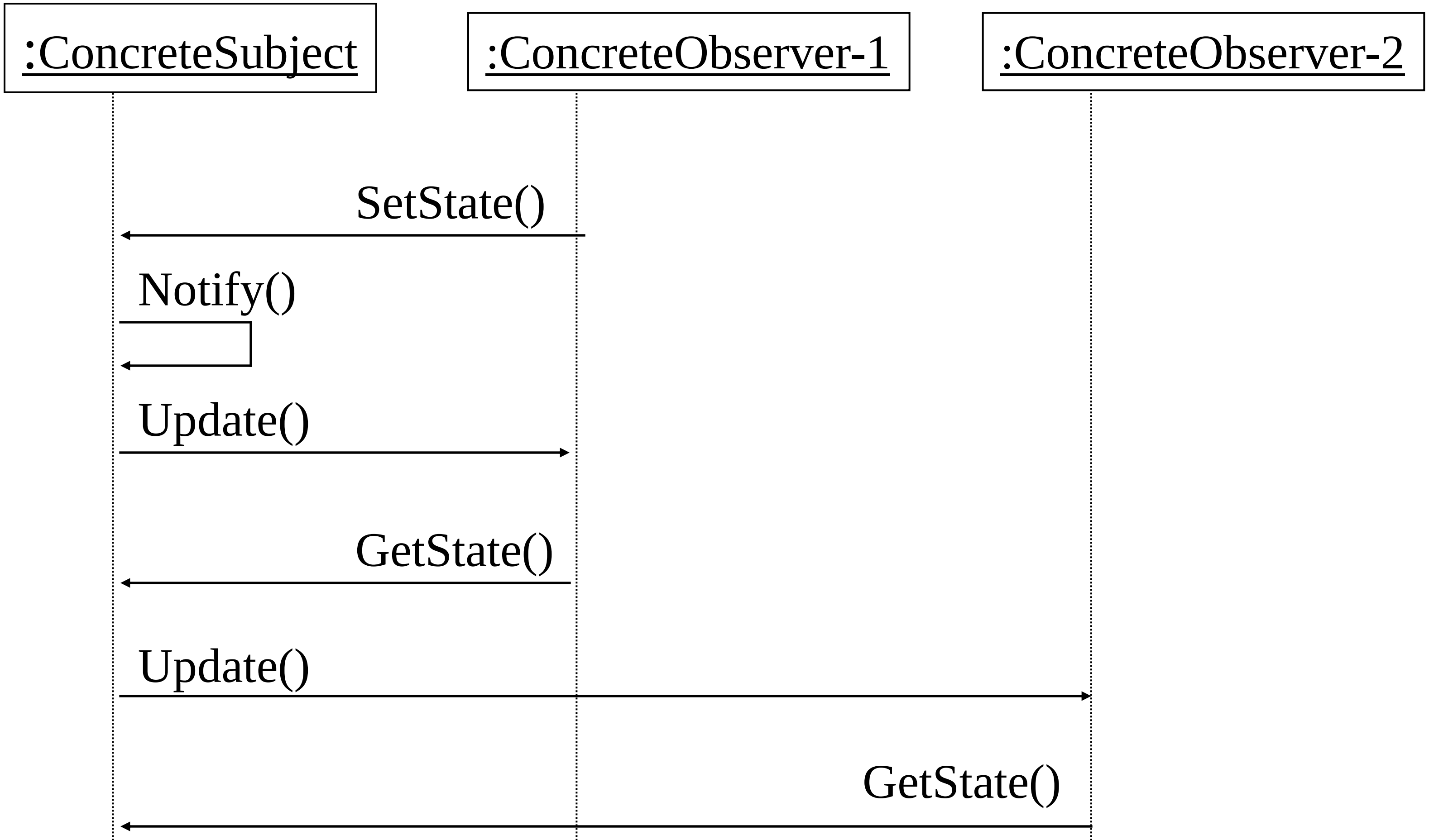
Observer Pattern [2]



Observer Pattern [3]



Class collaboration in Observer



Observer Pattern: Observer code

```
class Subject;

class observer {
public:
        virtual ~observer;

        virtual void Update (Subject* theChangedSubject)=0;

protected:

         observer ();

};
```

Abstract class defining the Observer interface.

Note the support for multiple subjects.

Observer Pattern: Subject Code [1]

```
class Subject {

public:
        virtual ~Subject;
        virtual   void Attach (observer*);
        virtual   void Detach (observer*) ;
        virtual   void Notify();
protected:
        Subject ();
private:
        List <Observer*> *_observers;
};
```

Abstract class defining the Subject interface.

Observer Pattern: Subject Code [2]

```
void Subject :: Attach (Observer* o){
        _observers -> Append(o);
}
void Subject :: Detach (Observer* o){
        _observers -> Remove(o);

}
void Subject :: Notify (){
        ListIterator<Observer*> iter(_observers);
        for ( iter.First(); !iter.IsDone(); iter.Next()) {
                iter.CurrentItem() -> Update(this);

        }
}
```

Observer Pattern: A Concrete Subject [1]

```
class ClockTimer : public Subject {
public:
        ClockTimer();
        virtual int GetHour();
        virtual int GetMinutes();
        virtual int GetSecond();
        void Tick ();
}
```

Observer Pattern: A Concrete Subject [2]

```
ClockTimer :: Tick {

    // Update internal time keeping state.
    // gets called on regular intervals by an internal timer.

            Notify();

}
```

Observer Pattern: A Concrete Observer [1]

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
    virtual void  Update(Subject*);
    virtual void  Draw();
private:
    ClockTimer* _subject;
    }
```

Override Observer operation.

Override Widget operation.

Observer Pattern: A Concrete Observer [2]

```
DigitalClock ::DigitalClock (ClockTimer* s) {
        _subject = s;
        _subject→Attach(this);
}

DigitalClock ::~DigitalClock() {
        _subject->Detach(this);

}
```

Observer Pattern: A Concrete Observer [3]

```
void DigitalClock ::Update (subject* theChangedSubject ) {
      If (theChangedSubject == _subject) {
            Draw();
      }
}

void DigitalClock ::Draw () {
  int hour = _subject->GetHour();
  int minute = _subject->GeMinute(); // etc.
  // Code for drawing the digital clock.
}
```

Check if this is the clock's subject.

Observer Pattern: Main (skeleton)

```
ClockTimer* timer = new ClockTimer;

DigitalClock* digitalClock = new DigitalClock (timer);
```

When to use the Observer Pattern?

- *When* an abstraction has two aspects: one dependent on the other. Encapsulating these aspects in separate objects allows one to vary and reuse them independently.

- *When* a change to one object requires changing others and the number of objects to be changed is not known.

- *When* an object should be able to notify others without knowing who they are. Avoid tight coupling between objects.

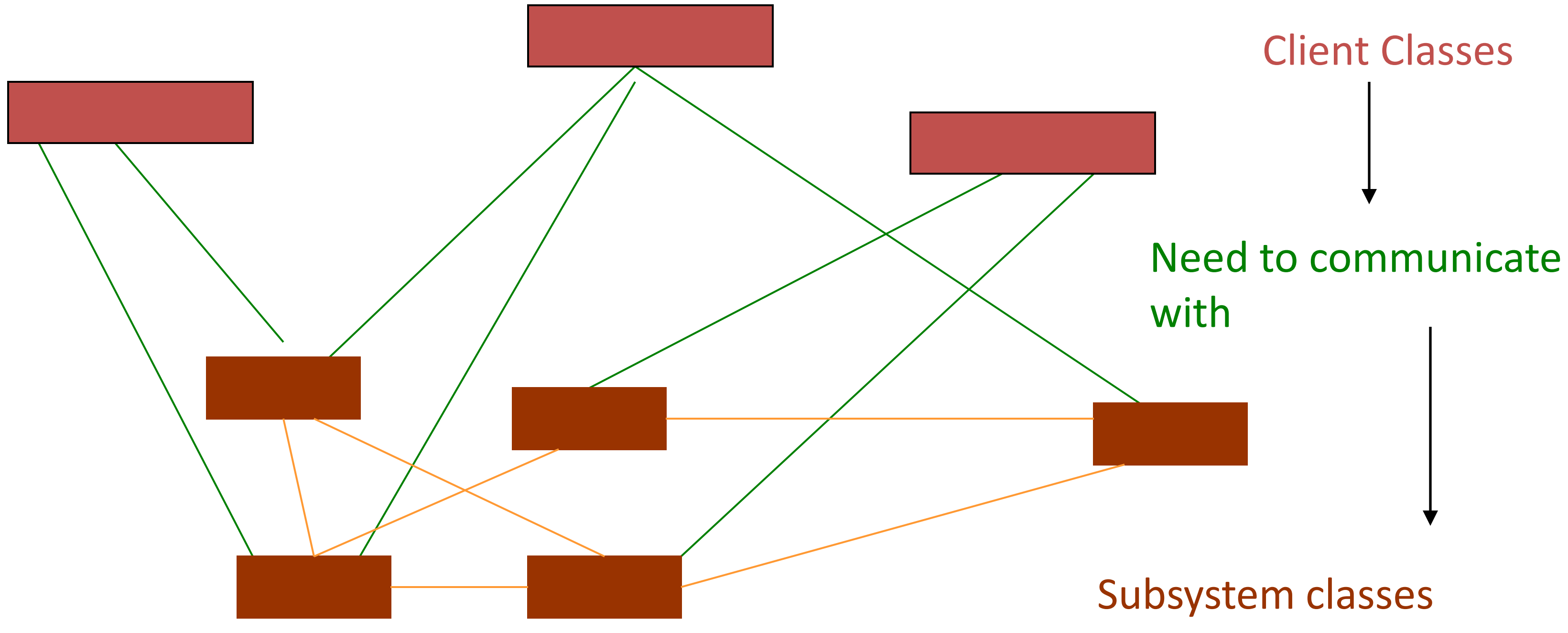
Observer Pattern: Consequences

- *Abstract coupling* between subject and observer. Subject has no knowledge of concrete observer classes. (What design principle is used?)

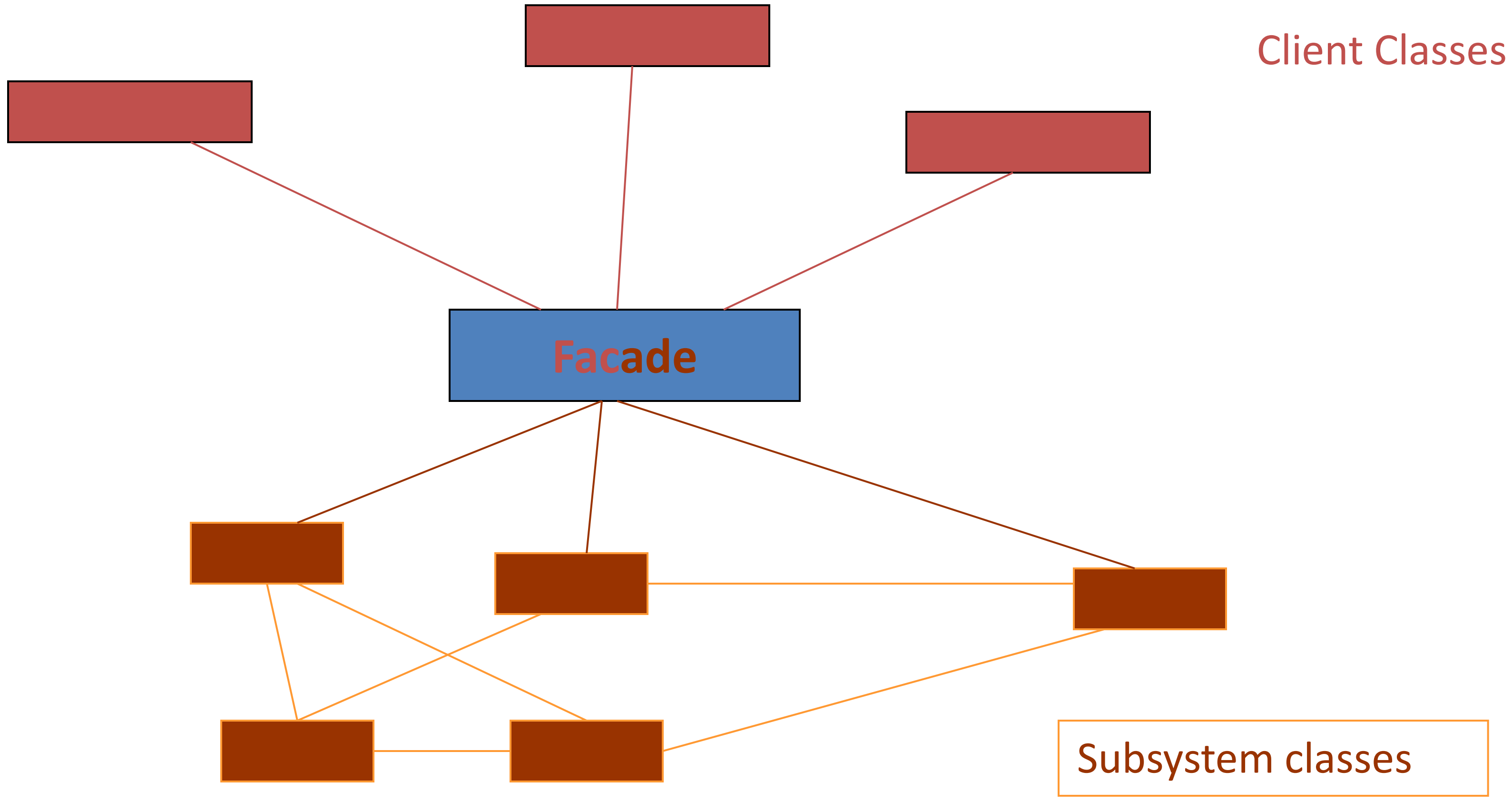
- *Support for broadcast communication*. A subject need not specify the receivers; all interested objects receive the notification.

- *Unexpected updates*: Observers need not be concerned about when then updates are to occur. They are not concerned about each other's presence. In some cases this may lead to unwanted updates.

Facade Pattern: Problem



Facade Pattern: Solution



Facade Pattern: Why and What?

- Subsystems often get complex as they evolve.
- Need to provide a simple interface to many, often small, classes. *But not necessarily to ALL classes of the subsystem.*
- Façade provides a simple default view good enough for most clients.
- Facade decouples a subsystem from its clients.
- A façade can be a single entry point to each subsystem level. This allows layering.

Facade Pattern: Participants and Communication

- Participants: Façade and subsystem classes
- Clients communicate with subsystem classes by sending requests to façade.
- Façade forwards requests to the appropriate subsystem classes.
- Clients do not have direct access to subsystem classes.

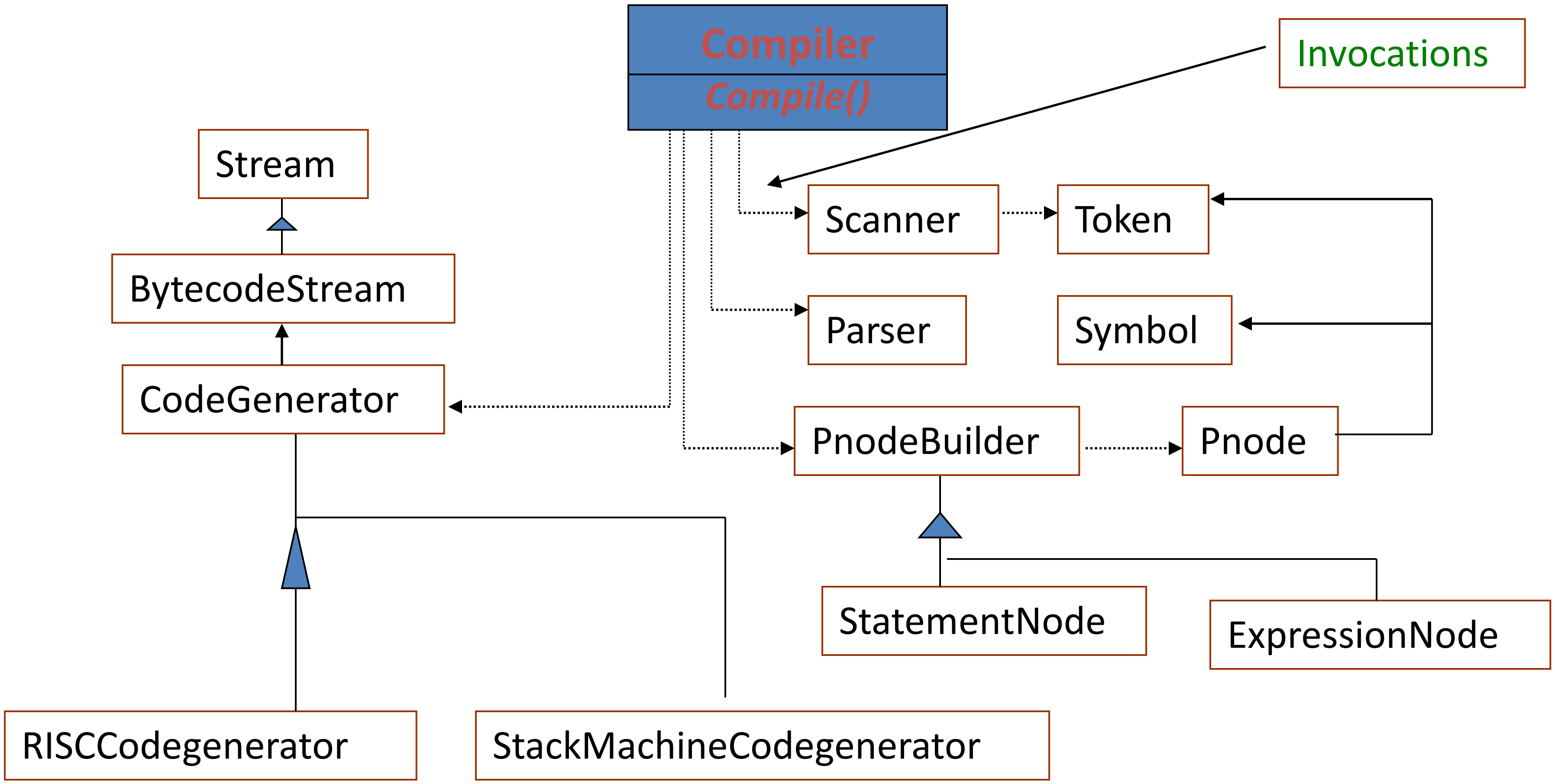
Facade Pattern: Benefits

- Shields clients from subsystem classes; reduces the number of objects that clients deal with.

- Promotes weak coupling between subsystem and its clients.

- Helps in layering the system. Helps eliminate circular dependencies.

Example: A compiler



Façade Pattern: Code [1]

```
class Scanner {      // Takes a stream of characters and produces a stream of tokens.

    public:
        Scanner (istream&);

        virtual Scanner();

        virtual Token& Scan();

    Private:

        istream& _inputStream;

    };
```

Façade Pattern: Code [2]

```
class parser {      // Builds a parse tree from tokens using the  PNodeBuilder.

    public:
        Parser ();

        virtual ~Parser()

        virtual void Parse (Scanner&, PNodeBuilder&);

 };
```

Façade Pattern: Code [3]

```
class Pnodebuilder {                    // Builds a parse tree incrementally. Parse tree
                                        // consists of Pnode objects.
  public:
        Pnodebuilder ();

        virtual Pnode* NewVariable (            // Node for a variable.
                Char* variableName
        ) const;

        virtual Pnode* NewAssignment (
                                                        // Node for an assignment.
                Pnode* variable, Pnode* expression
        ) const;
                                                // Similarly...more nodes.
 Private:
        Pnode* _node;
 };
```

Façade Pattern: Code [4]

```
class Pnode {     // An interface to manipulate the program node and its children.

  public:
     // Manipulate program node.

          virtual void GetSourcePosition (int& line, int& index);

     // Manipulate child node.

          virtual void Add (Pnode*);
          virtual void Remove  (Pnode*);
       // ....
          virtual void traverse (Codegenerator&);        // Traverse tree to generate code.

  protected:

       PNode();

  };
```

Façade Pattern: Code [5]

```
class CodeGenerator {                    // Generate bytecode.

  public:
     // Manipulate program node.

         virtual void Visit (StatementNode*);
         virtual void Visit (ExpressionNode*);
        // ....
 Protected:
        CodeGenerator (BytecodeStream&);

        BytecodeStream& _output;

 };
```

Façade Pattern: Code [6]

```
void ExpressionNode::Traverse (CodeGenerator& cg) {

    cg.Visit (this);

      ListIterator<Pnode*> i(_children);
      For (i.First(); !i.IsDone(); i.Next();{
          i.CurrentItem()→Traverse(cg);
        };
    };
```

Façade Pattern: Code [7]

```
class Compiler {              // Façade. Offers a simple interface to compile and
                              // Generate code.
  public:
        Compiler();
        virtual void Compile (istream&, BytecodeStream&);
  }
void Compiler:: Compile (istream& input, BytecodeStream& output) {
        Scanner scanner (input);
        PnodeBuilder builder;
        Parser parser;
        parser.Parse (scanner, builder);
        RISCCodeGenerator generator (output);
        Pnode* parseTree = builder.GetRootNode();
        parseTree→Traverse (generator);
}
```

Could also take a CodeGenerator Parameter for increased generality.

Facade Pattern: Another Example from POS [1]

- Assume that rules are desired to invalidate an action:
  - Suppose that when a new Sale is created, it will be paid by a gift certificate
  - Only one item can be purchased using a gift certificate.
  - Hence, subsequent enterItem operations must be invalidated in some cases. (Which ones?)

How does a designer factor out the handling of such rules?

Facade Pattern: Another Example [2]

- Define a "rule engine" subsystem (e.g. POSRuleEngineFacade).
- It evaluates a set of rules against an operation and indicates if the rule has invalidated an operation.
- Calls to this façade are placed near the start of the methods that need to be validated.
  - Example: Invoke the façade to check if a new salesLineItem created by makeLineItem is valid or not. (See page 370 of Larman.)