Lecture.
Design Patterns

Dr. Miryung Kim

Recap

- Design Principle for Ease of Change: Information Hiding
  - *Hide design decisions that are likely to change*
  - *Reveal decisions that are unlikely to change as interfaces*
- Class activity---Critique MortgageCalculator
- Review questions about the application of the IH principle

THE UNIVERSITY OF TEXAS AT AUSTIN

SEAL Software Evolution & Analysis Laboratory

Today's Agenda

- Introduction of design patterns
- AbstractFactory and FactoryMethod

Announcement

- Topics for today and next two lectures
  - Why Design Patterns?
  - *Abstract Factory, Factory Method, Singleton*
  - *Adapter, Flyweight, Bridge*
  - *Observer, Mediator, Strategy, Visitor*
- Read relevant GoF design patterns.
- Head First Design Patterns

Reprise: What is "Engineering"?

Definitions abound. They have in common:

Creating cost-effective solutions ...

... to practical problems ...

... by applying scientific knowledge ...

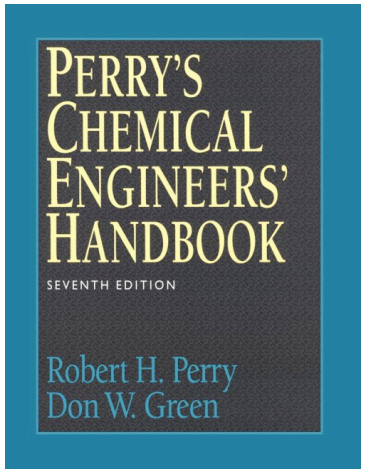
... building things ...

... in the service of mankind

Engineering enables ordinary people to do things that formerly required virtuosos

Slide from Mary Shaw @ CMU

Example handbook



- Contents
  - Chemical and physical property data
  - Fundamentals (e.g. thermodynamics)
  - Processes (the bulk of the book)
    - heat transfer operations
    - distillation
    - kinetics
    - liquid–liquid
    - liquid–solid
    - etc.
  - Materials of construction
  - Waste management

Slide from Rob DeLine @ Microsoft Research

Other Precedents

- Polya's How to Solve It
  - Catalogs techniques for solving mathematical (geometry) problems
  - Two categories: problems to prove, problems to find/construct
- Christopher Alexander's books, e.g. A Pattern Language
  - Saw building architecture/urban design as recurring patterns
  - Gives 253 patterns as: name; example; context; problem; solution
- **Pattern languages as engineering handbooks**
  - Hype aside, it's about recording known solutions to problems
  - Pattern languages exist for many problems, but we'll look at design
  - Best known: Gamma, Helm, Johnson, Vlissides ("Gang of four")
    Design Patterns: Elements of reusable object-oriented software
  - Notice the subtitle: here, design is about objects and their interactions

THE UNIVERSITY OF TEXAS AT AUSTIN

Slide from Rob DeLine @ Microsoft Research

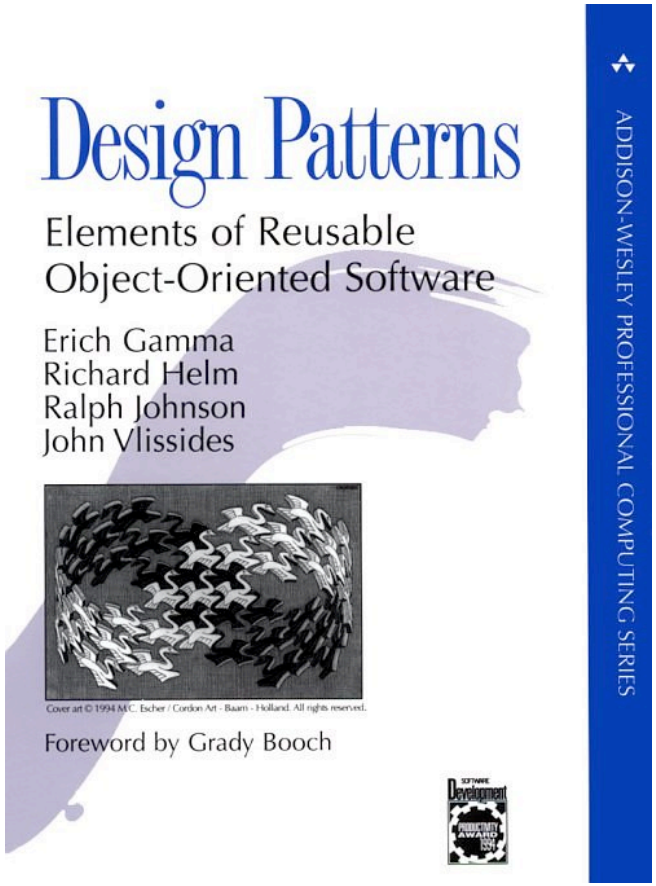
SEAL Software Evolution & Analysis Laboratory

Why do we need Design Patterns?

THE UNIVERSITY OF TEXAS AT AUSTIN

SEAL Software Evolution & Analysis Laboratory

Design Patterns



Why do we need Design Patterns?

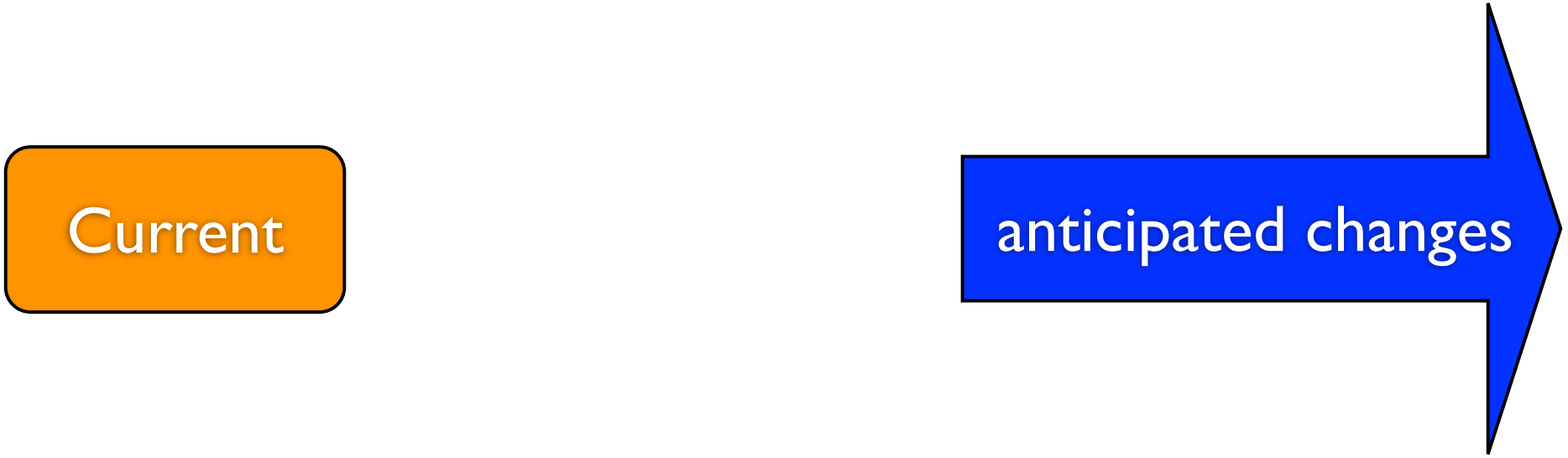
1. Abstract design experience => a reusable base of experience
2. Provide a common vocabulary for discussing design
3. Reduce system complexity by naming abstractions => reduce the learning time for a class library / program comprehension

THE UNIVERSITY OF TEXAS AT AUSTIN

SEAL Software Evolution & Analysis Laboratory

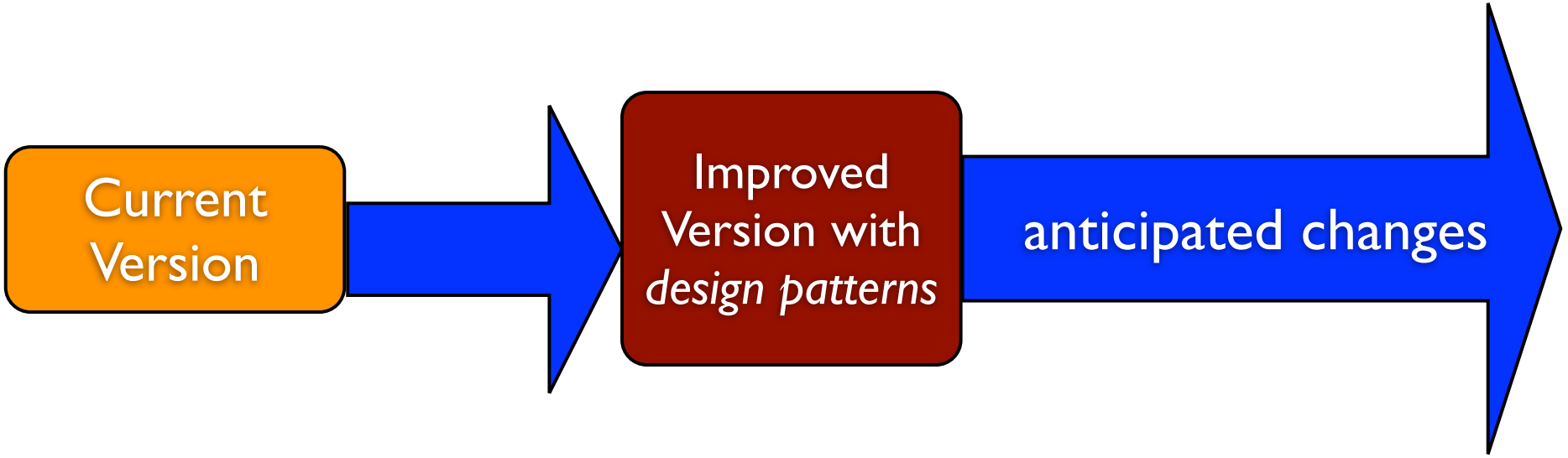
Why do we need Design Patterns?

4. Provide a target for the reorganization or refactoring of class hierarchies



Why do we need Design Patterns?

4. Provide a target for the reorganization or refactoring of class hierarchies



THE UNIVERSITY OF TEXAS AT AUSTIN

SEAL Software Evolution & Analysis Laboratory

Which aspects of design does *Gang Of Four* discuss?

- a class or object collaboration and its structure

Design Patterns

Problems / Goals

Solutions

What types of changes are easier to implement due to this design pattern

Case studies

THE UNIVERSITY OF TEXAS AT AUSTIN

SEAL Software Evolution & Analysis Laboratory

Example: Abstract Factory

Problem / Goal
: Having an explicit dependencies on concrete product classes makes it difficult to change product types or add a new product type.

Example: Abstract Factory

Typical OOP program hard-codes type choices

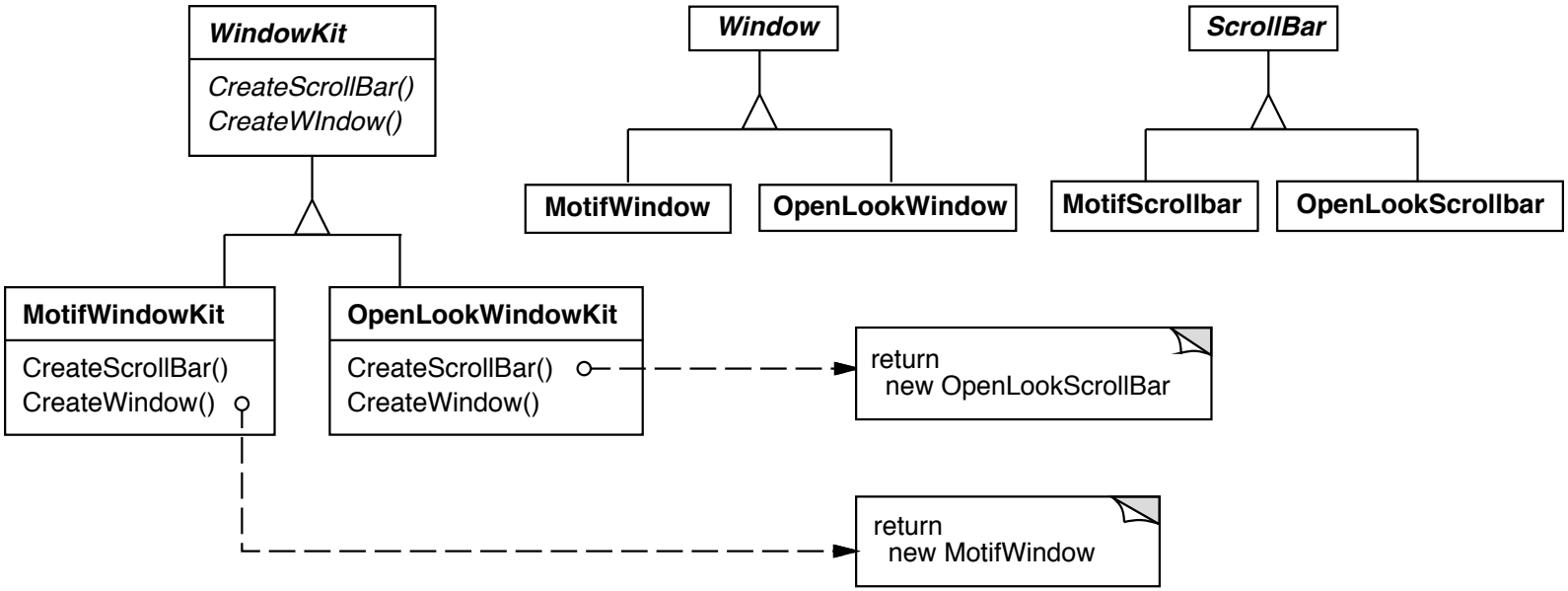
```
void AppInit () {
#if Motif
  Window w = new MotifWindow(...);
  ScrollBar b = new MotifScrollBar(...);
#else if OpenLook
  Window w = new OpenLookWindow(...);
  ScrollBar b = new
  OpenLookScrollBar(...);
#endif
  w.Add(b);
}
```

We want to easily change the app's "look and feel", which means calling different constructors.

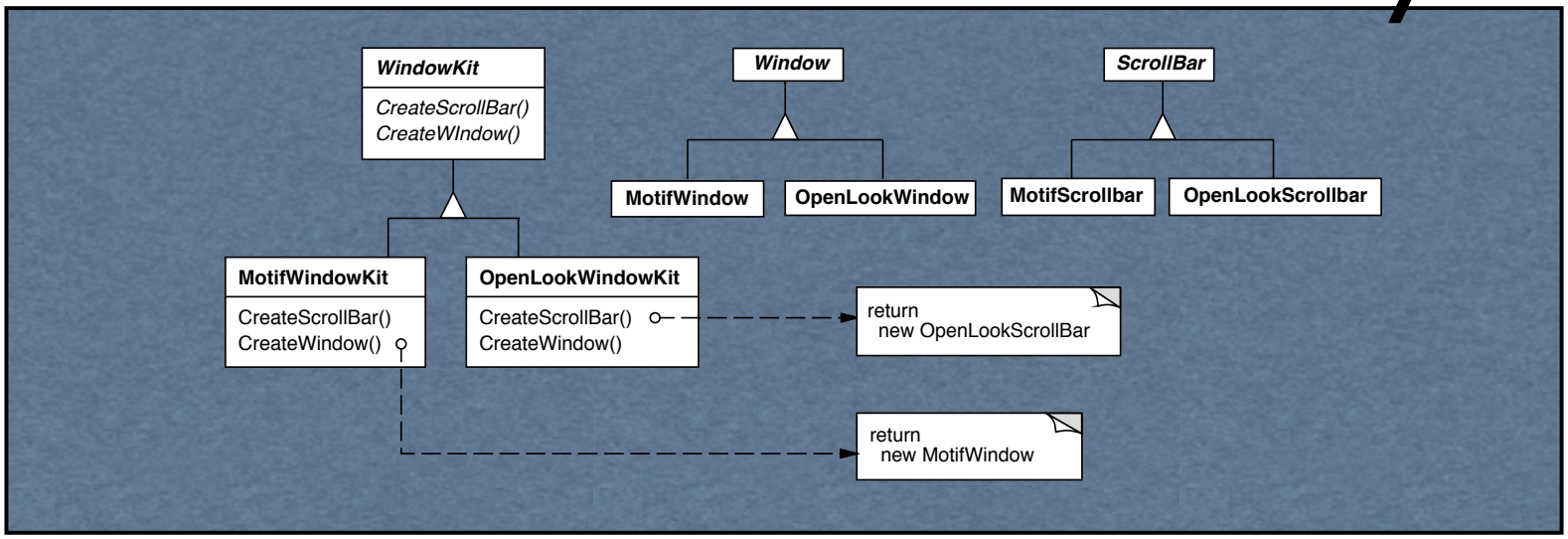
Solution: Abstract Factory

Solution
: Wrap the constructors in factory methods

Solution: Abstract Factory



Solution: Abstract Factory



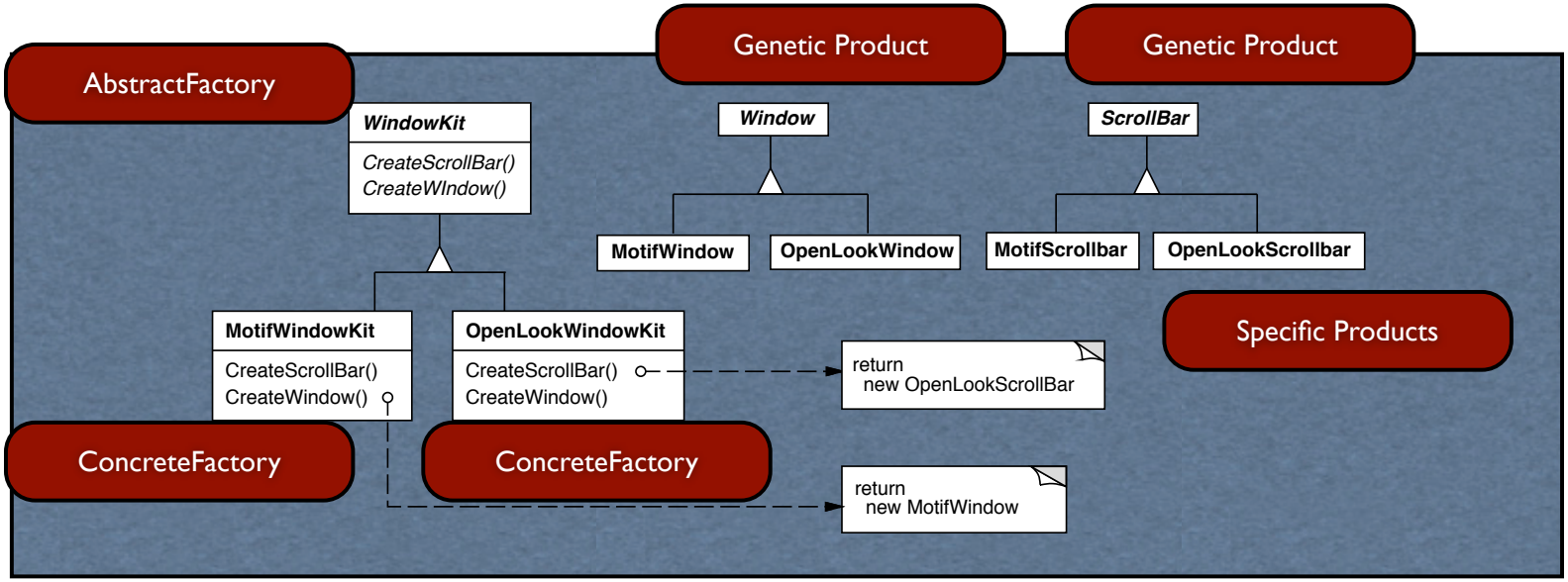
Client Code

```
WindowKit kit = new MotifWindowKit();
kit.appInit();
```

```
class WindowKit {
  WindowKit ();
  Window CreateWindow (...);
  ScrollBar CreateScrollBar (...);
```

```
void appInit () {
  Window w = CreateWindow(...);
  ScrollBar b = CreateScrollBar(...);
  w.Add(b);
}
```

Participants



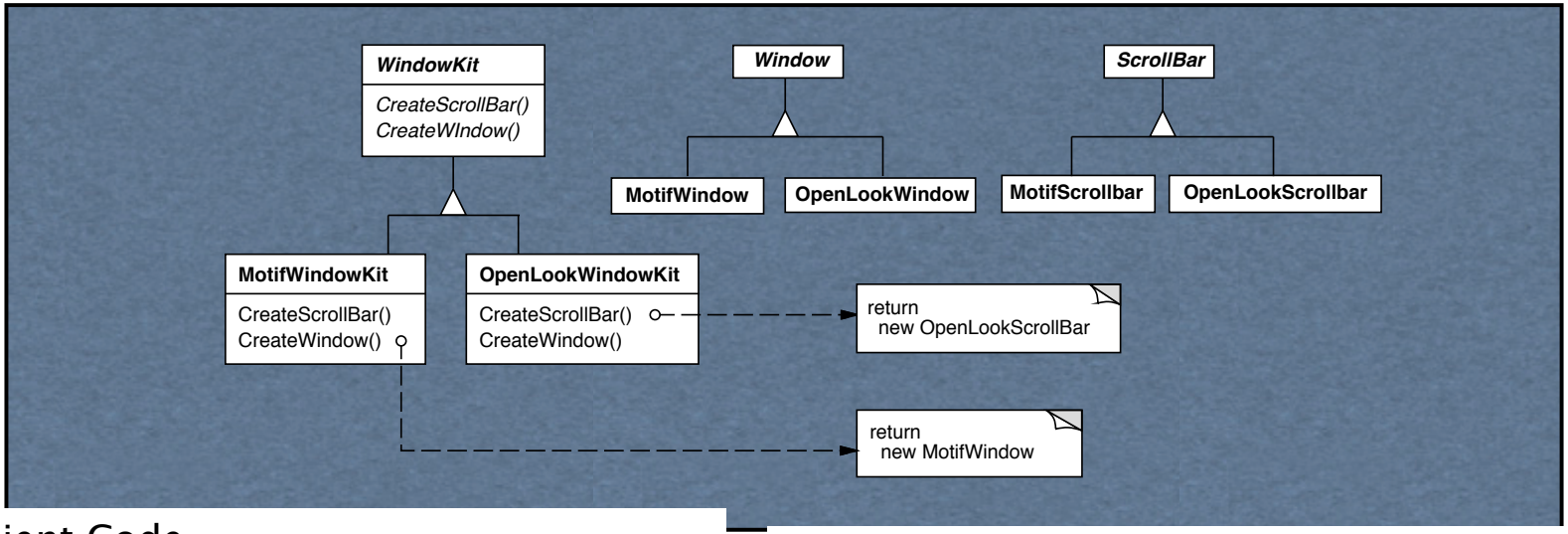
What types of changes can you anticipate?

- What happens if we have multiple types of windows?
- What happens if we need different types of windows that take different arguments?
- What happens if we want to define a window as combination of window, scroll bar and button

What types of changes can you anticipate?

- Adding a different look and feel such as MacWindowKit
- Adding a new type of object such as a button as a part of WindowKit

How about adding a different look and feel such as MacWindowKit?



Client Code

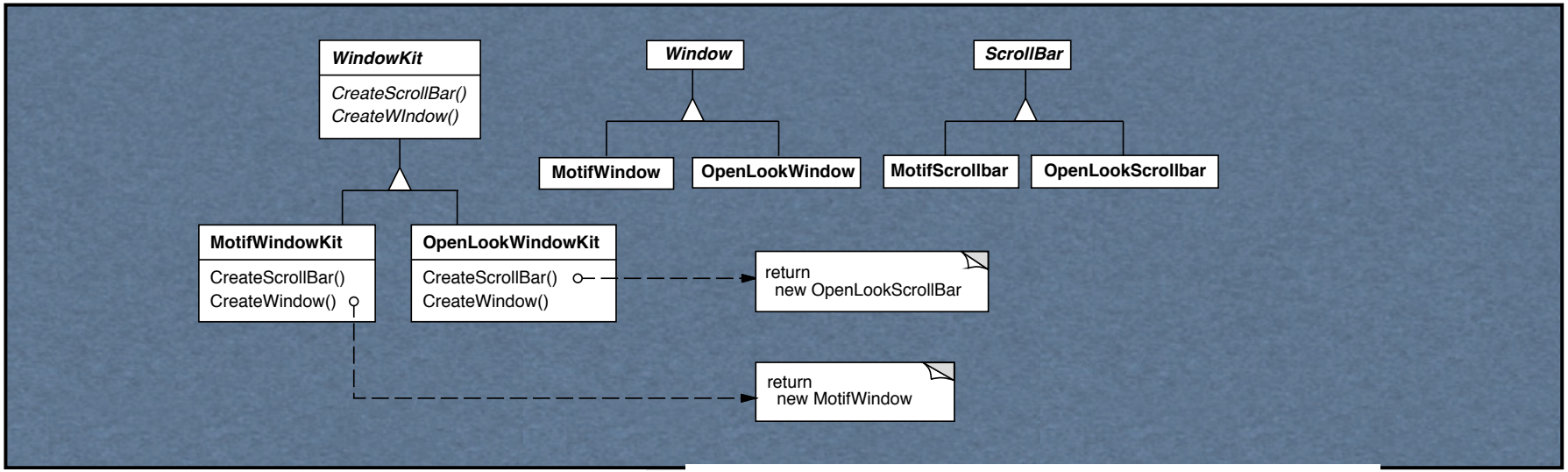
```
WindowKit kit = new MotifWindowKit();
kit.appInit();
```

```
class WindowKit {
  WindowKit ();
  Window CreateWindow (...);
  ScrollBar CreateScrollBar (...);
void appInit () {
  Window w = CreateWindow(...);
  ScrollBar b = CreateScrollBar(...);
  w.Add(b);
}
```

THE UNIVERSITY OF TEXAS AT AUSTIN

SEAL Software Evolution & Analysis Laboratory

How about adding a new type of object such as button?



Client Code

```
WindowKit kit = new MotifWindowKit();
kit.appInit();
```

```
class WindowKit {
   WindowKit ();
   Window CreateWindow (...);
   ScrollBar CreateScrollBar (...);
void appInit () {
   Window w = CreateWindow(...);
   ScrollBar b = CreateScrollBar(...);
   w.Add(b);
}
```

THE UNIVERSITY OF TEXAS AT AUSTIN

SEAL Software Evolution & Analysis Laboratory

Factory Method

- Problem: A framework needs to standardize the architectural model for a range of applications but allow individual applications to define their own domain objects and instantiations.

Factory Method

- Intent: Define an interface for creating an object but let subclasses decide which class to instantiate. Factory Method lets a class to defer instantiation to subclasses.
- Participants: Product, Concrete Product, Creator, Concrete Creator

THE UNIVERSITY OF TEXAS AT AUSTIN

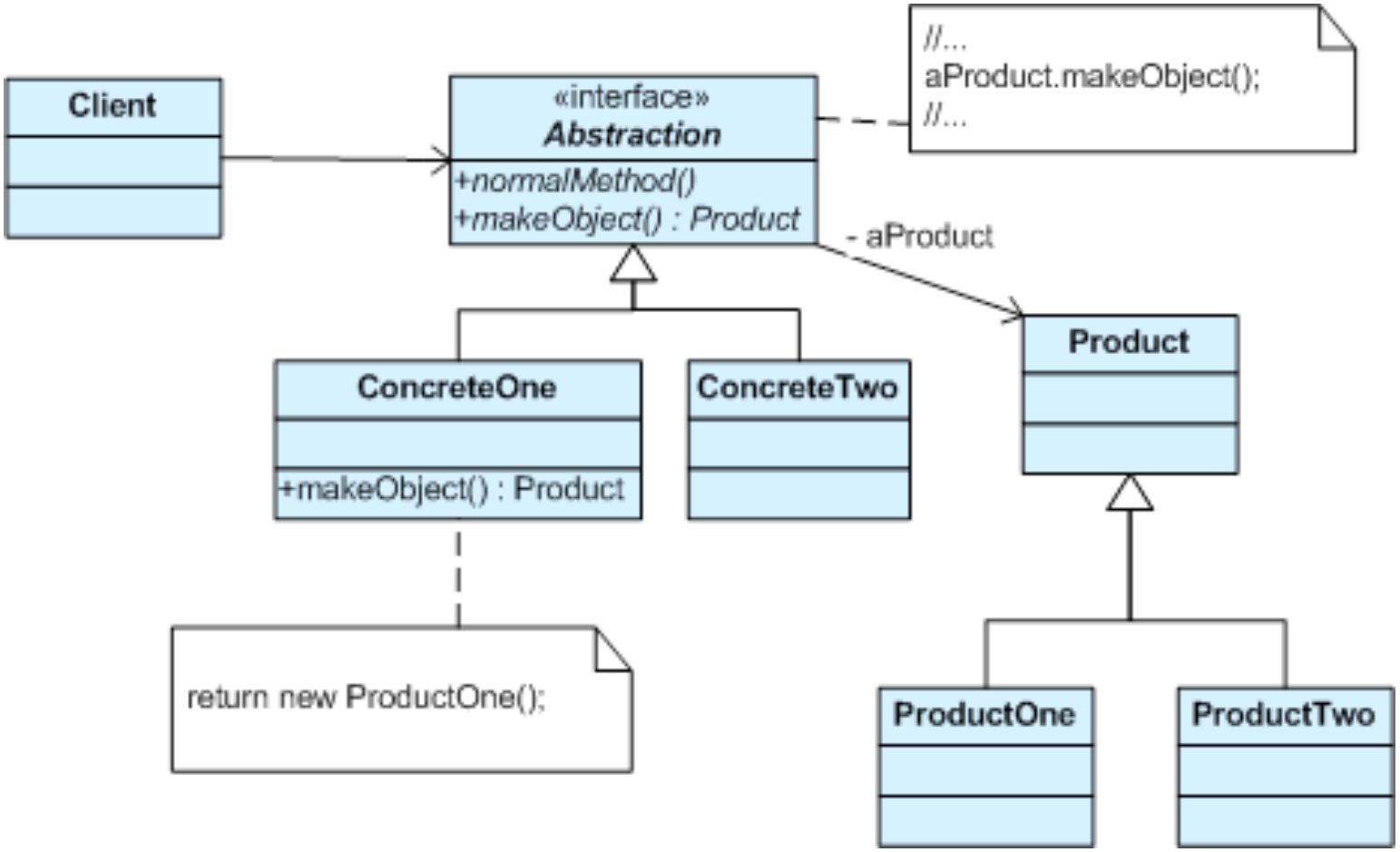
SEAL Software Evolution & Analysis Laboratory

Factory Method

Factory Method is to **creating objects as Template Method** is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

Factory Method makes a design more **customizable** and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.

Factory Method



Factory Method

```
public interface ImageReader {
    public DecodedImage getDecodedImage();
}

public class GifReader implements ImageReader {
    public GifReader( InputStream in ) {
        // check that it's a gif, throw exception if it's not, then if it
is decode it.
    }

    public DecodedImage getDecodedImage() {
       return decodedImage;
    }
}

public class JpegReader implements ImageReader {
    //...
}
```

Recap

- Design patterns are canonical, well known design solutions to recurring / reusable problems.

Review Question 1

- What are design problems associated with the following code?

```
Duck duck;
  if (picnic) {
    duck = new MallardDuck();
  } else if (hunting) {
    duck = new DecoyDuck();
  } else if (inBathTub) {
    duck = new RubberDuck();
  }
```

Review Question 2

- Refactor the following code by creating SimplePizzaFactory

```
Pizza orderPizza(String type) {
   Pizza pizza;
   if (type.equals("cheese")) {
      pizza = new CheesePizza();

   }else if (type.equals("greek")) {
      pizza = new GreekPizza() ;

   }else if (type.equals("pepperoni")) {
      pizza = new PepperoniPizza();
   }
   pizza.bake();
   pizza.box();
   return pizza;
}
```

Review Question 3

- Refactor the following code to be able to order pizzas from NewYorkPizzaFactory and ChicagoPizzaFactory.

```
public class PizzaStore {
  SimplePizzaFactory factory;
  public PizzaStore(SimplePizzaFactory factory) {
    this.factory =factory;
  }

  Pizza orderPizza(String type) {
    Pizza pizza=factory.createPizza(type);
    pizza.bake();
    pizza.box();
    return pizza;
  }
}
```

Review Question 4.

- What are the design problems with the following code?

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("Veggie");

ChicagoPizzaFactory chicagoFactory = new
ChicagoPizzaFactory();
  PizzaStore chicagoStore = new
PizzaStore(chicagoFactory);
chicagoStore.order("Veggie");
```

Review Question 5.

- Refactor the following code using AbstractFactory

```
package headfirst.factory.pizzafm;

public class DependentPizzaStore {

   public Pizza createPizza(String style, String type) {
      Pizza pizza = null;
      if (style.equals("NY")) {
         if (type.equals("cheese")) {
            pizza = new NYStyleCheesePizza();
         } else if (type.equals("veggie")) {
            pizza = new NYStyleVeggiePizza();
         } else if (type.equals("clam")) {
            pizza = new NYStyleClamPizza();
         } else if (type.equals("pepperoni")) {
            pizza = new NYStylePepperoniPizza();
         }
      } ....
```

```
...
    } else if (style.equals("Chicago")) {
      if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
      } else if (type.equals("veggie")) {
        pizza = new ChicagoStyleVeggiePizza();
      } else if (type.equals("clam")) {
        pizza = new ChicagoStyleClamPizza();
      } else if (type.equals("pepperoni")) {
        pizza = new ChicagoStylePepperoniPizza();
      }
    } else {
      System.out.println("Error: invalid type of pizza");
      return null;
    }
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

Today's Agenda

- Continuation with design patterns & associated class activities
  - singleton
  - adaptor
  - flyweight
  - bridge

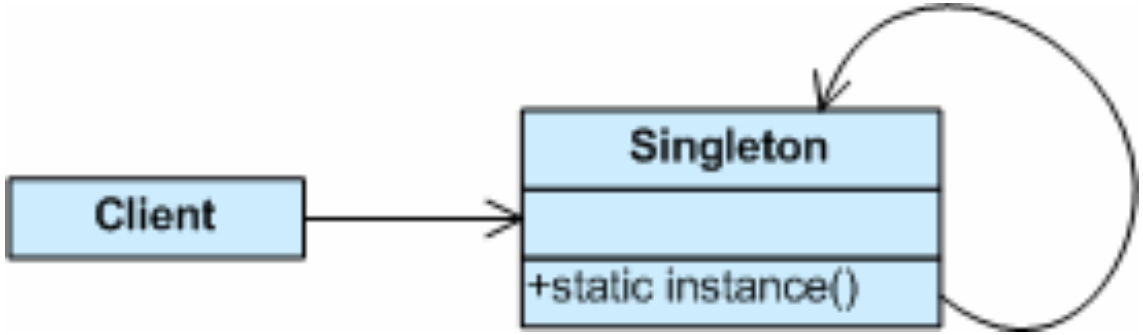
Singleton

- Problem: Application needs one and only one instance of an object. Additionally, lazy instantiation and global access are necessary.

Singleton

- Intent: Ensure a class has only one instance, and provide a global point of access to it.
- Encapsulated "just-in-time realization" or "initialization on first use"

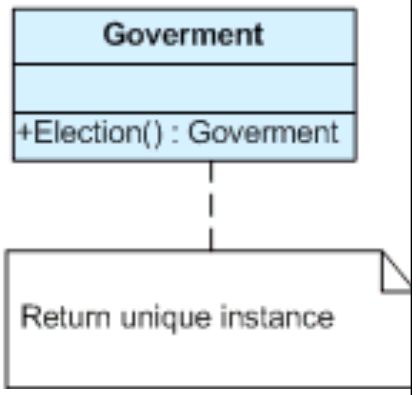
Singleton



Singleton Example

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.

Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.



```
public class Singleton {
  // Private constructor prevents instantiation
from other classes
  private Singleton() {}

  /**
   * SingletonHolder is loaded on the first
execution of Singleton.getInstance()
   * or the first access to
SingletonHolder.INSTANCE, not before.
   */
  private static class SingletonHolder {
    private static final Singleton INSTANCE =
new Singleton();
  }

  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

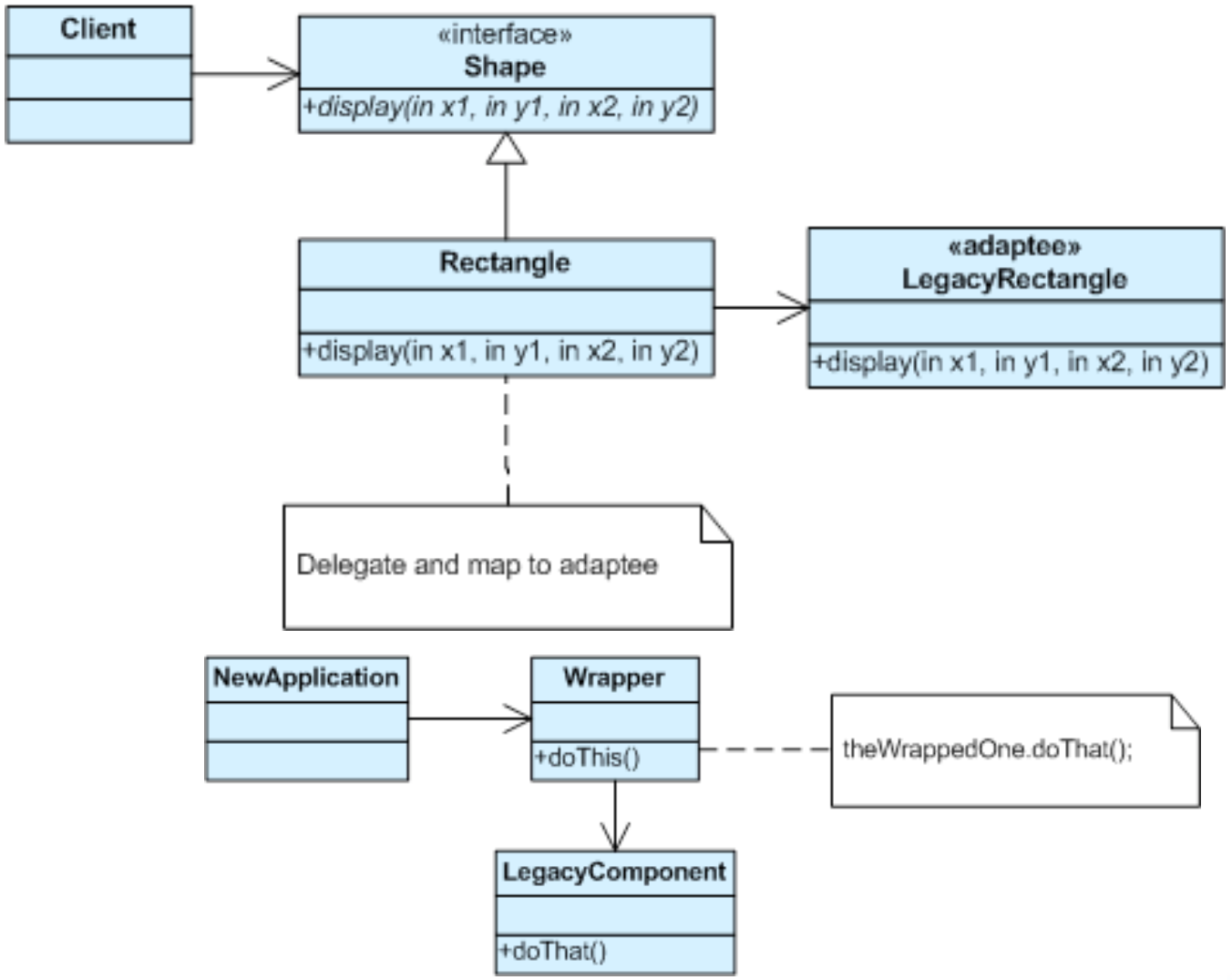
Adapter

- Problem
  - An "off-the-shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system being developed.

Adapter

- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface.
  - Impedance match an old component to a new system

Adaptor



Flyweight

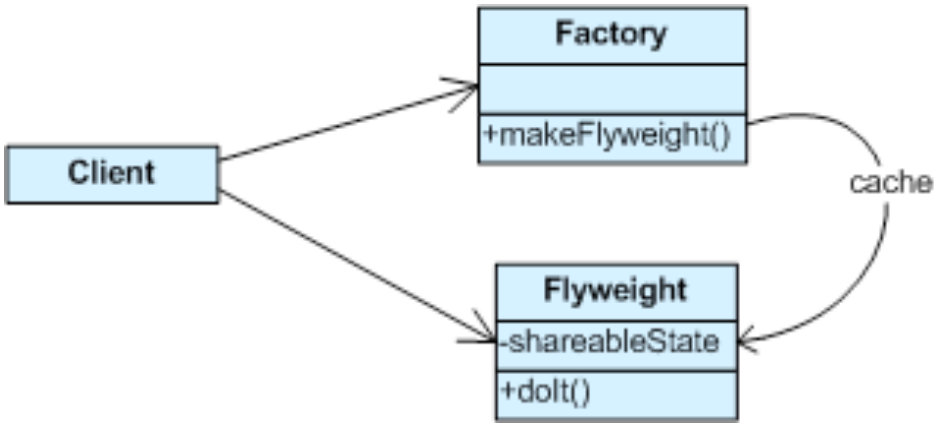
- Problem
  - Designing objects down to the lowest levels of system 'granularity' provides optimal flexibility but can be unacceptably expensive in terms of performance and memory usage

Flyweight

Intent

- Use sharing to support large numbers of fine-grained objects efficiently.
- The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets

Flyweight



Bridge

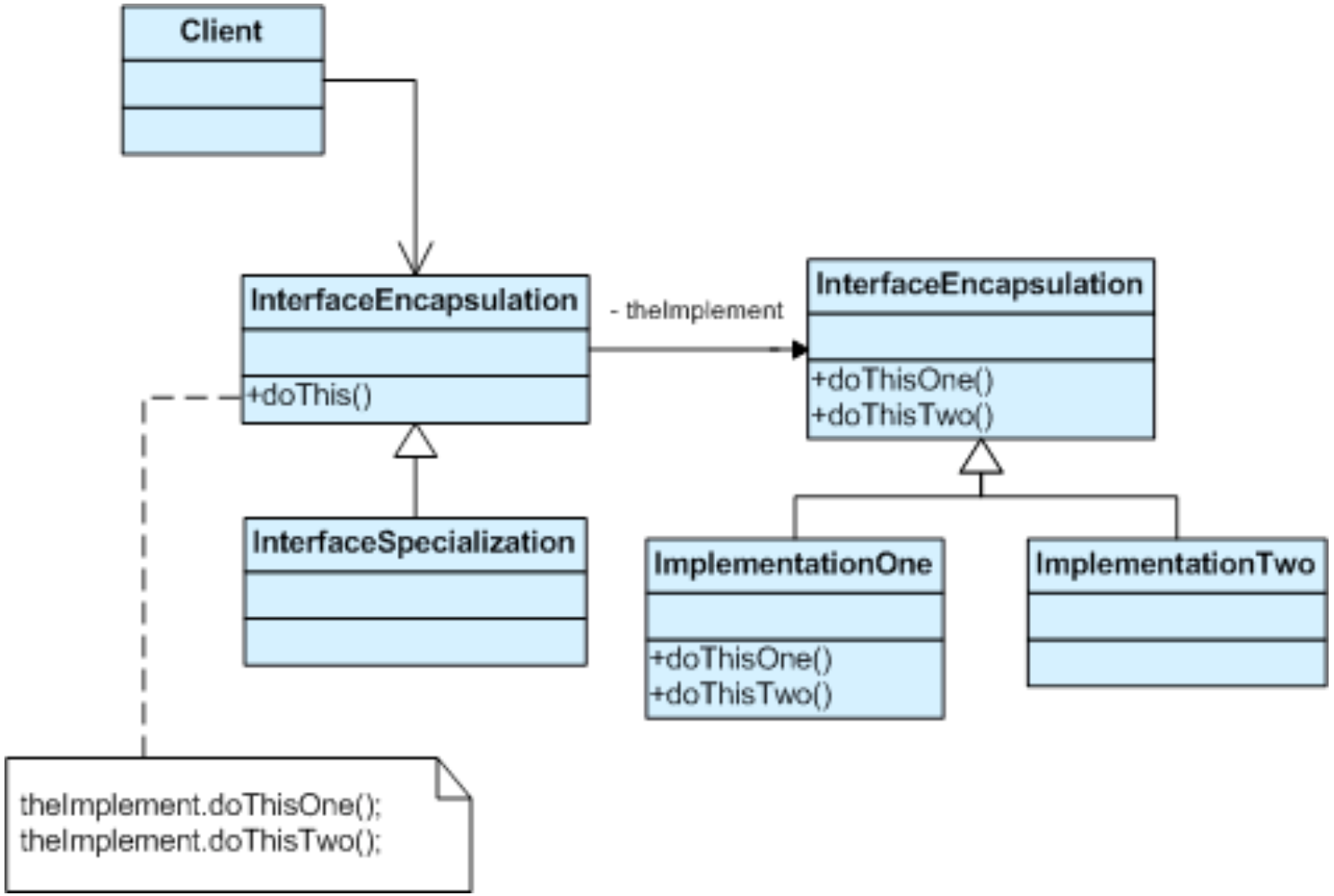
- Problem
  - "Hardening of the software arteries" has occurred by using subclasses of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation

Bridge

Intent

- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.

Bridge



Recap

- Please review Singleton, Adaptor, Flyweight, and Bridge Patterns.
- At home, think about other example applications/ context whether these patterns might be applicable to.
- At home, identify change scenarios under which these design patterns may not be effective.

Review Question 1

- Refactor the following code to implement a singleton.

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }

}
```

Review Question 2

- Describe design issues associated with the following code.

```
class LegacyLine
{
    public void draw(int x1, int y1, int x2, int y2)
    {
        System.out.println("line from (" + x1 + ',' + y1 + ") to (" + x2 + ','
          + y2 + ')');
    }
}
class LegacyRectangle
{
    public void draw(int x, int y, int w, int h)
    {
        System.out.println("rectangle at (" + x + ',' + y + ") with width " + w
          + " and height " + h);
    }
}
```

```
public class AdapterDemo
{
    public static void main(String[] args)
    {
        Object[] shapes =
        {
            new LegacyLine(), new LegacyRectangle()
        };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
          if (shapes[i].getClass().getName().equals("LegacyLine"))
            ((LegacyLine)shapes[i]).draw(x1, y1, x2, y2);
          else if (shapes[i].getClass().getName().equals("LegacyRectangle"))
            ((LegacyRectangle)shapes[i]).draw(Math.min(x1, x2), Math.min(y1, y2)
              , Math.abs(x2 - x1), Math.abs(y2 - y1));
    }
}
```

Review Question 3

- The following code shows an improved design of the previous code using the Adapter design patterns. What kinds of improvement have you observed?

```
interface Shape
{
  void draw(int x1, int y1, int x2, int y2);
}

class Line implements Shape
{
    private LegacyLine adaptee = new LegacyLine();
    public void draw(int x1, int y1, int x2, int y2)
    {
        adaptee.draw(x1, y1, x2, y2);
    }
}

class Rectangle implements Shape
{
    private LegacyRectangle adaptee = new LegacyRectangle();
    public void draw(int x1, int y1, int x2, int y2)
    {
        adaptee.draw(Math.min(x1, x2), Math.min(y1, y2), Math.abs(x2 - x1),
          Math.abs(y2 - y1));
    }
}
```

```
public class AdapterDemo
{
    public static void main(String[] args)
    {
        Shape[] shapes =
        {
            new Line(), new Rectangle()
        };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
          shapes[i].draw(x1, y1, x2, y2);
    }
}
```

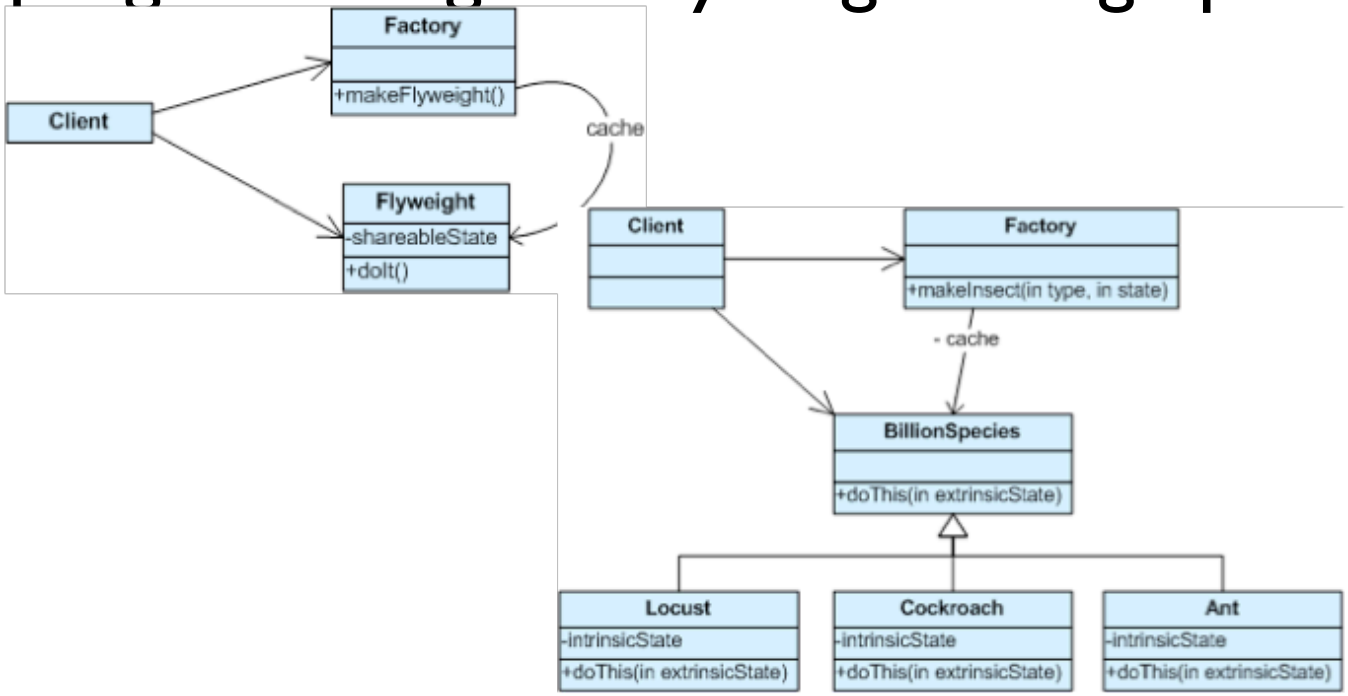
Review Question 4

- Describe performance issues associated with the following code

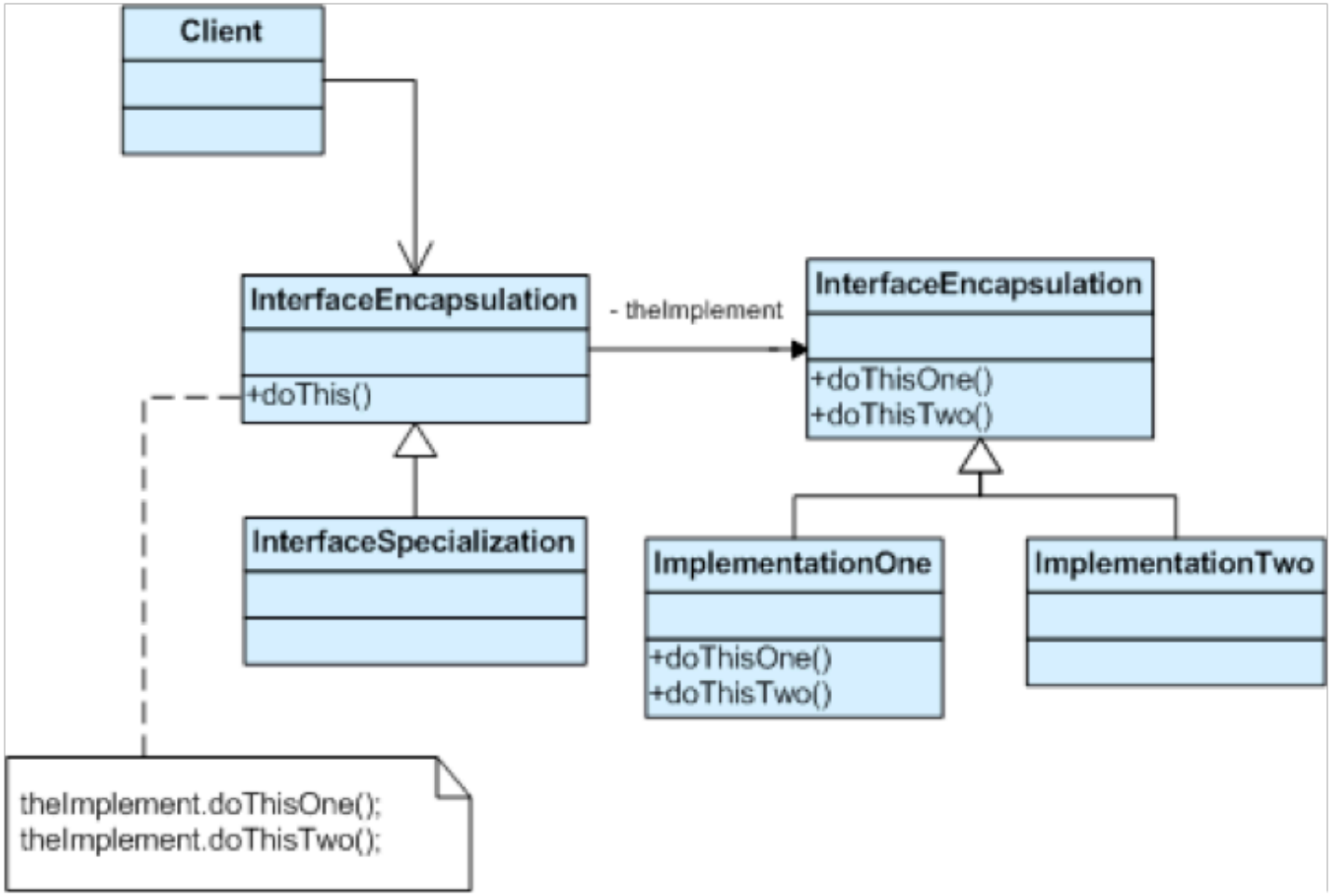
```
class Gazillion {
   private static int num = 0;
   private int        row, col;
   public Gazillion( int maxPerRow ) {
      row = num / maxPerRow;
      col = num % maxPerRow;
      num++;
   }
   void report() {
      System.out.print( " " + row + col );
} }

public class FlyweightDemo {
   public static final int ROWS = 6, COLS = 10;

   public static void main( String[] args ) {
      Gazillion[][] matrix = new Gazillion[ROWS][COLS];
      for (int i=0; i < ROWS; i++)
         for (int j=0; j < COLS; j++)
            matrix[i][j] = new Gazillion( COLS );
      for (int i=0; i < ROWS; i++) {
         for (int j=0; j < COLS; j++)
            matrix[i][j].report();
         System.out.println();
```

Review Question 5

- Produce a new version of the above program using the Flyweight design pattern.



Review Question 6.



- The above UML diagram describes the participants of the Bridge Design Pattern. Bridge emphasizes identifying and decoupling "interface" abstraction from "implementation" abstraction. For the following code fragments shown in the following page, identify class names that correspond to the participants of the Bridge patterns shown in the above UML diagram, (e.g., InterfaceEncapsulation, InterfaceSpecialization, etc.).

```
class Stack {
protected StackImp imp;
  public Stack( String s ) {
    if (s.equals("java")) {
      imp = new StackJava();
    }
    else {
      imp = new StackMine();
    }
  }
  public Stack() {
    this( "java" );
  }
  public void push( int in ) {
    imp.push( new Integer(in) );
  }
  public int pop() {
    return ((Integer)imp.pop()).intValue();
  }
  public boolean isEmpty() {
    return imp.empty();
  }
}
```

```
// Embellish the interface class with derived classes if desired
class StackHanoi extends Stack {
  private int totalRejected = 0;
  public StackHanoi()             { super( "java" ); }
  public StackHanoi( String s ) { super( s ); }
  public int reportRejected()    { return totalRejected; }
  public void push( int in ) {
    if ( ! imp.empty()  &&  in > ((Integer)imp.peek()).intValue())
      totalRejected++;
    else
      imp.push( new Integer(in) );
  }
}

// Create an implementation/body base class
interface StackImp {
  Object push( Object o );
  Object peek();
  boolean empty();
  Object  pop();
}
```

```
class StackJava extends java.util.Stack implements StackImp { }

// Derive the separate implementations from the common abstraction
class StackMine implements StackImp {
  private Object[] items = new Object[20];
  private int total = -1;

  public Object push( Object o ) {
    return items[++total] = o;
  }
  public Object peek() {
    return items[total];
  }
  public Object pop() {
    return items[total--];
  }
  public boolean empty() {
    return total == -1;
  }
}
```

```
class BridgeDemo {
  public static void main(String[] args) {
    Stack[] stacks = { new Stack("java"), new Stack("mine"),
                       new StackHanoi("java"), new
StackHanoi("mine") };
    for (int i=0, num; i < 20; i++) {
      num = (int) (Math.random() * 1000) % 40;
      for (int j=0; j < stacks.length; j++)
stacks[j].push( num );
    }
    for (int i=0, num; i < stacks.length; i++) {
      while ( ! stacks[i].isEmpty()) {
        System.out.print( stacks[i].pop() + "  " );
      }
      System.out.println();
    }
    System.out.println( "total rejected is " +
((StackHanoi)stacks[3]).reportRejected() );
  }
}
```

Today's Agenda

- Continuation with design patterns & associated class activities
  - observer
  - mediator
  - strategy
  - visitor

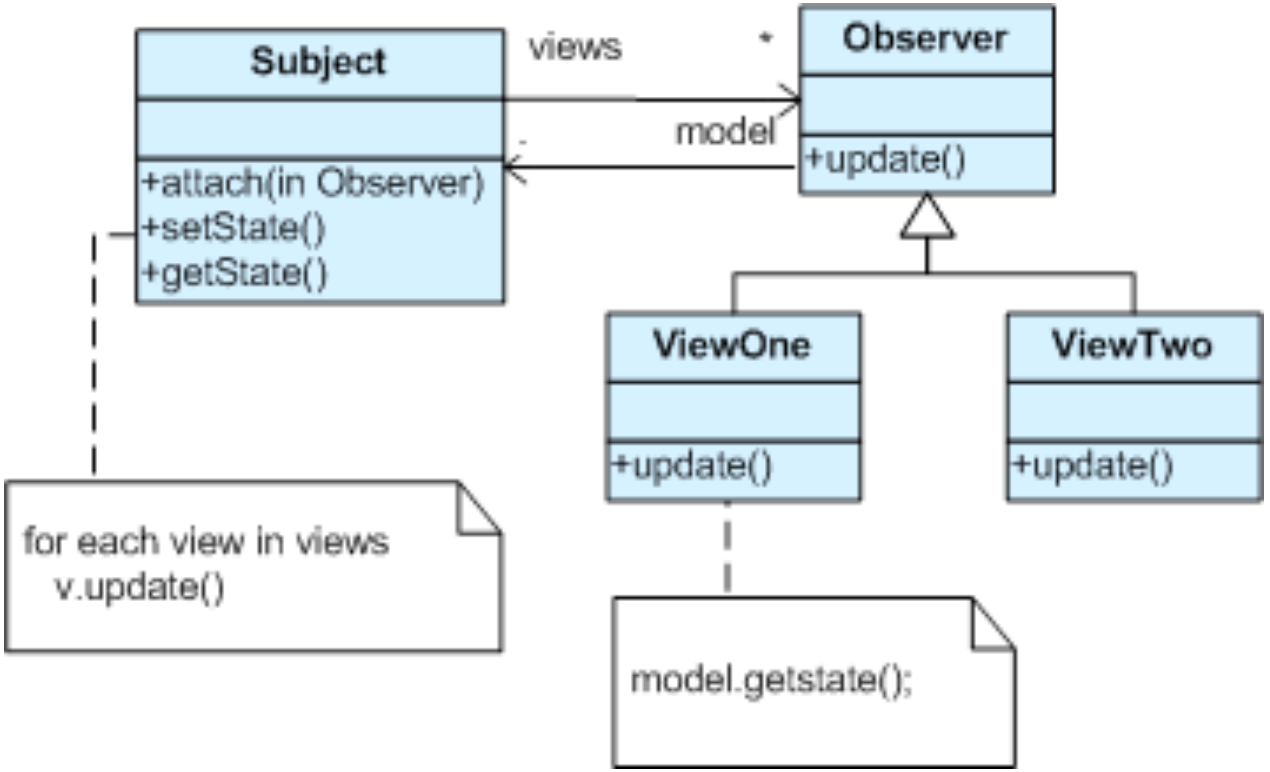
Observer

- Problem
  - A large monolithic design does not scale well as new monitoring requirements are levied

Observer

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

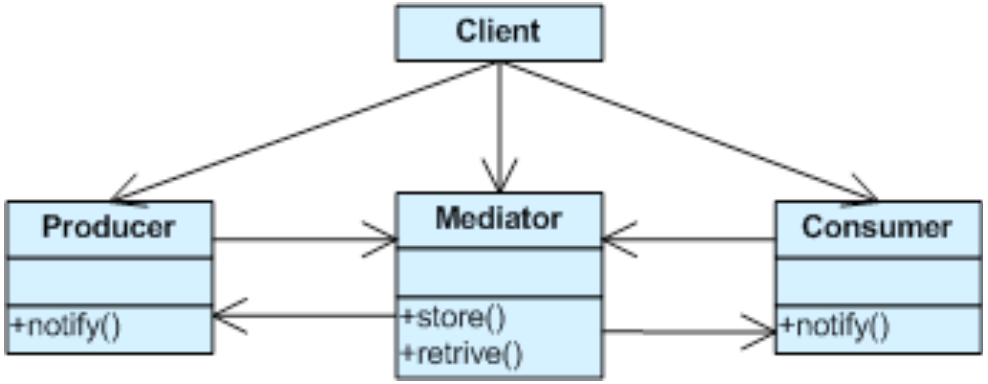
Observer

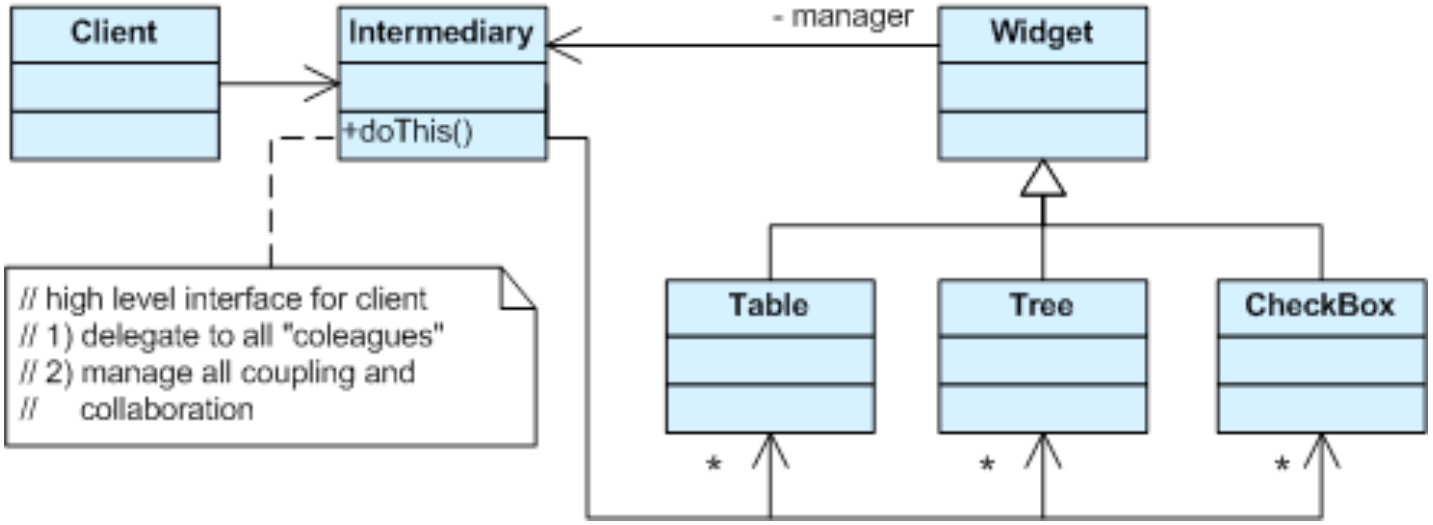


Mediator

- Intent: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interaction independently

Mediator

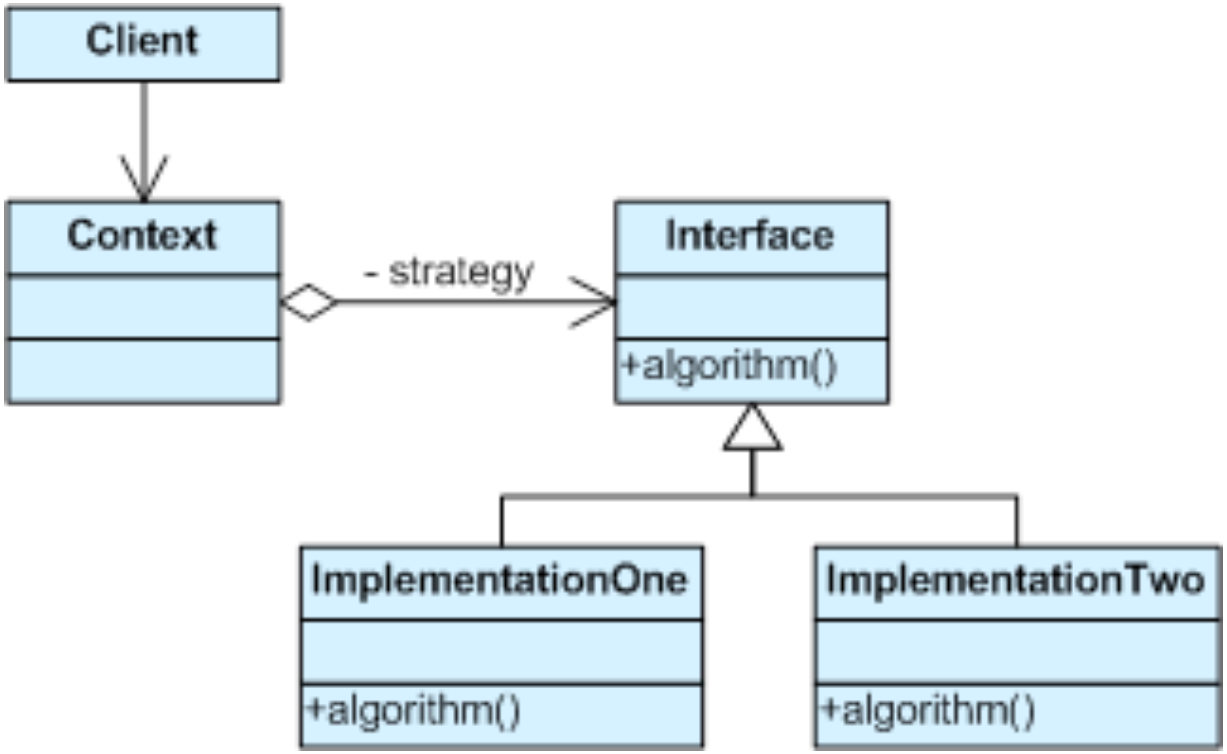




Strategy

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Strategy



THE UNIVERSITY OF TEXAS AT AUSTIN

SEAL Software Evolution & Analysis Laboratory

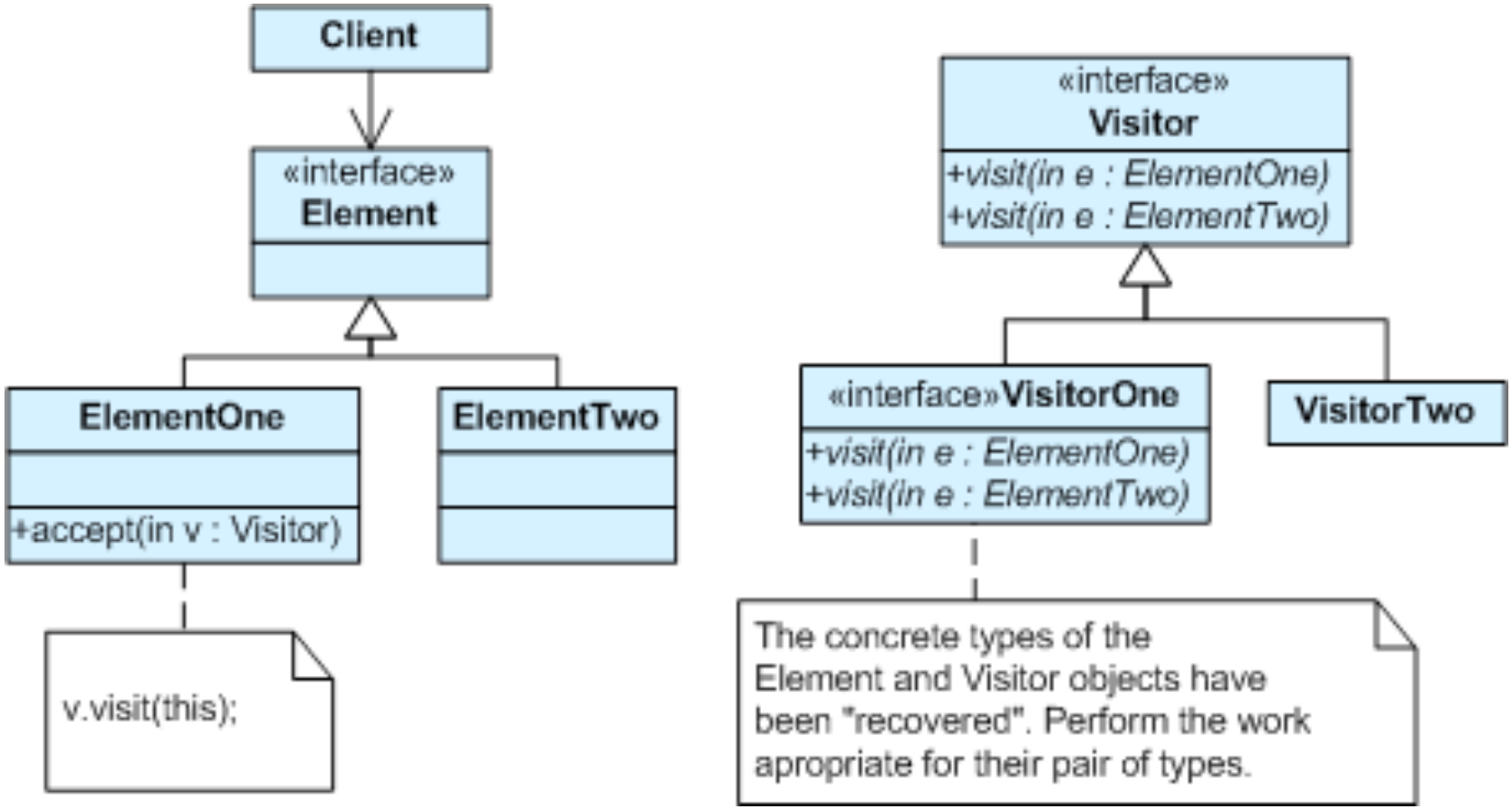
Visitor

- Intent: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

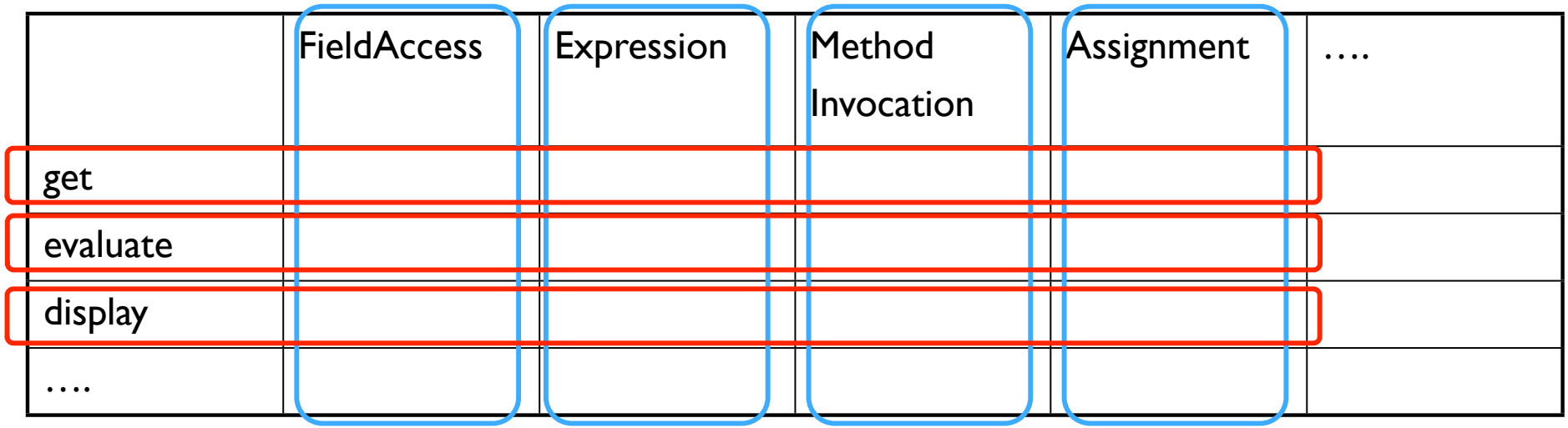
Visitor Design Pattern

- It allows OO programs to localize functional concerns using double dispatching.

Visitor



Functional vs. Data Concerns



Visitor Pattern
Step 1. Add Visitor Class

```
class ASTnode {
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
}
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
}
class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
}
class Assignment extends Expression{
```

```
class ASTNodeVisitor {
  void visitExpression(Expression e) {}
  void visitFieldAccess(FieldAccess f) {}
  void visitMethodInvocation(MethodInvocation m) {}
  void visitAssignment(Assignment a) {}
```

Visitor Pattern
Step 2. Extend the Visitor Class

```
class ASTnode {
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
}
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
}
class MethodInvocation extends Expression{
  int evaluate(Env e) {
  …}
}
class Assignment extends Expression{
```

```
abstract class ASTNodeVisitor {
  void visitExpression(Expression e) {}
  void visitFieldAccess(FieldAccess f) {}
  void visitMethodInvocation(MethodInvocation m) {}
  void visitAssignment(Assignment a) {}

class TypeCheckVisitor extends ASTNodeVisitor{
  void visitExpression(Expression e) {
  // type checking for Expression }
  void visitFieldAccess(FieldAccess f) {
  // type checking for FieldAccess}
...
```

Visitor Pattern
Step 3. Weave the Visitor Class

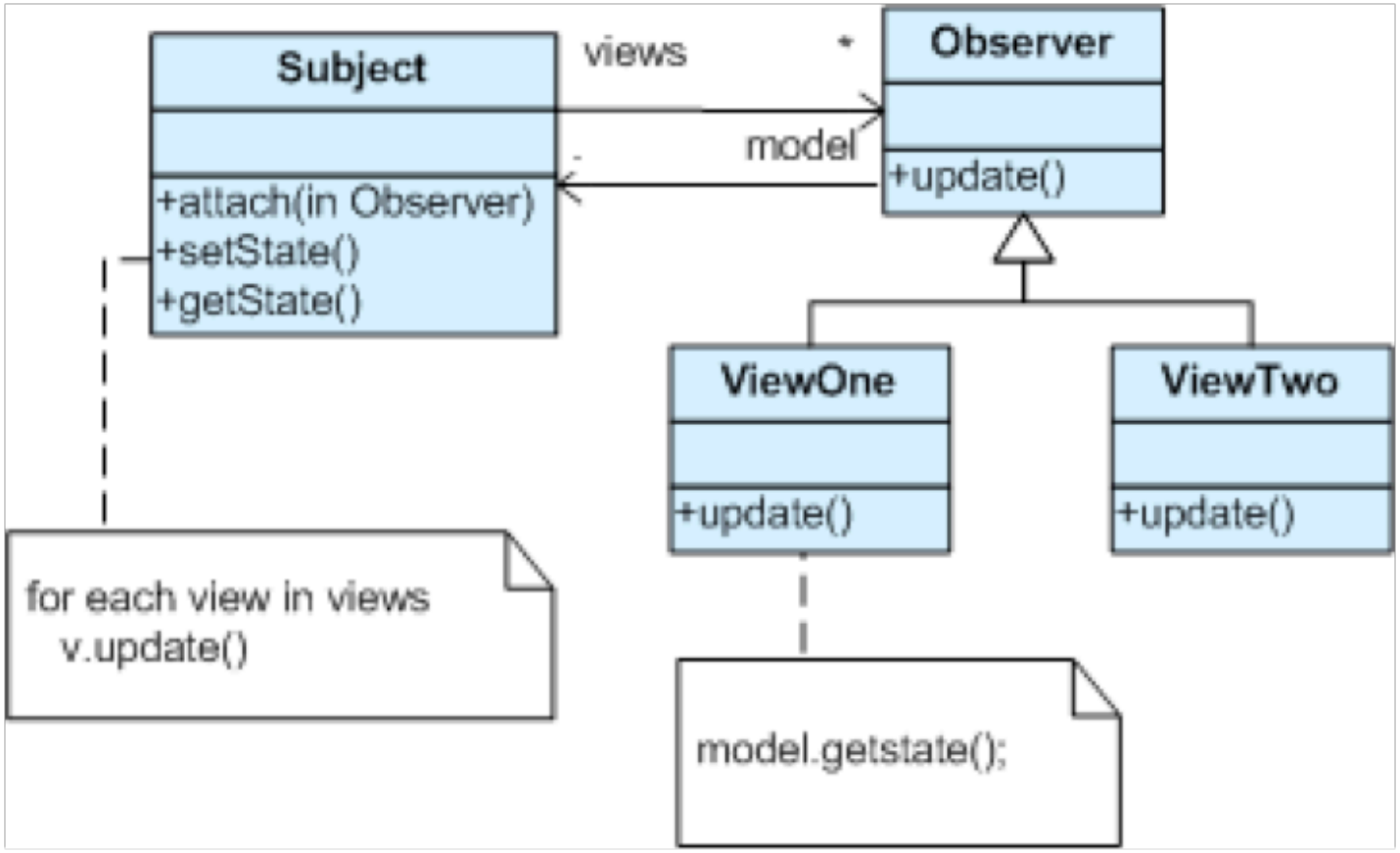
```
class ASTnode {
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
  void accept(Visitor v) { }
}
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
   void accept (Visitor v) {
   v.visitExpression(this); }
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
   void accept (Visitor v) {
   v.visitFieldAccess(this); }
}
```

```
class ASTNodeVisitor {
  void visitExpression(Expression e) {}
  void visitFieldAccess(FieldAccess f) {}
  void visitMethodInvocation(MethodInvocation m) {}
  void visitAssignment(Assignment a) {}

class TypeCheckVisitor extends ASTNodeVisitor{
  void visitExpression(Expression e) {
  // type checking for Expression }
  void visitFieldAccess(FieldAccess f) {
  // type checking for FieldAccess}
...

ASTNodeVisitor checker= new TypeCheckVisitor();
AST ast = getASTRoot();
// control  logic for recursively traversing AST
nodes{
        ASTNode node = ...
        node.accept(checker);
}
```

Recap

- Please review Observer, Visitor, Mediator, and Strategy patterns.
- At home, think about other example applications/ contexts whether these patterns might be applicable to.
- At home, identify change scenarios under which these design patterns may not be effective.

Review Question 1



- In the following code, which class corresponds to "Subject"?
  - Which classes correspond to Views?
  - Which classes correspond to Observer?
  - Can you provide applications or contexts where this design pattern is very useful?

```
interface AlarmListener { public void alarm(); }

class SensorSystem {
   private java.util.Vector listeners = new
java.util.Vector();

   public void register( AlarmListener al )
{ listeners.addElement( al ); }
   public void soundTheAlarm() {
      for (java.util.Enumeration e=listeners.elements();
e.hasMoreElements(); )
         ((AlarmListener)e.nextElement()).alarm();
}  }

class Lighting implements AlarmListener {
   public void alarm() { System.out.println( "lights
up" ); }
}

class Gates implements AlarmListener {
   public void alarm() { System.out.println( "gates
close" ); }
```

```
class CheckList {
   public void byTheNumbers() {  // Template Method design
pattern
      localize();
      isolate();
      identify(); }
   protected void localize() {
      System.out.println( "   establish a perimeter" ); }
   protected void isolate()  {
      System.out.println( "   isolate the grid" ); }
   protected void identify() {
      System.out.println( "   identify the source" ); }
}
                    // class inheri.  // type inheritance
class Surveillance extends CheckList implements
AlarmListener {
   public void alarm() {
      System.out.println( "Surveillance - by the
numbers:" );
      byTheNumbers(); }
   protected void isolate() { System.out.println( "   train
the cameras" ); }
}
```

```
public class ClassVersusInterface {
    public static void main( String[] args ) {
        SensorSystem ss = new SensorSystem();
        ss.register( new Gates()         );
        ss.register( new Lighting()      );
        ss.register( new Surveillance() );
        ss.soundTheAlarm()

}}
```

Review Question 2

1. Discuss the potential weaknesses of this following implementation from a program understanding perspective.

2. Add a new AST node type ArrayCreation expression to this implementation. Add all necessary extensions.

Review Question 2

3. Add a new feature, prettyPrint using the Visitor Pattern. Add all necessary extensions

4. Discuss strengths and weaknesses of this Visitor Pattern implementation with respect to changeability. You answer must address these two questions. (1) What kinds of evolutionary changes can be easily localized based on this implementation? (2) What kinds of evolutionary changes cannot be easily localized based on this implementation?

```
class ASTnode {
  int evaluate(Env e){
  …}
  void set(ASTnode n) {
  …}
  ASTnode get() {
  …}
  String display() {
  }
  void accept(Visitor v) { }
}
class Expression extends ASTnode {
   int evaluate(Env e) {
   …}
   void accept (Visitor v) {
      v.visitExpression(this); }
}
class FieldAccess extends Expression {
   int evaluate(Env e) {
   …}
   void accept (Visitor v) {
      v.visitFieldAccess(this); }
}
```

```
class ASTNodeVisitor {
  void visitExpression(Expression e) {}
  void visitFieldAccess(FieldAccess f) {}
  void visitMethodInvocation(MethodInvocation m) {}
  void visitAssignment(Assignment a) {}

class TypeCheckVisitor extends ASTNodeVisitor{
  void visitExpression(Expression e) {
  // type checking for Expression }
  void visitFieldAccess(FieldAccess f) {
  // type checking for FieldAccess}
  void visitMethodInvocation(MethodInvocation m) {
  // type checking for MethodInvocation}
  void visitAssignment(Assignment a) {
  // type checking for Assignment }
}
ASTNodeVisitor visitor= new TypeCheckVisitor();
AST ast = getASTRoot();
// control  logic for recursively traversing AST nodes{
          ASTNode node = ...
          node.accept(visitor);
}
```