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~~

Today's Agenda

- Introduction of design patterns
- ~~Class activity on AbstractFactory~~

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.

Reprise: What is "Engineering"?

Definitions abound. They have in common:

Creating cost-effective solutions ...

... to practical problems ...

... by applying scientific knowledge ...

... building things ...

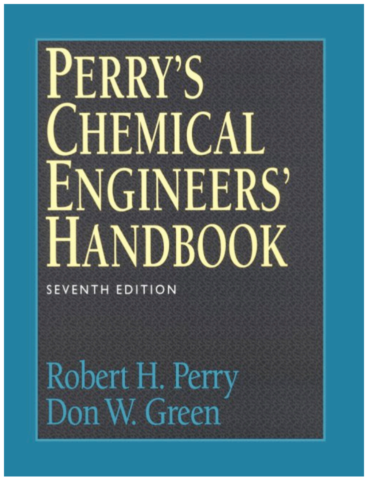
... in the service of mankind

Engineering enables ordinary people to do things that formerly required virtuosos

Software Architectures

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

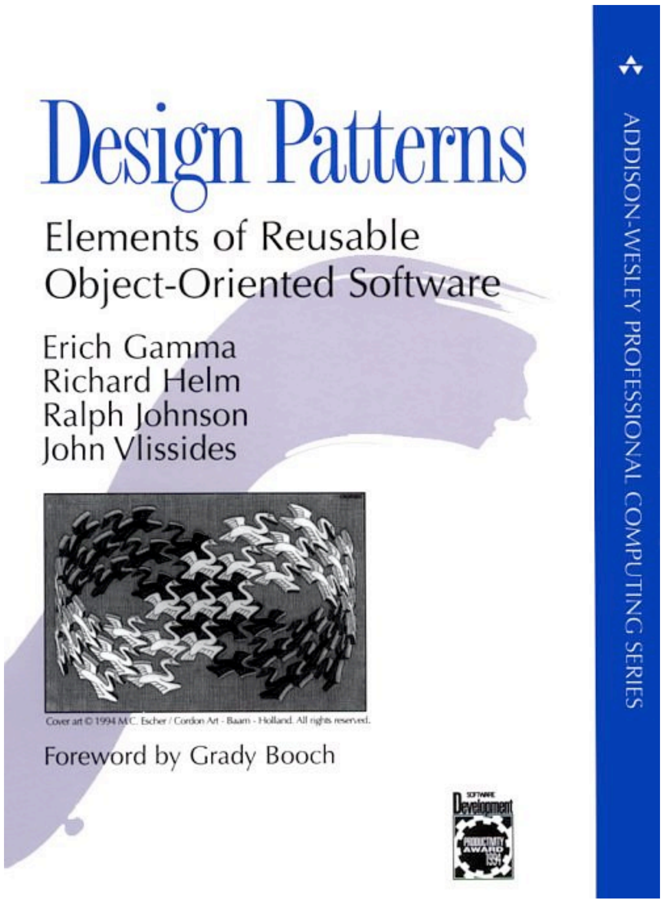
Other Precedents

- 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

Slide from Rob DeLine @ Microsoft Research

Why do we need Design Patterns?

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

Why do we need Design Patterns?

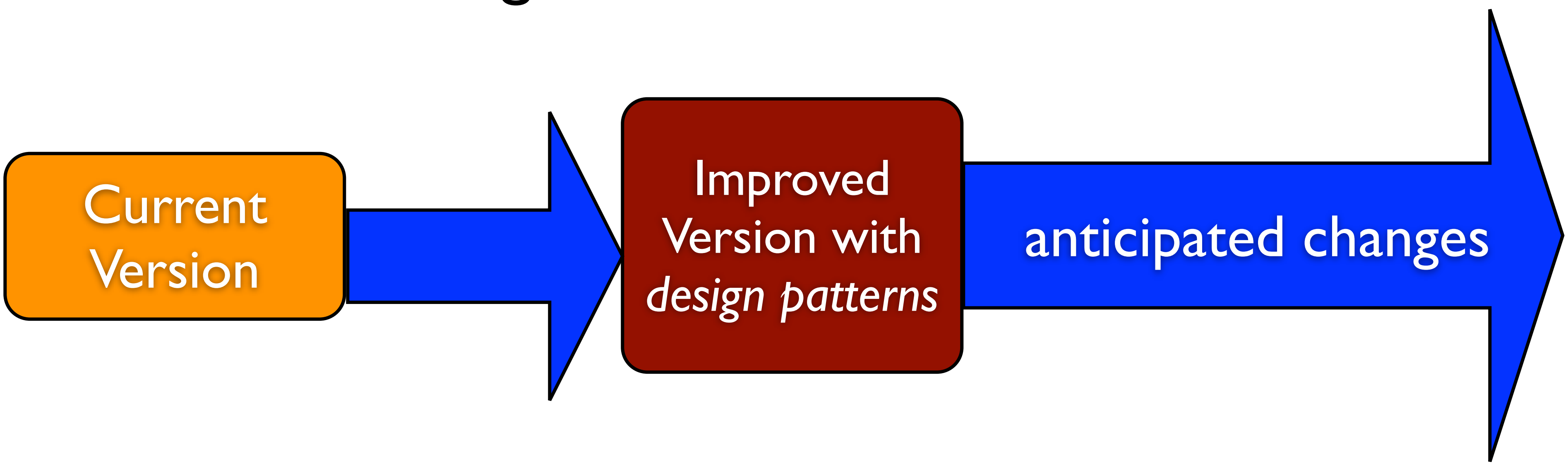
4. Provide a target for the reorganization or refactoring of class hierarchies

Current

anticipated changes

Why do we need Design Patterns?

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



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

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.

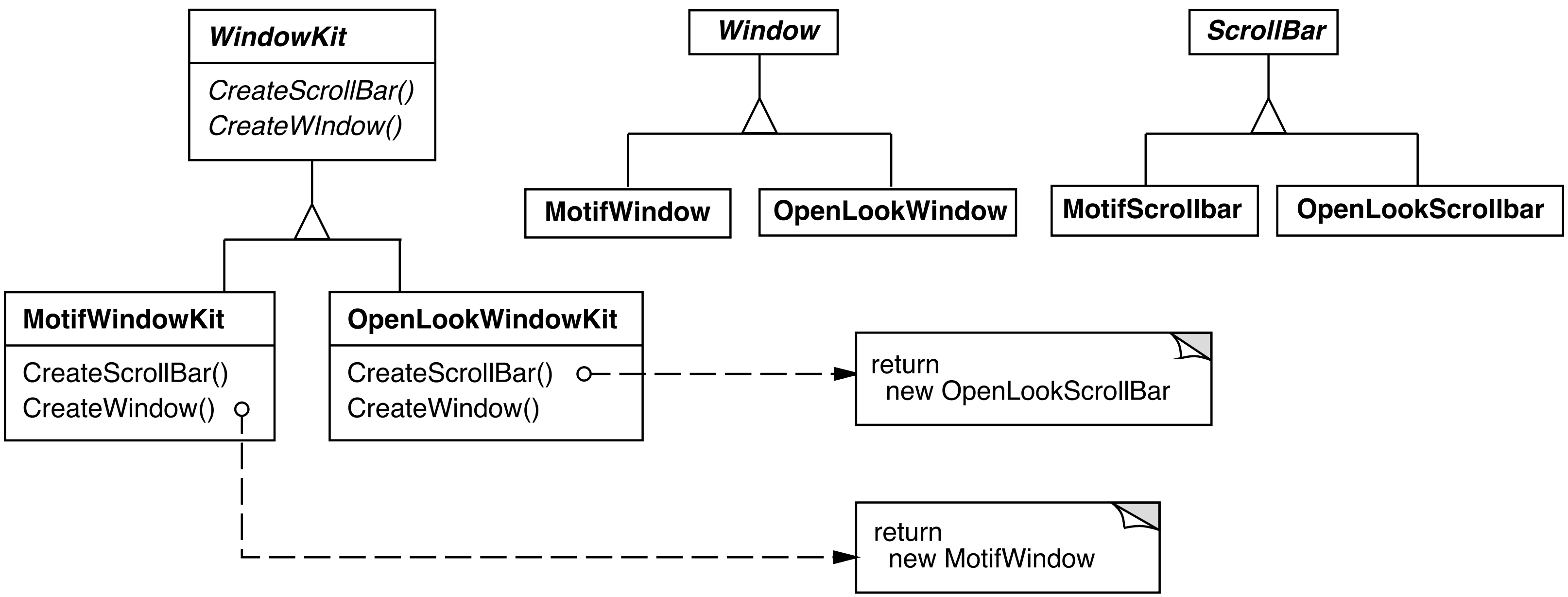
THE UNIVERSITY OF TEXAS AT AUSTIN

SEAL Software Evolution & Analysis Laboratory

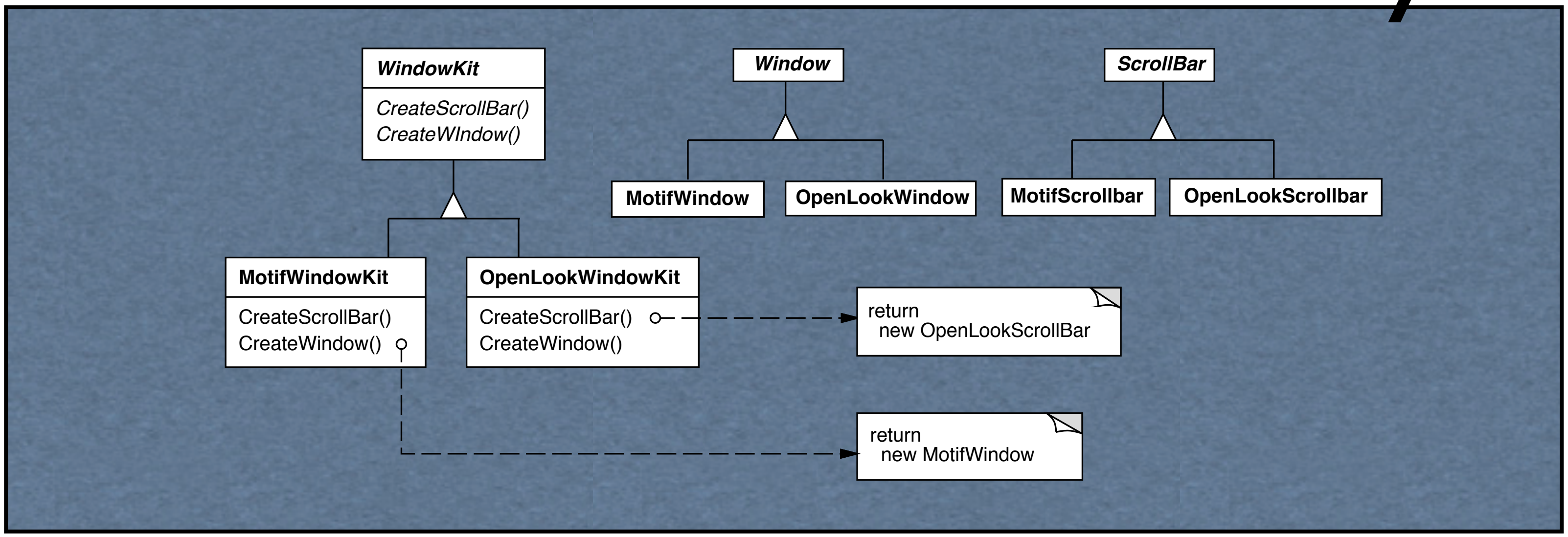
Solution: Abstract Factory

Solution
:Wrap the constructors in factory methods

Solution: Abstract Factory



Solution: Abstract Factory



Client Code

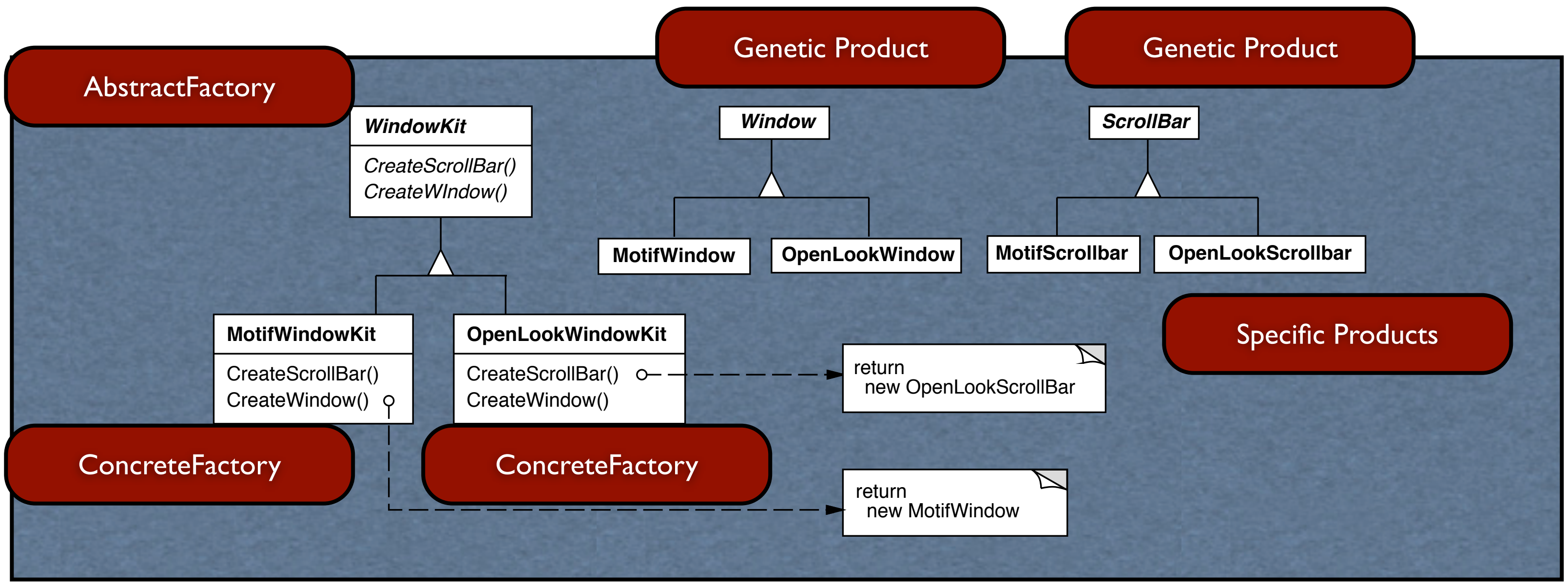
```
WindowKit kit = new MotifWindowKit();
kit.applnit();
```

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


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

Participants

generic



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

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

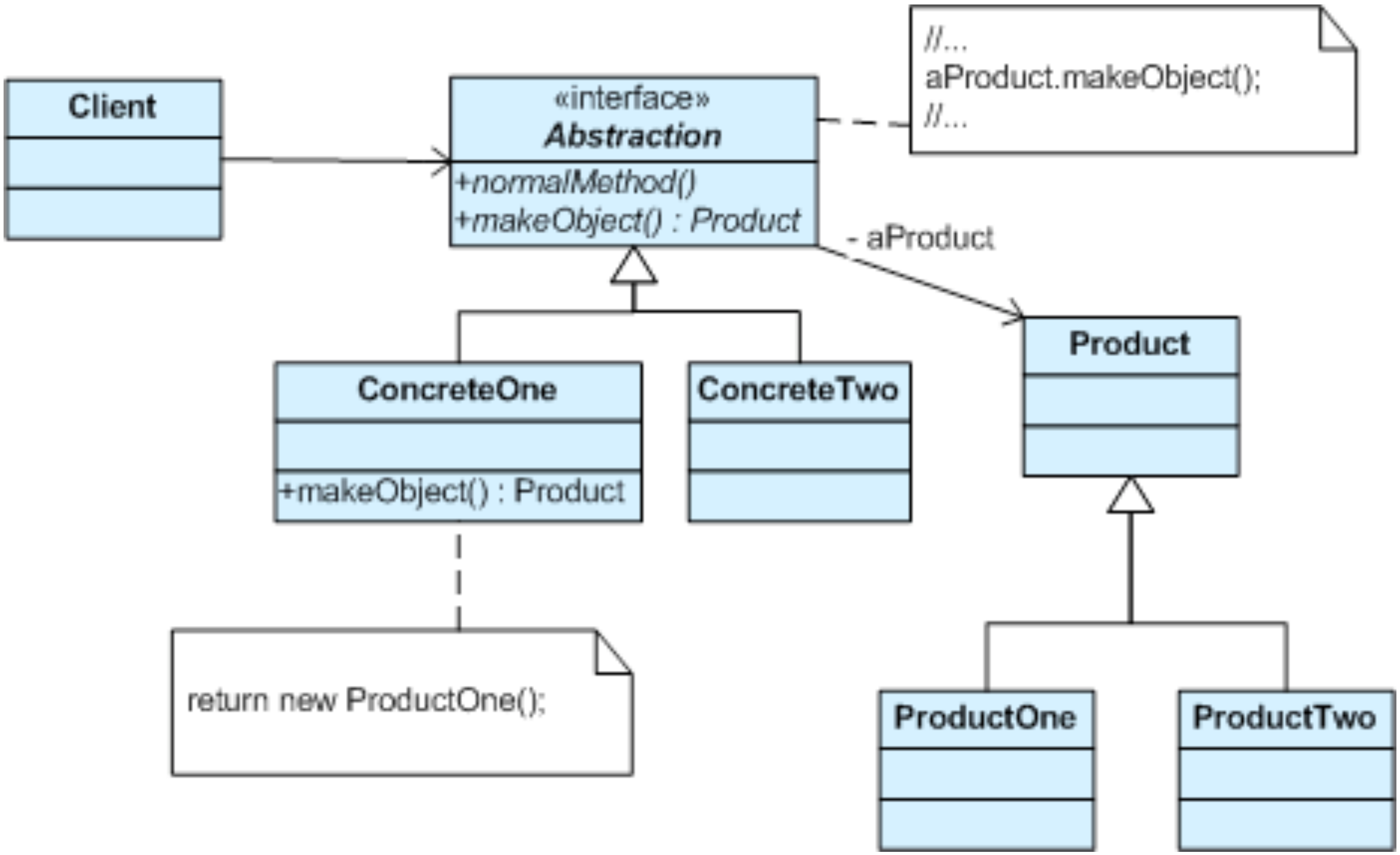
SEAL Software Evolution & Analysis Laboratory

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 example

```
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.

Today's Agenda

skip

- 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

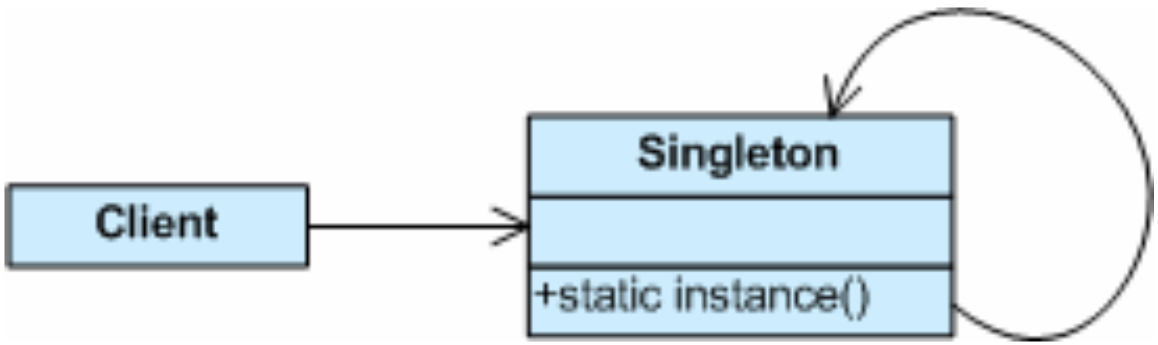
- Intent: Ensure a class has only one instance, and provide a global point of access to it.
- Encapsulated "just-in-time realization" or "initialization on first use"

"lazy"

THE UNIVERSITY OF TEXAS AT AUSTIN

SEAL Software Evolution & Analysis Laboratory

Singleton

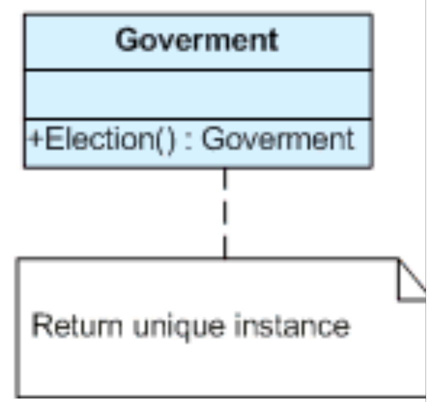

Client
Singleton
+static instance()


GlobalResource
-theInstance : GlobalResource
+getInstance() : GlobalResource

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;
  }
}
```

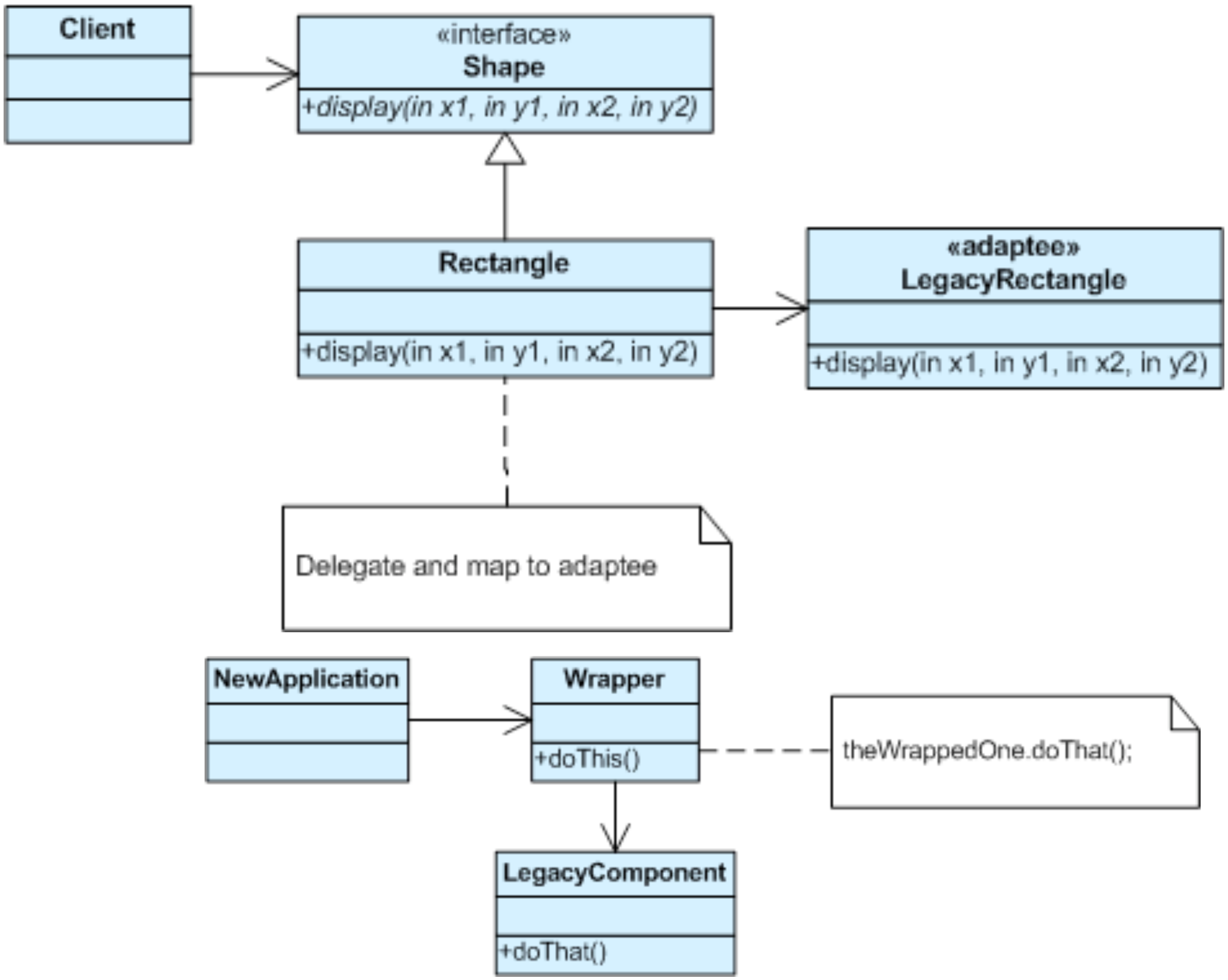
Adapter

- 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



Client
«interface»
Shape
+display(in x1, in y1, in x2, in y2)
Rectangle
+display(in x1, in y1, in x2, in y2)
«adaptee»
LegacyRectangle
+display(in x1, in y1, in x2, in y2)
Delegate and map to adaptee
NewApplication
Wrapper
+doThis()
theWrappedOne.doThat();
LegacyComponent
+doThat()

Class Activity on Adaptor

skip

Flyweight

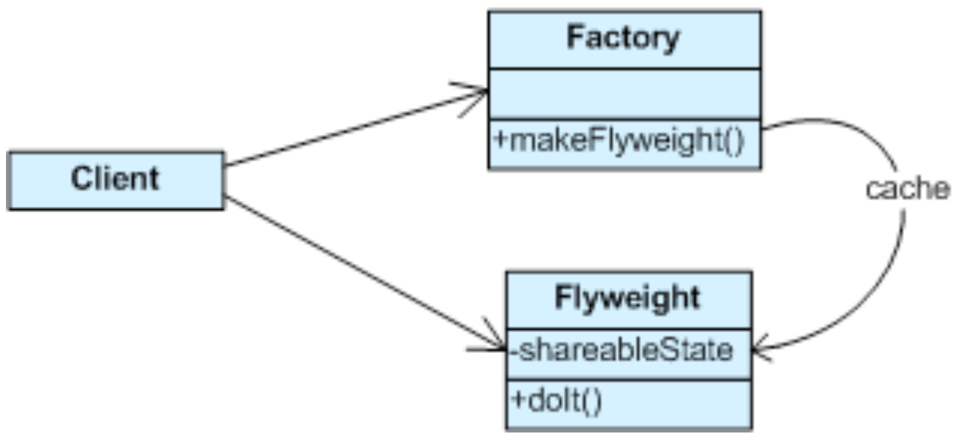
- Problem
  - Designing objects down to the lowest levels of system 'granularity' provides optimal flexibility but can be unacceptably expensive in terms of performance and memory usage

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


Factory
+makeFlyweight()
Client
cache
Flyweight
-shareableState
+doIt()

Bridge

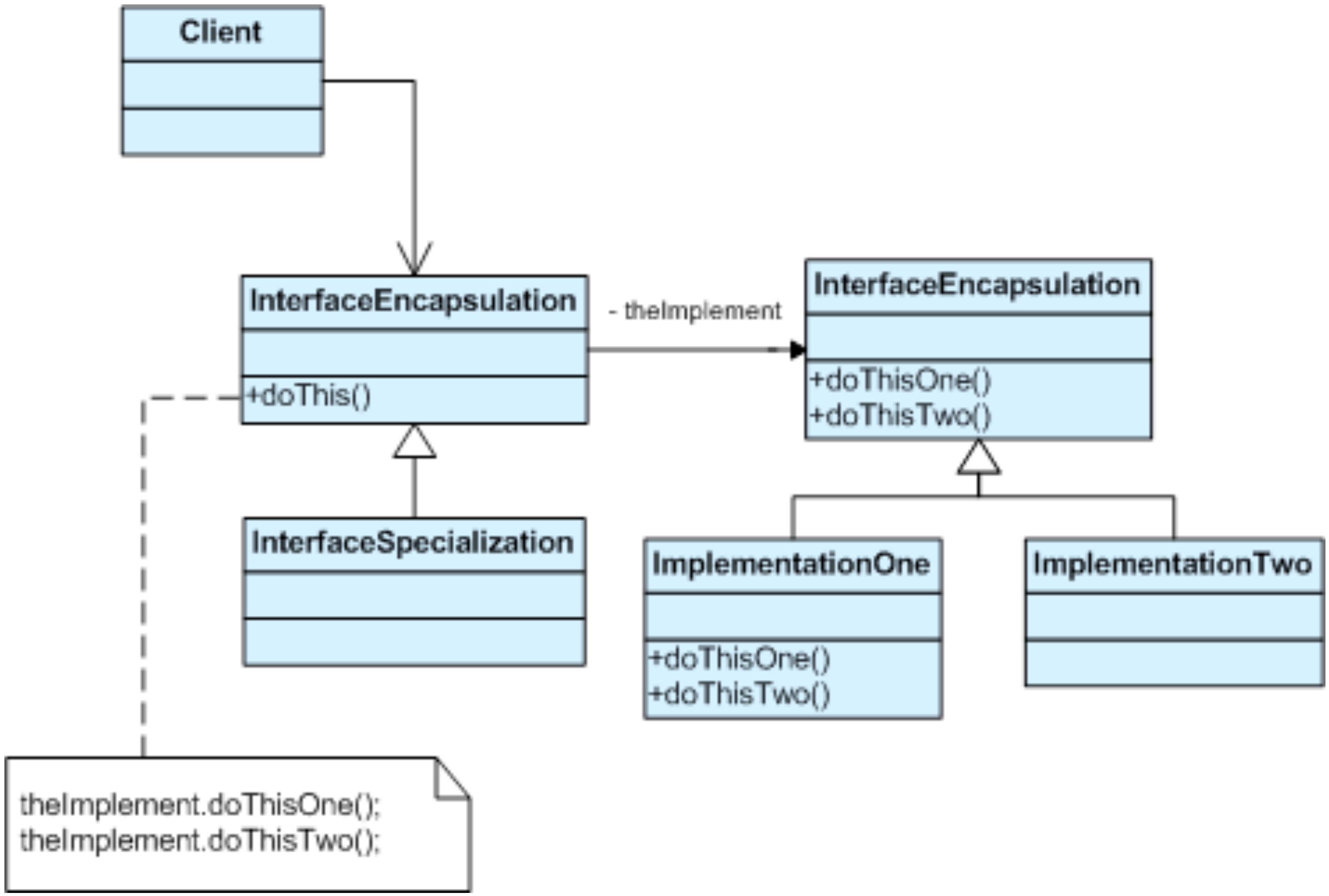
- Problem
  - "Hardening of the software arteries" has occurred by using subclasses of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation

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


Client
InterfaceEncapsulation
+doThis()
- theImplement
InterfaceEncapsulation
+doThisOne()
+doThisTwo()
InterfaceSpecialization
ImplementationOne
+doThisOne()
+doThisTwo()
ImplementationTwo
theImplement.doThisOne();
theImplement.doThisTwo();

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.

~~Today's Agenda~~

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

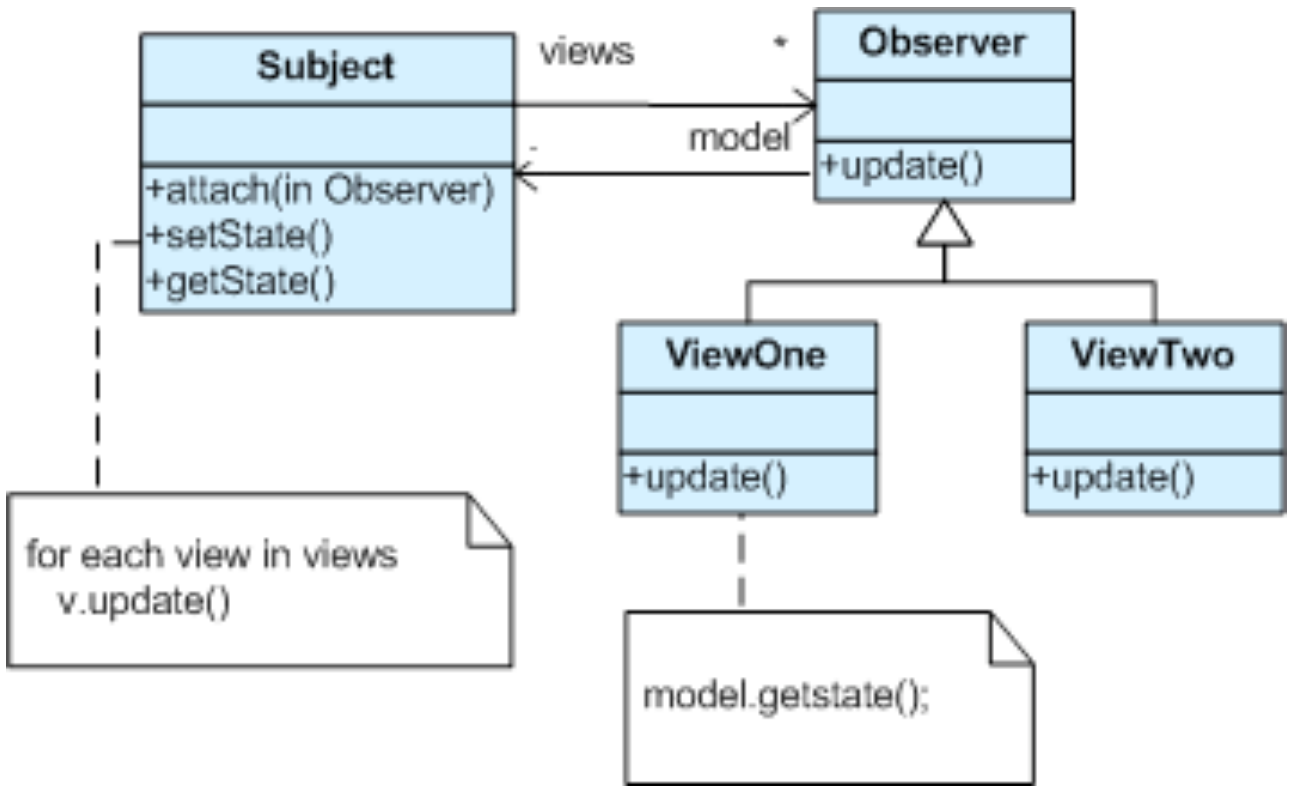
Observer

- 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

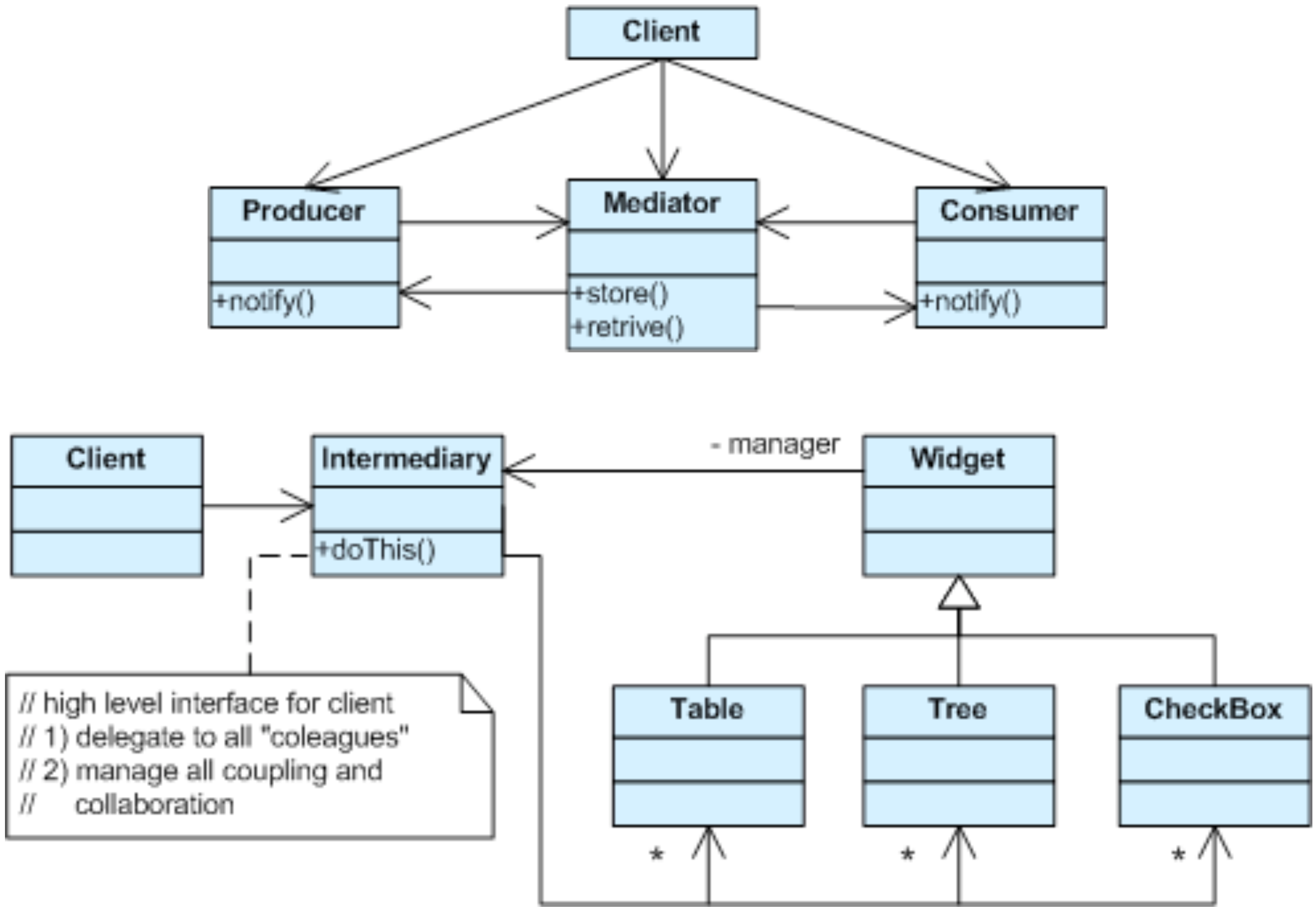
Observer


Subject
+attach(in Observer)
+setState()
+getState()
views
*
model
Observer
+update()
ViewOne
+update()
ViewTwo
+update()
for each view in views
v.update()
model.getstate();

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

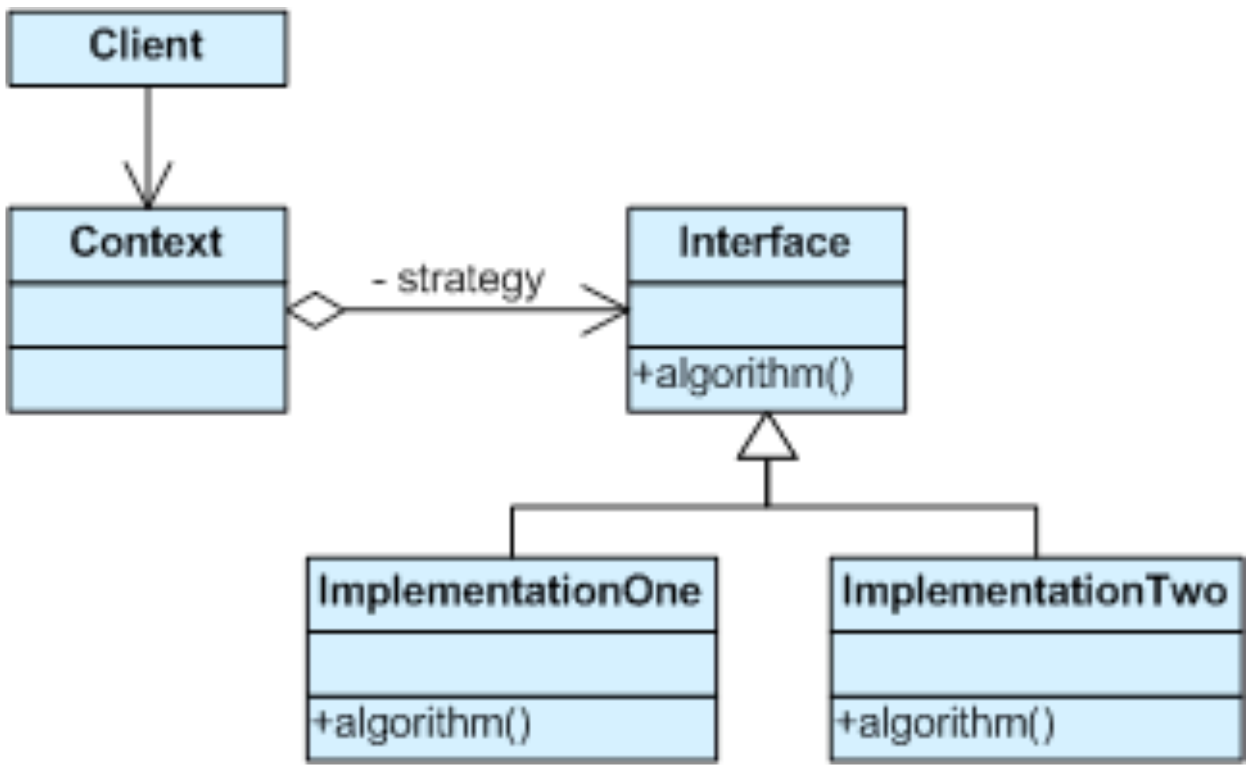


Client
Producer
+notify()
Mediator
+store()
+retrive()
Consumer
+notify()
Client
Intermediary
+doThis()
- manager
Widget
// high level interface for client
// 1) delegate to all "coleagues"
// 2) manage all coupling and
// collaboration
Table
Tree
CheckBox
*
*
*

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



Client
Context
- strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()

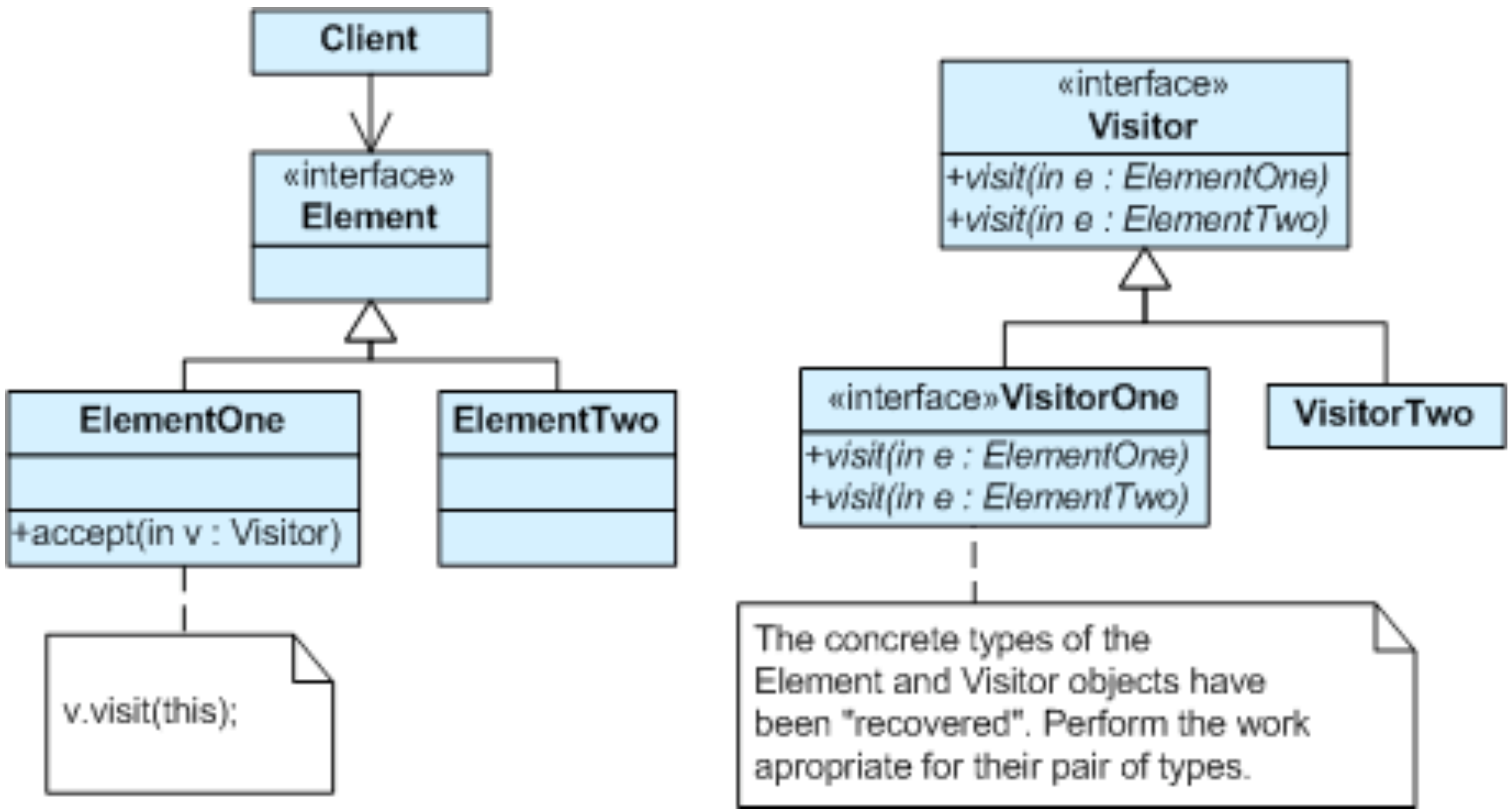
Visitor

- 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


Client
«interface»
Element
ElementOne
+accept(in v : Visitor)
ElementTwo
v.visit(this);
«interface»
Visitor
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
«interface»VisitorOne
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
VisitorTwo
The concrete types of the Element and Visitor objects have been "recovered". Perform the work apropriate for their pair of types.

Functional vs. Data Concerns

	FieldAccess	Expression	Method Invocation	Assignment	....
get					
evaluate					
display					
....					

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) {}
```

AST
abstract
syntax
tree

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}
...
```

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

Recap

- Please review Observer, Visitor, Mediator, 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.