Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 6:

Using Design Patterns

Lecture 6

www.lloseng.com

Object Oriented Software Engineering

DESIGN PATTERNS

The Gang-of-Four: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Design Patterns: Elements of Reusable Object Oriented Software (1995).

Background

- Designing reusable software is difficult
  - finding good objects and abstractions
  - flexibility, modularity, elegance ⇒ reuse
  - takes time for the to emerge, trial and error
- Successful design do exist
  - exhibit recurring class and object structure
- How to describe these recurring structures?

Alexander's Pattern Languages

- What is it that gives a building its Quality ?
  - freedom, life, comfort, harmony
- Pattern: solution to a problem in a context
  - Entrance transition
  - Intimacy gradient
  - Light on two sides of every room
- Linked patterns ⇒ Pattern Language
  - 2534 patterns, coarse to fine grain

The Timeless Way of Building (1979)
A Pattern Language (1977)

A Design Pattern

- Describes a recurring design structure
  - abstracts from concrete designs
  - identifies classes, collaborations, responsibilities
  - applicability, trade-offs, consequences
- Examples
  - Observer: MVC
  - Strategy: algorithms as objects
  - Composite: recursive structures

Pattern description

Context:

- The general situation in which the pattern applies

Problem:

—A short sentence or two raising the main difficulty.

Forces:

- The issues or concerns to consider when solving the problem

Solution:

- The recommended way to solve the problem in the given context.
  —'to balance the forces'

Antipatterns: (Optional)

- Solutions that are inferior or do not work in this context.

Related patterns: (Optional)

- Patterns that are similar to this pattern.

References:

- Who developed or inspired the pattern.

Patterns are not Designs

- Must be instantiated
  - evaluate trade-offs and consequences
  - make design and implementation decisions
  - implement, combine with other patterns

Patterns are not Frameworks

- Frameworks codify designs for solving a family of problems in a specific domain
  - abstract, cooperating classes
- Customized by
  - user defined subclasses
  - composition of objects
- Contain instances of multiple patterns

Catalog of 23 Design Patterns

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Façade	Mediator
	Flyweight	Observer
	Proxy	State
		Strategy
		Template Method
		Visitor

- "Used twice" rule
  - patterns discovered not invented

Robustness to Change

- Change is intrinsic to software
  - requirements, technology, platforms
  - alternative usage scenarios
- Robustness to change determines
  - ease of evolution
  - subsequent maintenance costs
  - ultimate reusability

Robustness to Change

- Each pattern addresses a particular variation
  - algorithms, implementations, instantiated classes
  - designs based on patterns robust to these variations
- Consider variations during life-cycle
  - up-front analysis of variations
  - understand nature of change and "hot-spots"
  - identify and apply pattern

Design Confidence

- General inexperience with objects
  - is my design OK?
- Patterns engender confidence
  - used-twice rule and blame the "Gang of Four"
  - still leaves room for creativity
- Most people know the patterns
  - but partially, not with full understanding
  - liberating to know that others have similar designs
  - patterns improve with use

Common Problems

- Taming over-enthusiasm
  - you "win" if you have the most patterns
  - solving the wrong problem
  - associated expense and cost
  - everything solved by the last pattern you learned
- Structure instead of Intent
  - everything is a Strategy
  - patterns use similar constructs

Finding the Right Pattern

- Not always clear when pattern is applicable
  - hard to ask right questions during design
  - requires expertise in both domain and patterns
- Once pattern found, design falls in place
  - evaluate tradeoffs, and implement variation

A Common Design Vocabulary

- Design language beyond technology
  - abstractions about problem not implementation
- "Let's use an Observer here"
  - increased design velocity, culture
- Shared vocabulary
  - within/across teams, up/down management
  - can exclude those not involved

Creational Patterns

- Abstract the creation of objects
  - inheritance allow variation in the instantiated class
  - encapsulate knowledge about concrete classes used
  - hide the creation process
- Creational patterns let users configure which objects are used
  - configuration can exhibit various degrees of dynamism

Example

- A Maze composed of Rooms, Doors and Walls

```
Maze create() {
      Maze maze = new Maze();
      Room r1   = new Room(1);
      Room r2   = new Room(2);
      Door door = new Door(r1, r2);
      maze.addRoom(r1);
      maze.addRoom(r2);
      r1->setSide(North, new Wall());
      r1->setSide(East, door);
      r1->setSide(South, new Wall());
      r1->setSide(West, new Wall());
      r2->setSide(North, new Wall());
      r2->setSide(East, new Wall());
      r2->setSide(South, new Wall());
      r2->setSide(West, door);
      return maze;
}
```


Example

- This design is inflexible
  - why ?

Example

- Creational patterns avoid hard coding the classes that get instantiated.
  - **Factory Method:** use methods instead of constructors
  - Abstract Factory: parameterize method with a creator object
  - **Builder:** parameterize method with an object that constructs a complete maze
  - **Prototype:** parameterize method with prototypical objects for all components of a maze

Factory Method

- **INTENT**
  - interface for creating an object, but choice of object's concrete class delegated to subclass
- **MOTIVATION**
  - frameworks often must instantiate classes but for each use of the framework different concrete classes may need to be created
- **APPLICABILITY**
  - a class can't anticipate the objects it must create
  - a class wants its subclasses to specify the objects it creates
- **CONSEQUENCES**
  - created class names not hard coded
  - may be used to connect parallel hierarchies

Factory Method

- **IMPLEMENTATION**
  - Varieties:
    - Creator class is abstract and does not implement creation methods (must be subclassed)
    - Creator class is concrete and provides a default implementation (can be subclassed)

Factory Method: The Problem

Factory Method: The Problem

1. Frameworks use abstract classes to define and maintain relationships between objects

Object Oriented Software Engineering

Factory Method: The Problem

1. Frameworks use abstract classes to define and maintain relationships between objects

2. Consider a framework for applications that present multiple documents to the user. A drawing application is an example.

Factory Method: The Problem

1. Frameworks use abstract classes to define and maintain relationships between objects
2. Consider a framework for applications that present multiple documents to the user. A drawing application is an example.
3. This framework defines two abstract classes: application and document. These ought to be sub classed by clients for application specific implementation.

Object Oriented Software Engineering

Factory Method: The Problem

1. Frameworks use abstract classes to define and maintain relationships between objects
2. Consider a framework for applications that present multiple documents to the user. A drawing application is an example.
3. This framework defines two abstract classes: application and document. These ought to be sub classed by clients for application specific implementation.
4. The application class will create and manage documents when required, e.g. when a New command is selected from the menu.

Factory Method Pattern: The Problem

5. Document sub class is *application specific*. Hence the Application class does not know *what kind of document* to create!

Factory Method Pattern: The Problem

Object Oriented Software Engineering

5. Document sub class is *application specific*. Hence the Application class does not know *what kind of document* to create!

6. *Problem*: The framework must instantiate classes but it only knows about the abstract classes, which it cannot initiate!

January 02, 2009 311

Factory Method Pattern: Structure

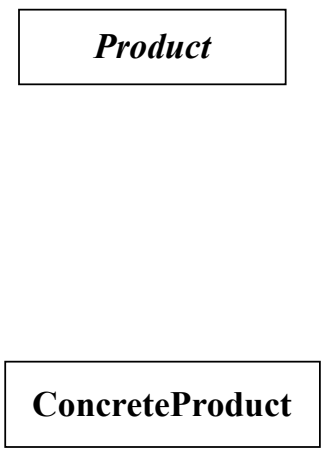
Object Oriented Software Engineering

Factory Method Pattern: Structure

Product

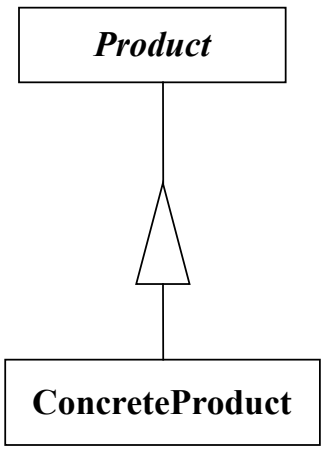
January 02, 2009 312

Factory Method Pattern: Structure

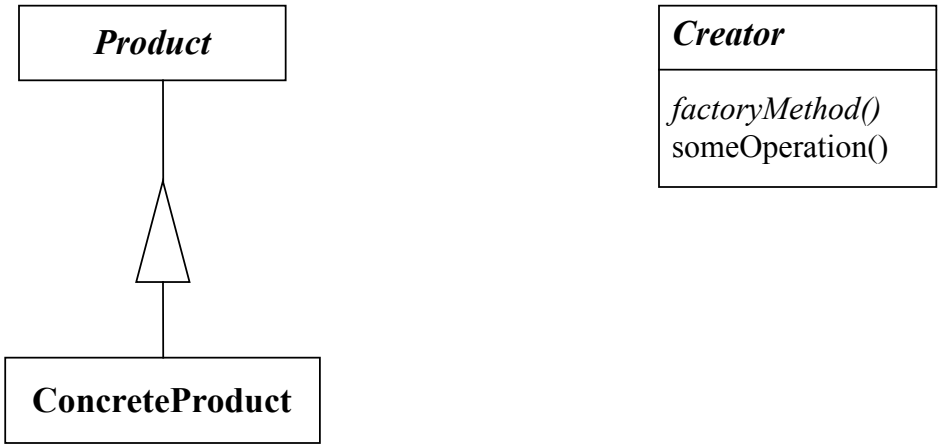


Object Oriented Software Engineering

Factory Method Pattern: Structure

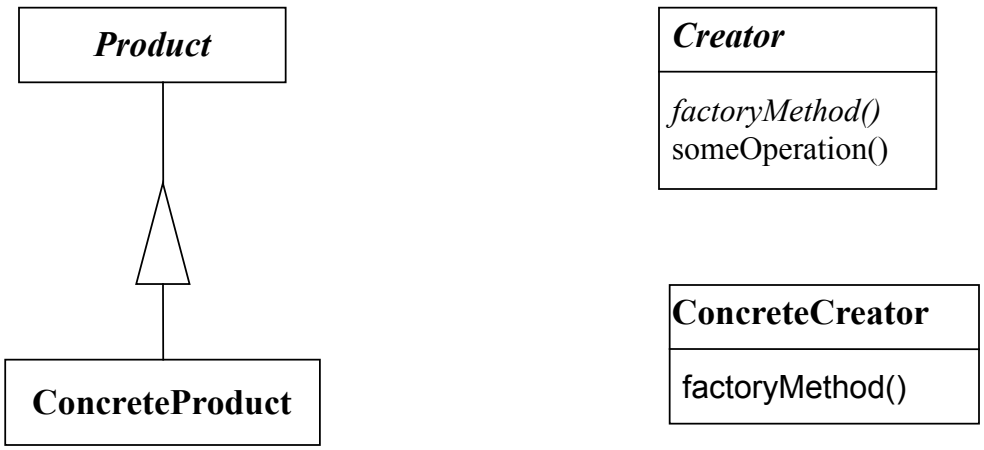


Factory Method Pattern: Structure

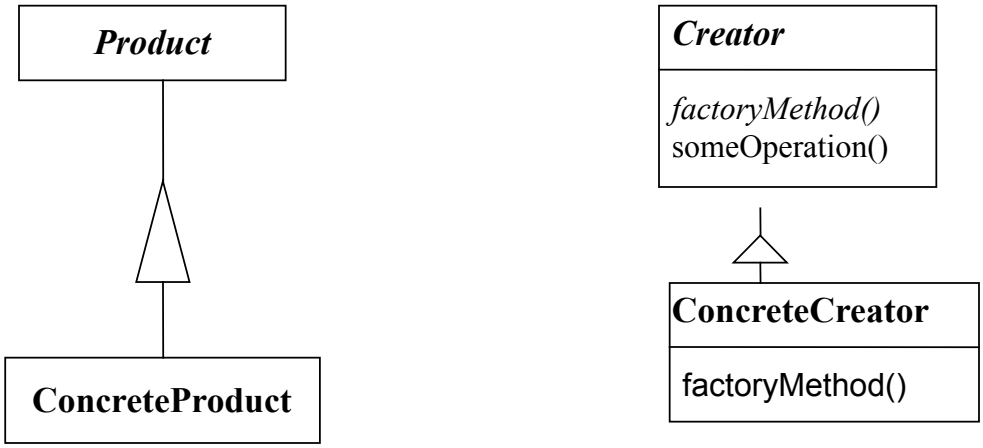


Object Oriented Software Engineering

Factory Method Pattern: Structure

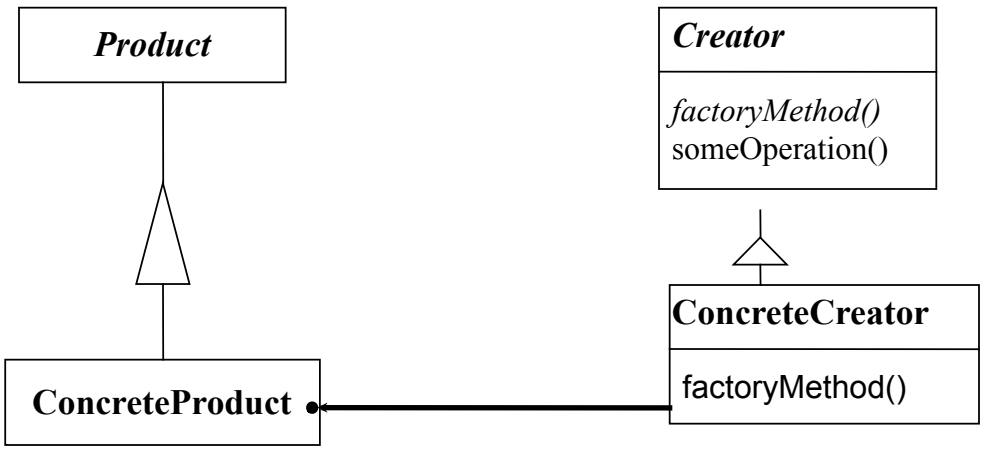


Factory Method Pattern: Structure

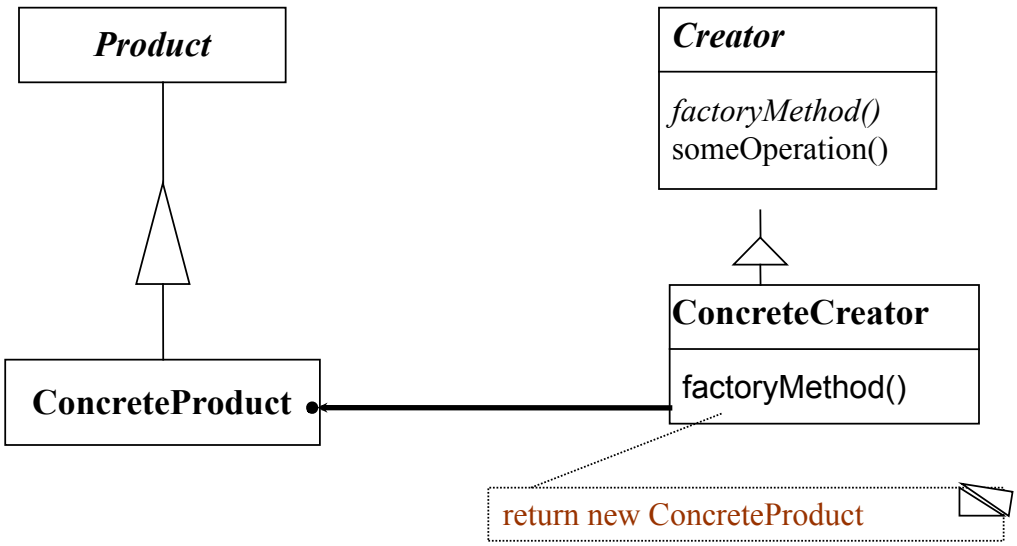


Object Oriented Software Engineering

Factory Method Pattern: Structure

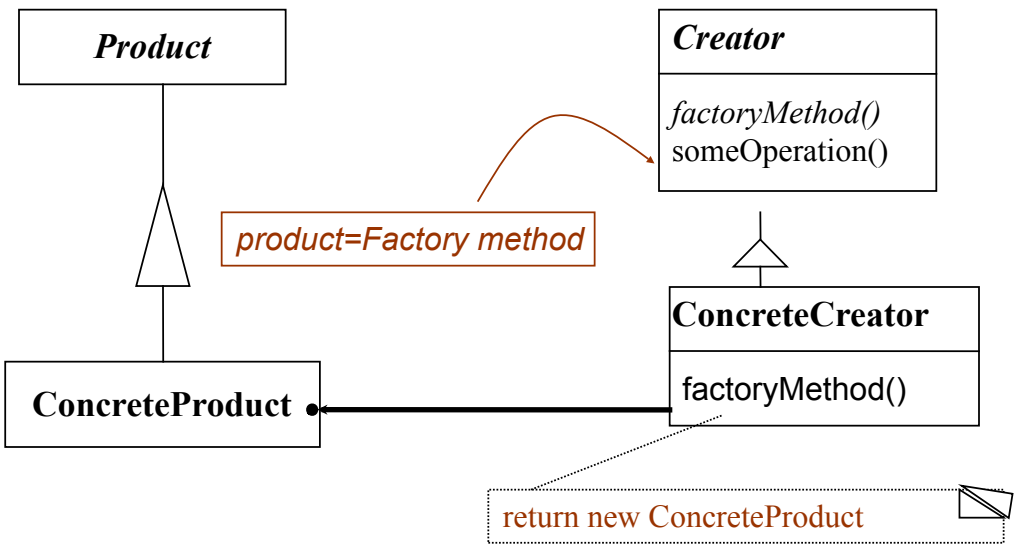


Factory Method Pattern: Structure



Object Oriented Software Engineering

Factory Method Pattern: Structure



Factory Method

- **IMPLEMENTATION**
  - Parameterized factory methods. eg:

```
class Creator {
  public Product create(ProductID id) {
        if (id == MINE) return new MyProduct();
        if (id == YOURS) return new YourProduct();
        return null;
} }
```

can be extended to

```
class MyCreator extends Creator {
  public Product create(ProductID id) {
        if (id == YOURS) return new MyProduct();
        if (id == THEIRS) return new YourProduct();
        return super.create(id);
} }
```


Factory Method

- **IMPLEMENTATION**
  - Genericity. e.g.:

```
abstract class Creator {
  public Product create();
}

class MyCreator<T> {
  public Product create() { return new T();}
}

...
MyCreator<YourProduct> factory = new
  MyCreator<YourProduct>();
```

Factory Method

- **Sample Code**

```
class Maze {
     public Maze create();

     public Maze makeMaze() { return new Maze(); }
     public Maze makeRoom(int n) { return new Room(n); }
     public Maze makeWall() { return new Wall(); }
     public Maze makeDoor(Room r1, Room r2) {return new Door(r1,r2);}
 }
```


Factory Method

- **Sample Code**

```
Maze create() {
     Maze maze = makeMaze();
     Room r1   = makeRoom(1);
     Room r2   = makeRoom(2);
     Door door = makeDoor(r1, r2);
     maze.addRoom(r1);
     maze.addRoom(r2);
     r1->setSide(North, makeWall());
     r1->setSide(East, door);
     r1->setSide(South, makeWall());
     r1->setSide(West, makeWall());
     r2->setSide(North, makeWall());
     r2->setSide(East, makeWall());
     r2->setSide(South, makeWall());
     r2->setSide(West, door);
     return maze;    }
```

Factory Method

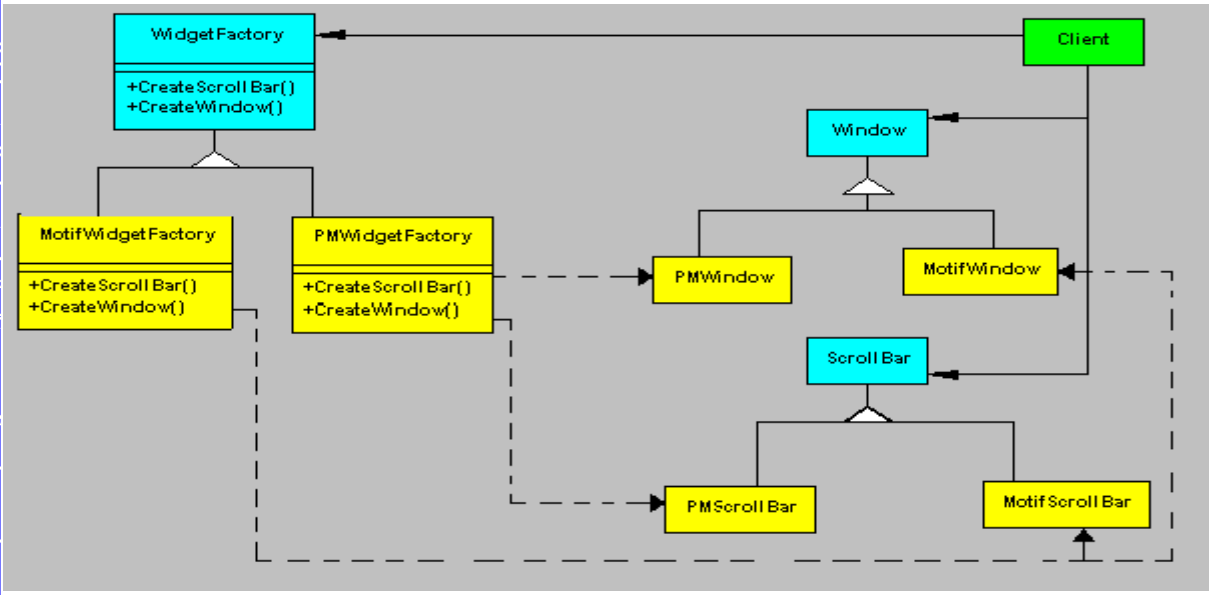
- **Sample Code**

```
class MazeBombsGame extends Maze {
      public Maze makeRoom(int n) { return new RoomWithABomb(n); }
      public Maze makeWall() { return new BombedWall();
}
```


Abstract Factory

- **INTENT**
  - interface for creating families of related objects without revealing their concrete classes
- **MOTIVATION**
  - user interface toolkit that supports multiple look-and-feel standards (e.g. Motif and Presentation Manager)
  - applications should be portable across window managers

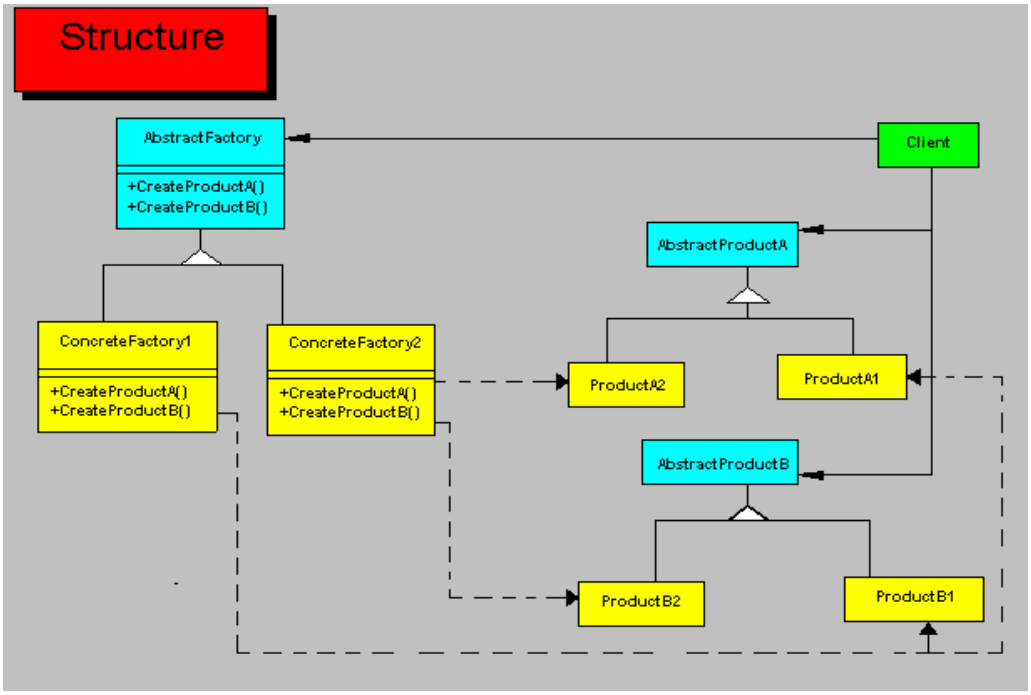
Abstract Factory



Abstract Factory

- **APPLICABILITY**
  - a system should be independent of how its products are created, composed and presented
  - a system should be configured with one of multiple families of products
  - a family of products is designed to be used together, and you need to enforce this constraint

Abstract Factory



Abstract Factory

- **CONSEQUENCES**
  - isolates concrete classes. clients only manipulate abstract interfaces
  - it makes changing product families easy.
  - it promotes consistency among products. forbids arbitrary mix and match across families
  - difficult to add new products, requires extending the interface of all factory classes

Abstract Factory

- **IMPLEMENTATION**
  - factories as singletons as only one instance of the class is needed (usually)
  - creating the products:
    - one factory method per kind of product (override the factory method to specify the actual objects to create)
    - when there can be a large number of families (i.e. requiring many concrete factory classes) use the prototype pattern to have a single concrete factory

Abstract Factory

- **SAMPLE CODE**

```
class MazeFactory {
  public Maze makeMaze() { return new Maze(); }
  public Maze makeRoom(int n) { return new Room(n); }
  public Maze makeWall() { return new Wall(); }
  public Maze makeDoor(Room r1, Room r2) { return new Door(r1,
  r2);}
}
```

Abstract Factory

- **SAMPLE CODE**

```
Maze create(MazeFactory factory) {
        Maze maze = factory.makeMaze();
        Room r1   = factory.makeRoom(1);
        Room r2   = factory.makeRoom(2);
        Door door = factory.makeDoor(r1, r2);
        maze.addRoom(r1);
        maze.addRoom(r2);
        r1->setSide(North, factory.makeWall());
        r1->setSide(East, door);
        r1->setSide(South, factory.makeWall());
        r1->setSide(West, factory.makeWall());
        r2->setSide(North, factory.makeWall());
        r2->setSide(East, factory.makeWall());
        r2->setSide(South, factory.makeWall());
        r2->setSide(West, door);
        return maze;    }
```


Abstract Factory

- **SAMPLE CODE**

```
class EnchantedMazeFacotry extends MazeFactory {
     public EnchantedMazeFactory();
     Room makeRoom(int n) { return new EnchantedRoom(n); }
     Door makeDoor(Room r1, Room r2){
             return new MagicDoor(r1,r2);
     }
}
```

Abstract Factory

- **SAMPLE CODE**

```
class BombedMazeFacotry extends MazeFactory {
     public BombedMazeFactory();
     Room makeRoom(int n) { return new RoomWithABomb(n); }
     Door makeWall(){ return new BombedWall(r1,r2);}
}
```

can a RoomWithABomb use the special features of its walls?

Observer Pattern

Observer Pattern

- Separate presentational aspects from data.

Observer Pattern

- Separate presentational aspects from data.

Object Oriented Software Engineering

January 02, 2009 328

Observer Pattern

- Separate presentational aspects from data.
- Classes defining data and presentation can be reused.

Object Oriented Software Engineering

Observer Pattern

- Separate presentational aspects from data.
- Classes defining data and presentation can be reused.

January 02, 2009 328

Observer Pattern

- Separate presentational aspects from data.
- Classes defining 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.

Object Oriented Software Engineering

Observer Pattern

- Separate presentational aspects from data.
- Classes defining 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.

Observer Pattern

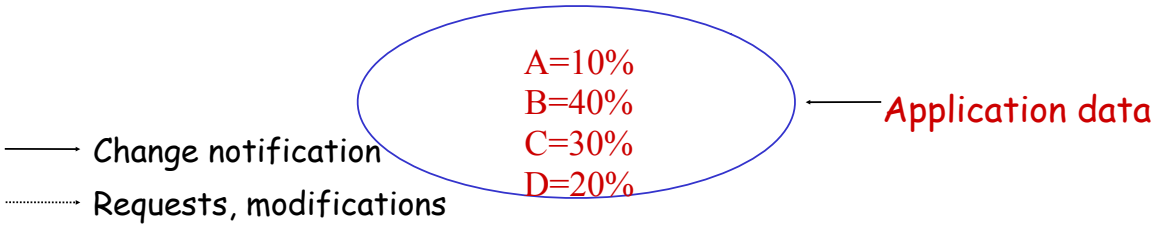
- Separate presentational aspects from data.
- Classes defining data and presentation can be reused.
- Change in one view automatically reflected in other views. Also, change in the application data is reflected in all views.
- Defines one-to-many dependency amongst objects so that when one object changes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

Observer Pattern

——→ Change notification

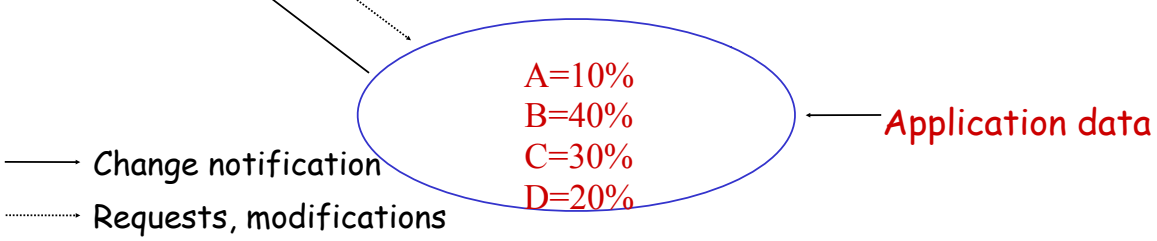
┈┈→ Requests, modifications

Observer Pattern

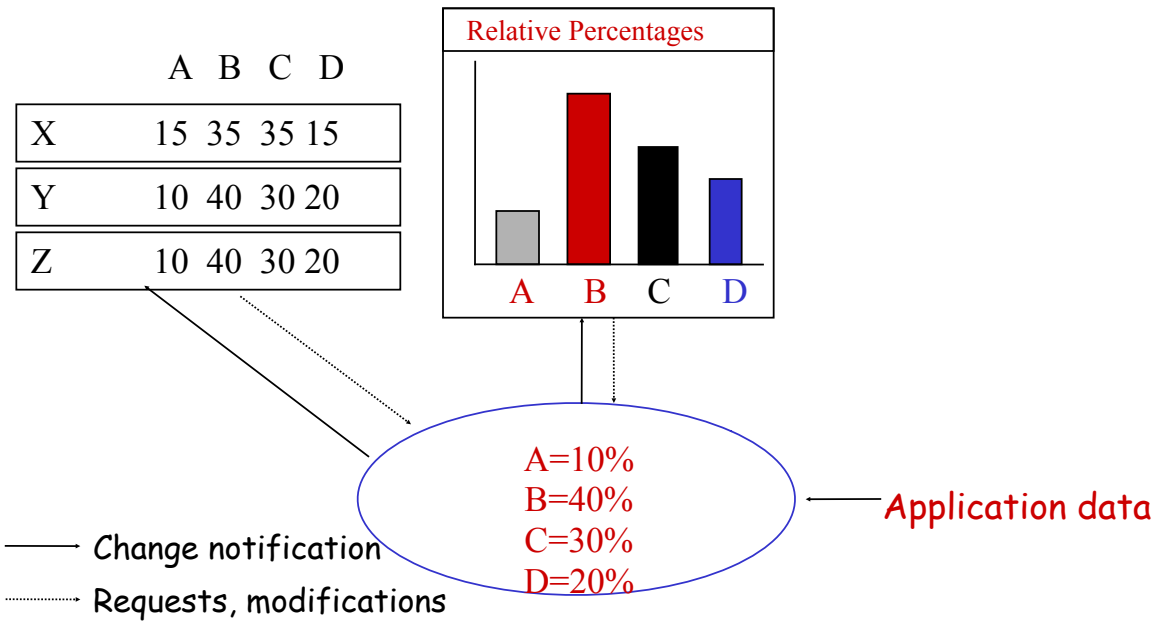


Observer Pattern

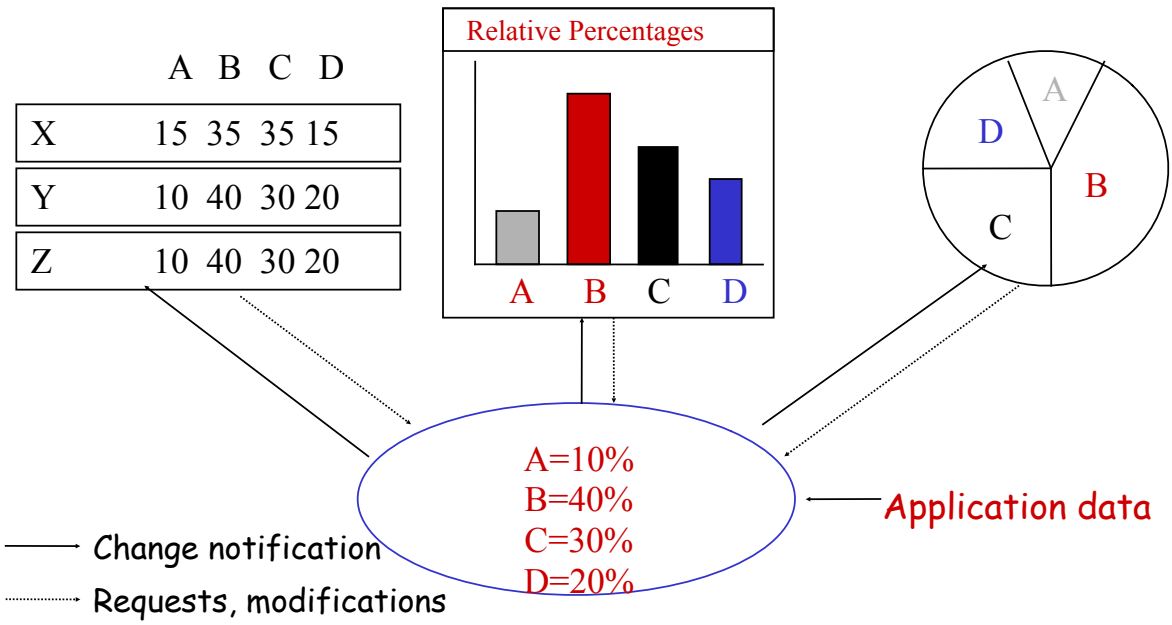
	A	B	C	D
X	15	35	35	15
Y	10	40	30	20
Z	10	40	30	20



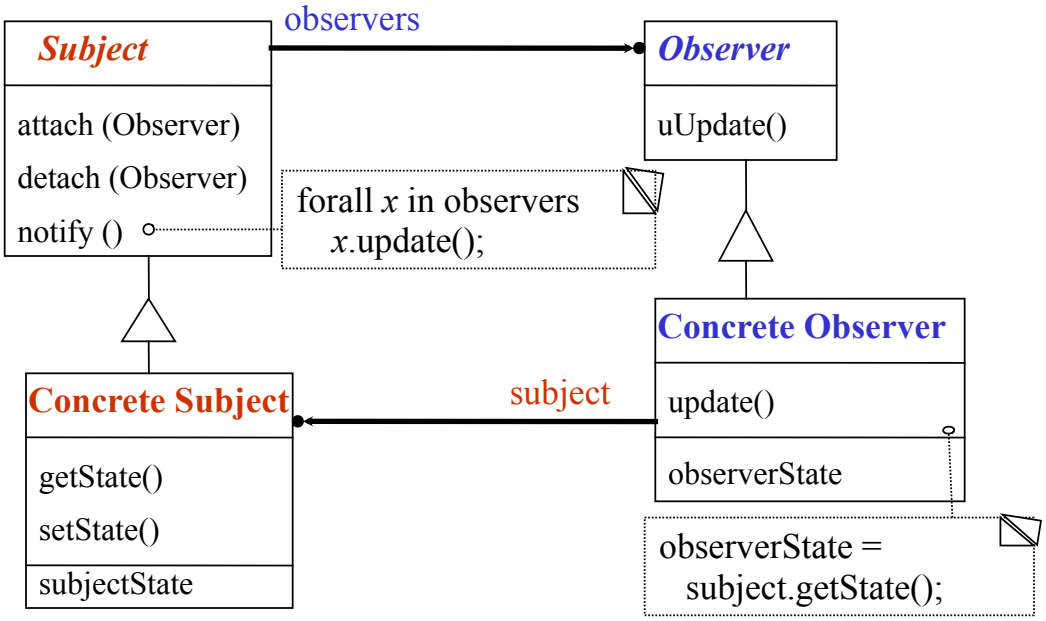
Observer Pattern



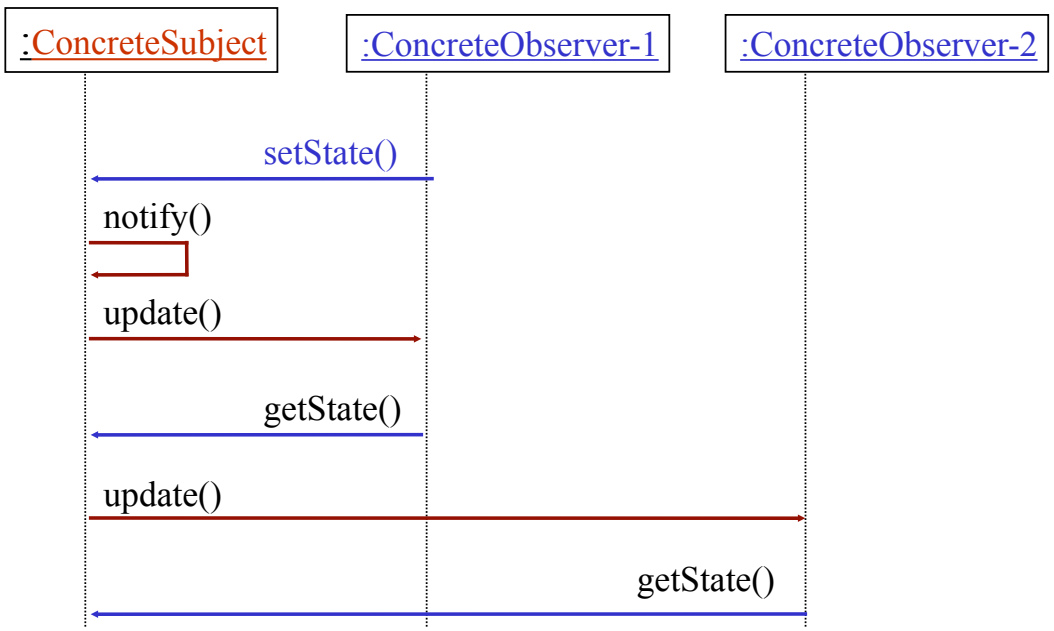
Observer Pattern



Observer Pattern



Class collaboration in Observer



Observer code

Object Oriented Software Engineering

Observer code

```
import Subject;
```

January 02, 2009 332

Observer code

```
import Subject;
```


Object Oriented Software Engineering

Observer code

import Subject;

Observer code

```
import Subject;
```


Object Oriented Software Engineering

Observer code

```
import Subject;


class Observer {
```

January 02, 2009 332

Observer code

```
import Subject;


class Observer {
```


Object Oriented Software Engineering

Observer code

import Subject;

class Observer {

Observer code

```
import Subject;


class Observer {
```


Object Oriented Software Engineering

Observer code

```
import Subject;


class Observer {


   public abstract void update(Subject theChangeSubject);
```

Observer code

```
import Subject;


class Observer {


   public abstract void update(Subject theChangeSubject);
   protected Observer();
```


Object Oriented Software Engineering

Observer code

```
import Subject;


class Observer {
```

Abstract class defining the Observer interface.

```
   public abstract void update(Subject theChangeSubject);
   protected Observer();
```

Observer code

```
import Subject;


class Observer {

   public abstract void update(Subject theChangeSubject);
   protected Observer();
```

Abstract class defining the Observer interface.

Note the support for multiple subjects.

Subject Code

Subject Code

```
abstract class Subject {
```


Object Oriented Software Engineering

Subject Code

```
abstract class Subject {

   public abstract void attach(Observer o);
```

Subject Code

```
abstract class Subject {

   public abstract void attach(Observer o);
   public abstract void detach(Observer o);
```


Object Oriented Software Engineering

Subject Code

```
abstract class Subject {

   public abstract void attach(Observer o);
   public abstract void detach(Observer o);
   public abstract void notify(Observer o);
```

January 02, 2009 333

Subject Code

```
abstract class Subject {

   public abstract void attach(Observer o);
   public abstract void detach(Observer o);
   public abstract void notify(Observer o);

   protected Subject();
```


Object Oriented Software Engineering

Subject Code

```
abstract class Subject {

   public abstract void attach(Observer o);
   public abstract void detach(Observer o);
   public abstract void notify(Observer o);

   protected Subject();

   private List observers_;
```

Subject Code

```
abstract class Subject {

   public abstract void attach(Observer o);
   public abstract void detach(Observer o);
   public abstract void notify(Observer o);

   protected Subject();

   private List observers_;
```

Could this be an interface?

Object Oriented Software Engineering

Subject Code

```
abstract class Subject {
```

Abstract class defining the Subject interface.

```
   public abstract void attach(Observer o);
   public abstract void detach(Observer o);
   public abstract void notify(Observer o);

   protected Subject();

   private List observers_;
```

Could this be an interface?

When to use the Observer Pattern?

Object Oriented Software Engineering

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 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.

Object Oriented Software Engineering

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.

January 02, 2009 334

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.

Object Oriented Software Engineering

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.

January 02, 2009 334

Observer Pattern: Consequences

Object Oriented Software Engineering

Observer Pattern: Consequences

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

Observer Pattern: Consequences

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

Object Oriented Software Engineering

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.

January 02, 2009 335

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.

Object Oriented Software Engineering

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.

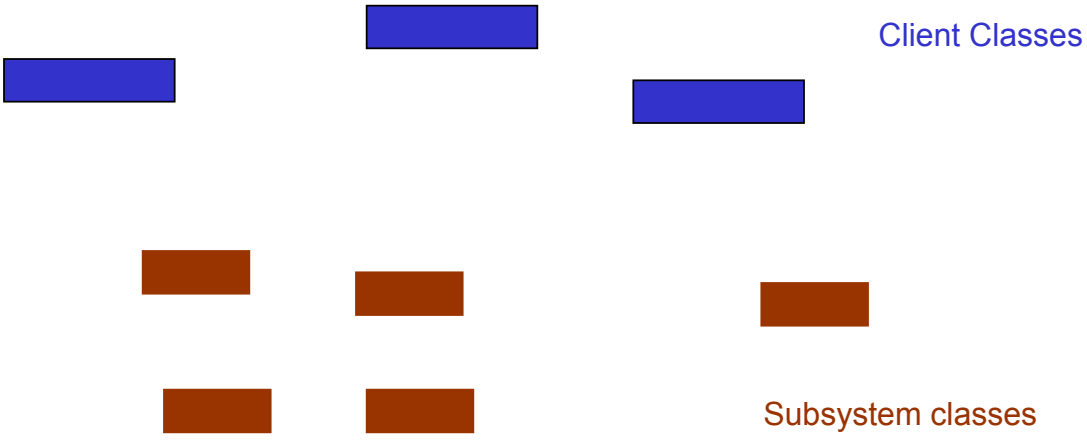
Façade Pattern: Problem

Object Oriented Software Engineering

Façade Pattern: Problem

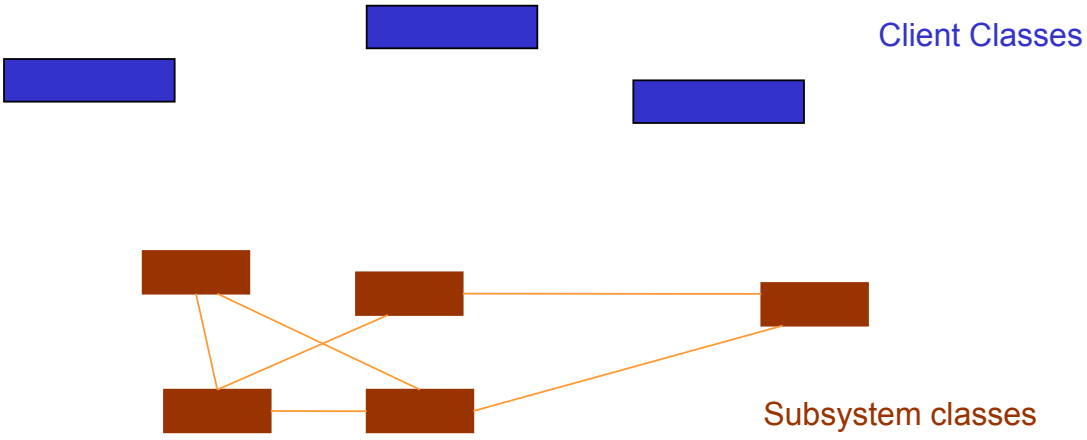
January 02, 2009 336

Façade Pattern: Problem

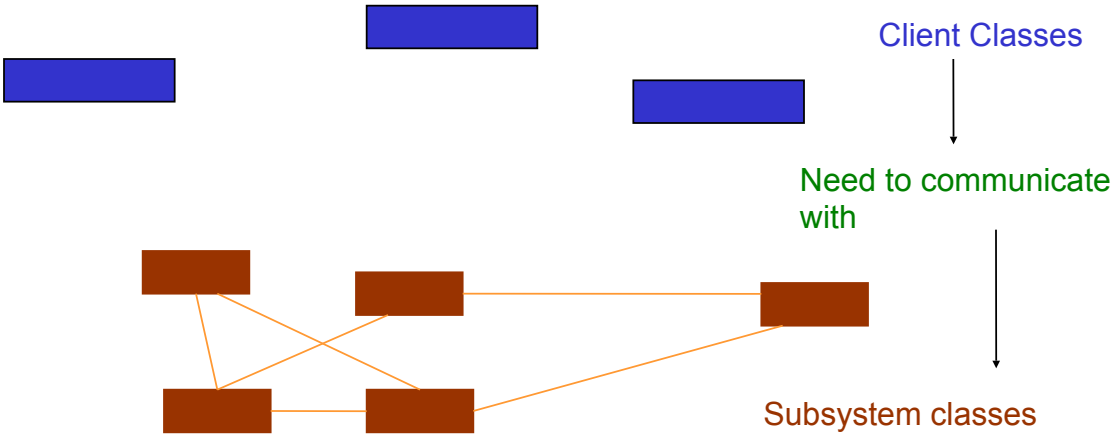


Object Oriented Software Engineering

Façade Pattern: Problem

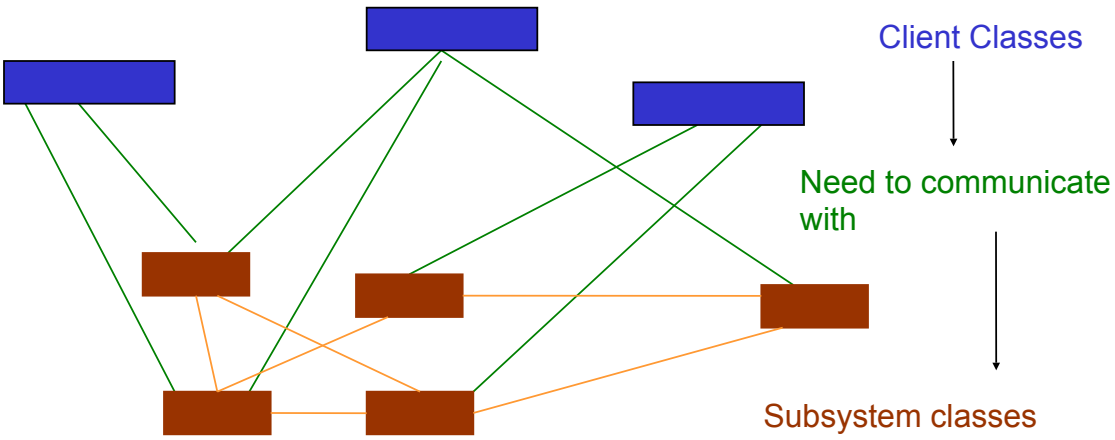


Façade Pattern: Problem

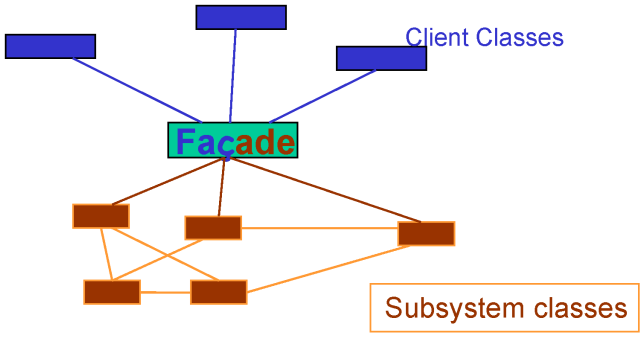


Object Oriented Software Engineering

Façade Pattern: Problem



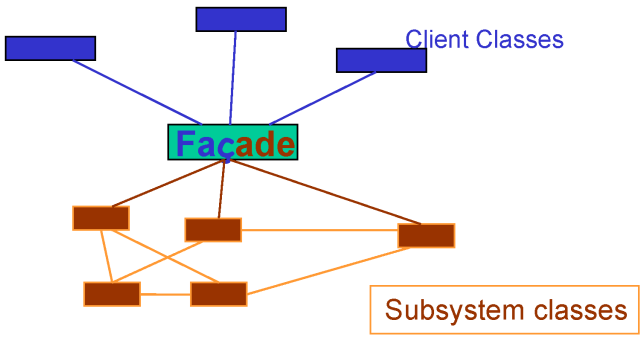
Façade Pattern: Why and What?



- 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.

Object Oriented Software Engineering

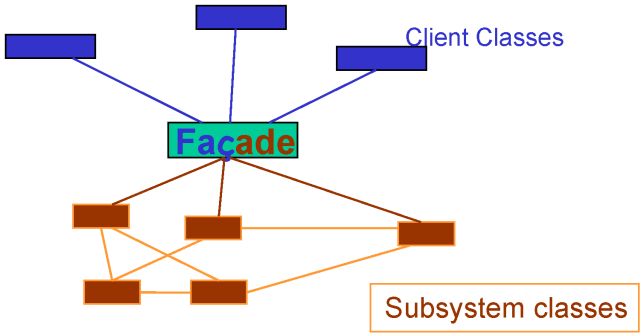
Façade Pattern: Why and What?



- Subsystems often get complex as they evolve.

- 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.

Façade 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.

Participants & Communication

Participants & Communication

- Participants: Façade and subsystem classes

Object Oriented Software Engineering

Participants & Communication

- Participants: Façade and subsystem classes

- Clients communicate with subsystem classes by sending requests to façade.

Participants & Communication

- Participants: Façade and subsystem classes
- Clients communicate with subsystem classes by sending requests to façade.
- Façade forwards requests to the appropriate subsystem classes.

Object Oriented Software Engineering

Participants & 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.

Benefits

Benefits

Object Oriented Software Engineering

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

Benefits

- Shields clients from subsystem classes; reduces the number of objects that clients deal with.
- Promotes weak coupling between subsystem and its clients.

Object Oriented Software Engineering

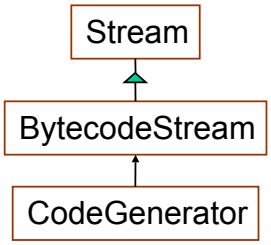
Benefits

- 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

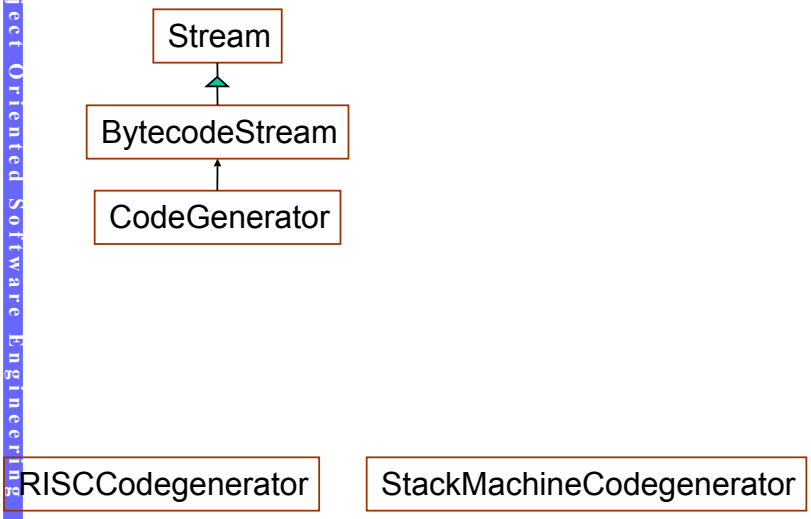
Example: A compiler

Object Oriented Software Engineering

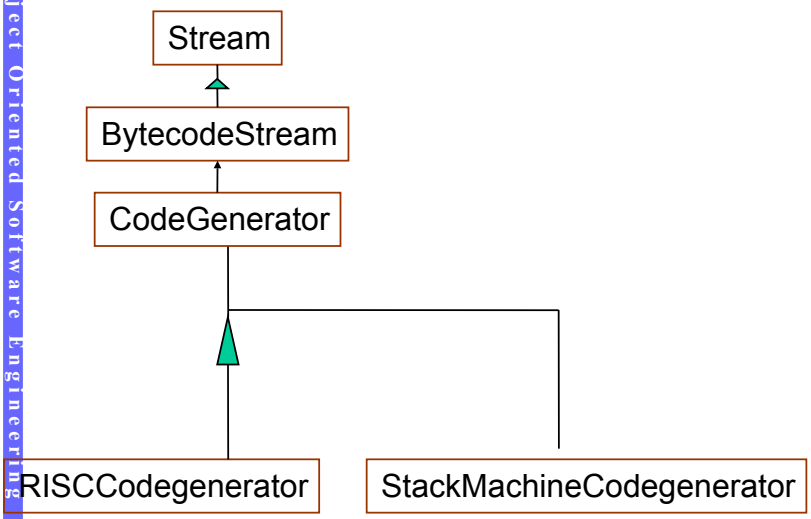


January 02, 2009 340

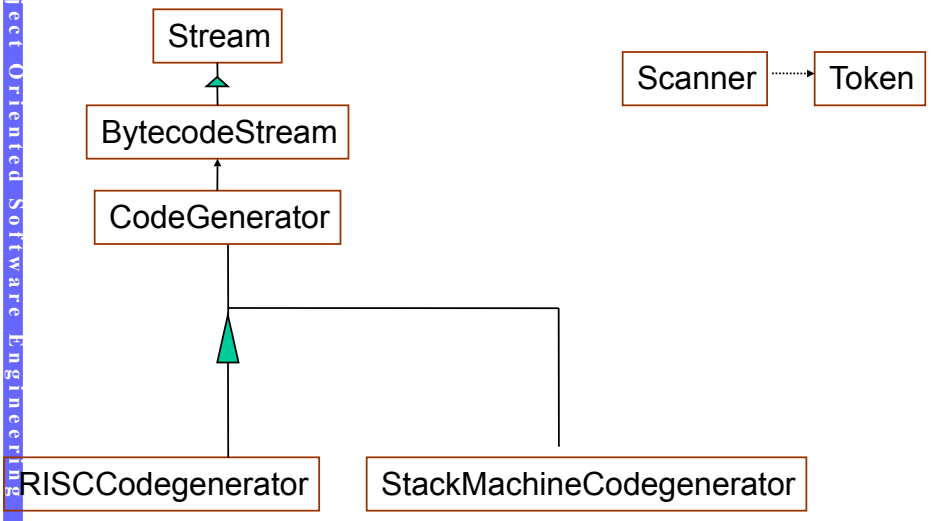
Example: A compiler



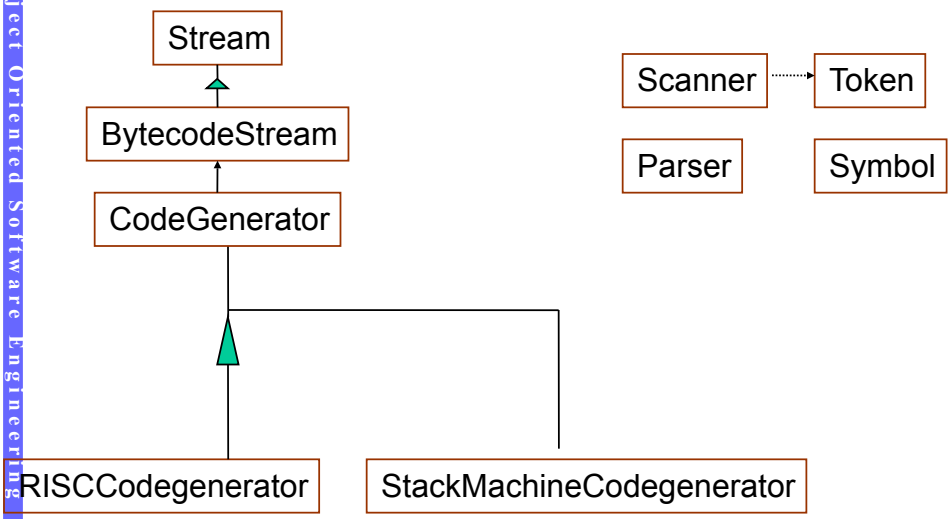
Example: A compiler



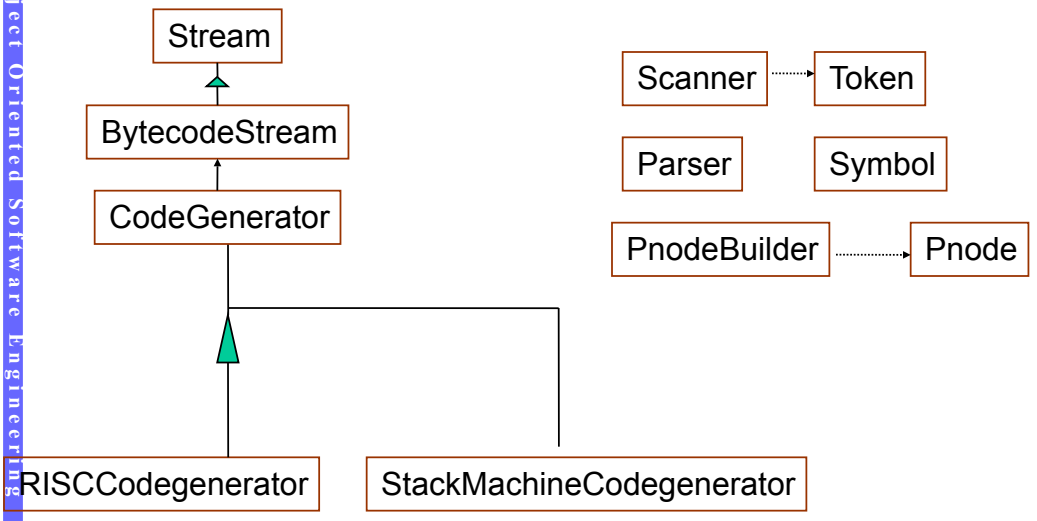
Example: A compiler



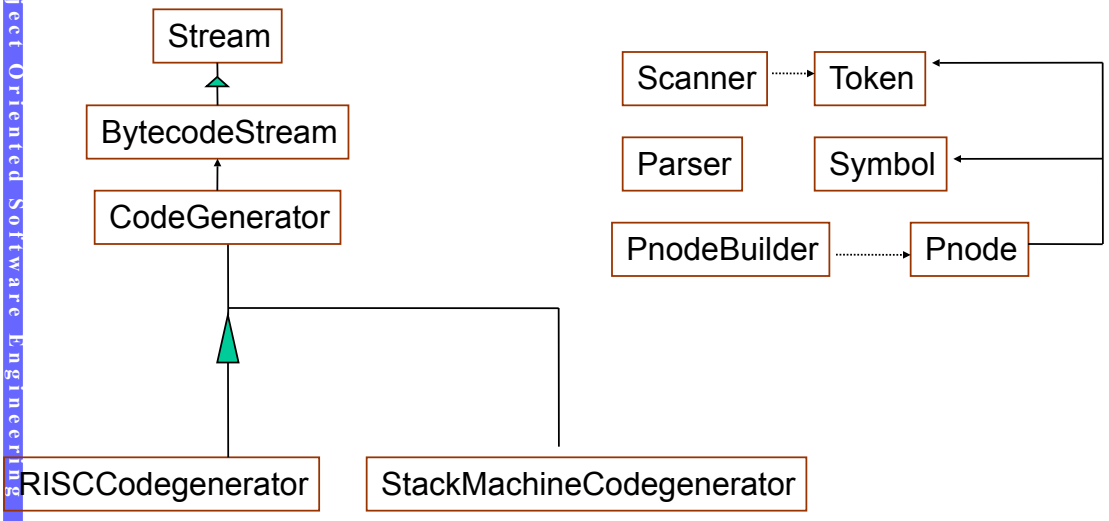
Example: A compiler



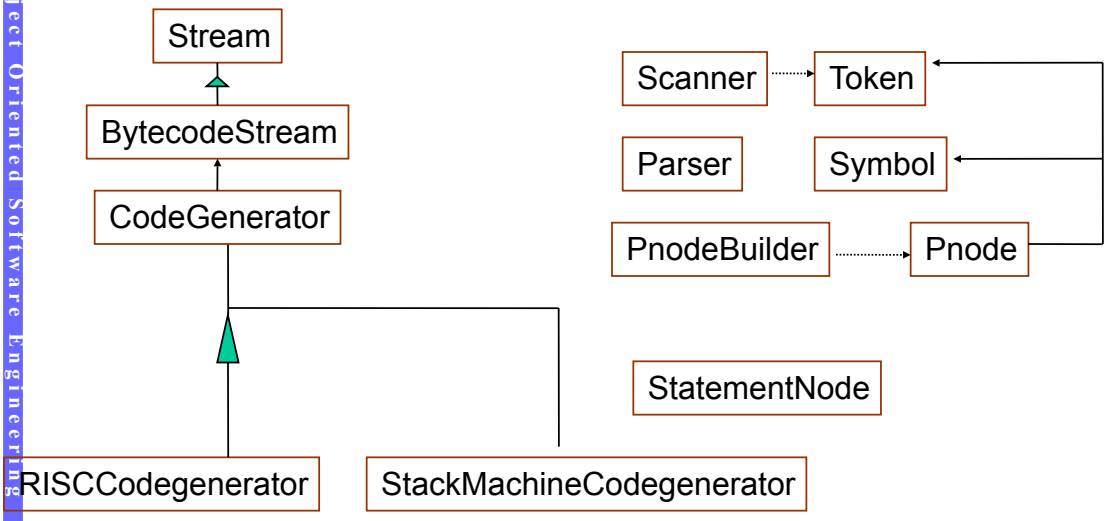
Example: A compiler



Example: A compiler

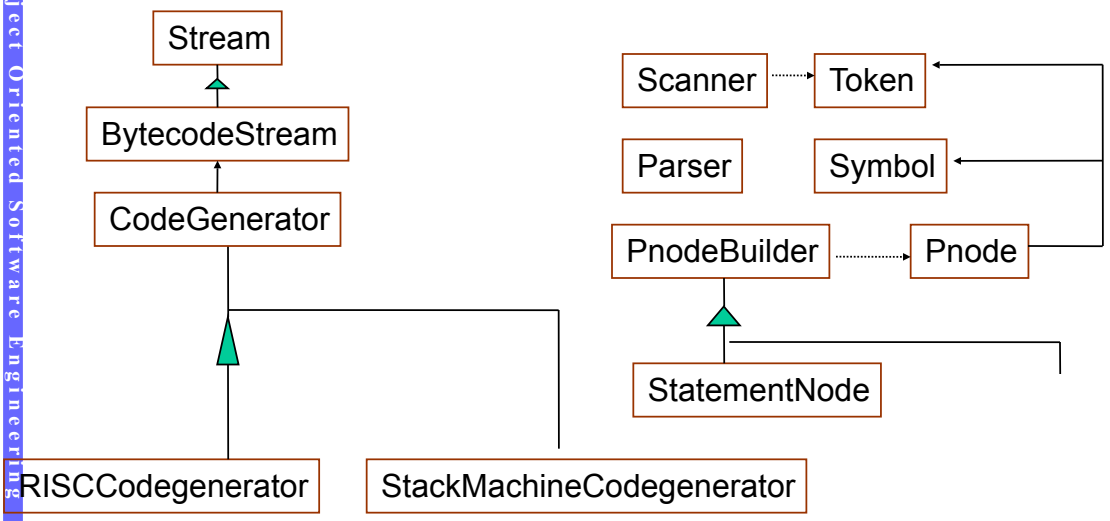


Example: A compiler

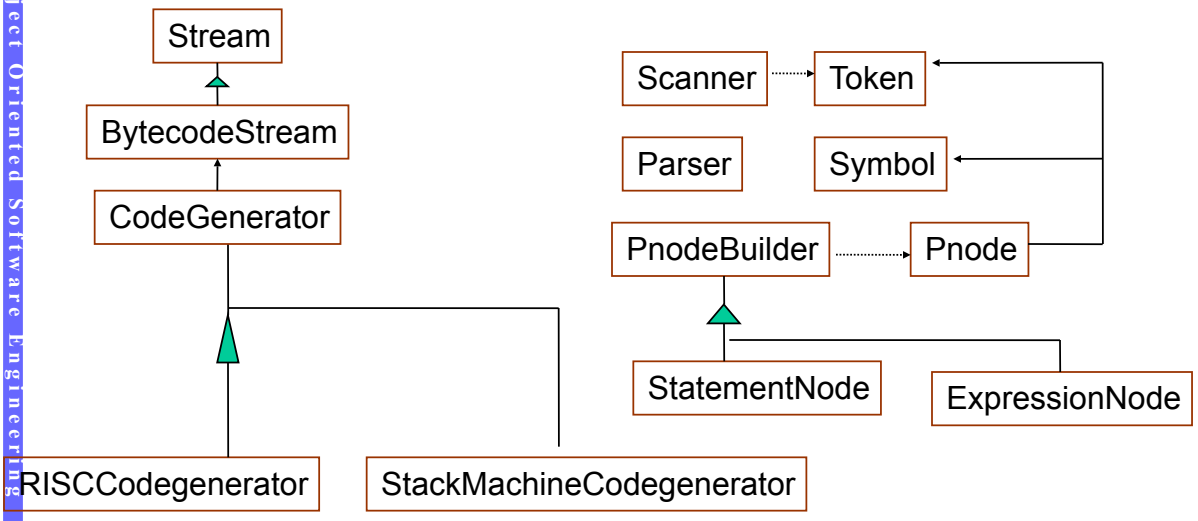


Example: A compiler

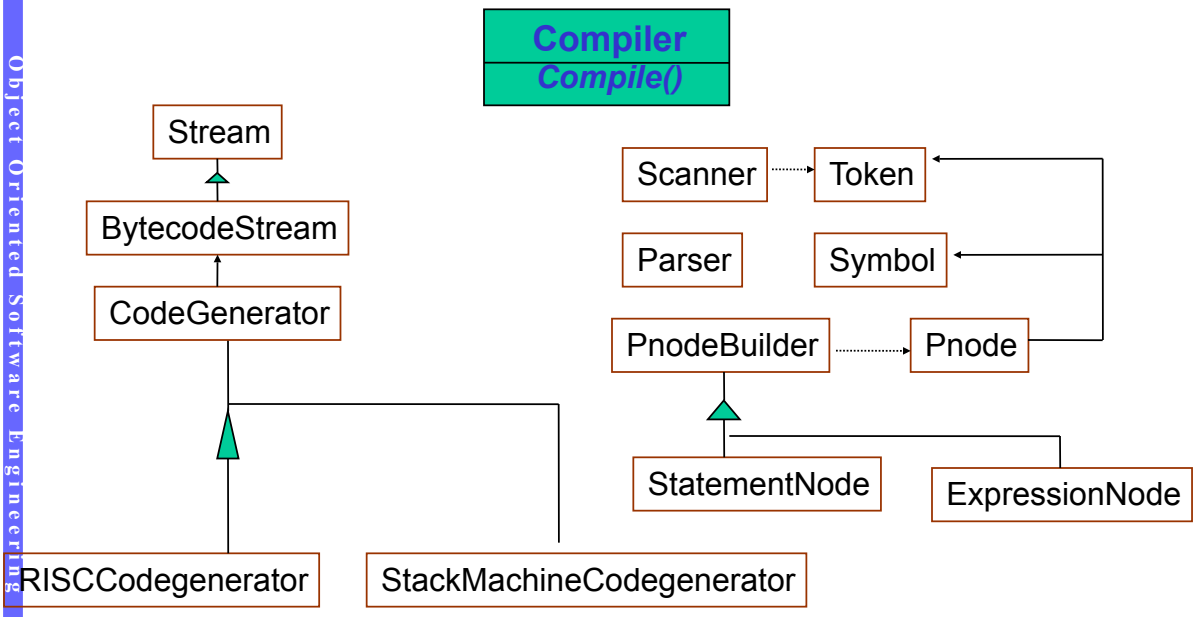
Object Oriented Software Engineering



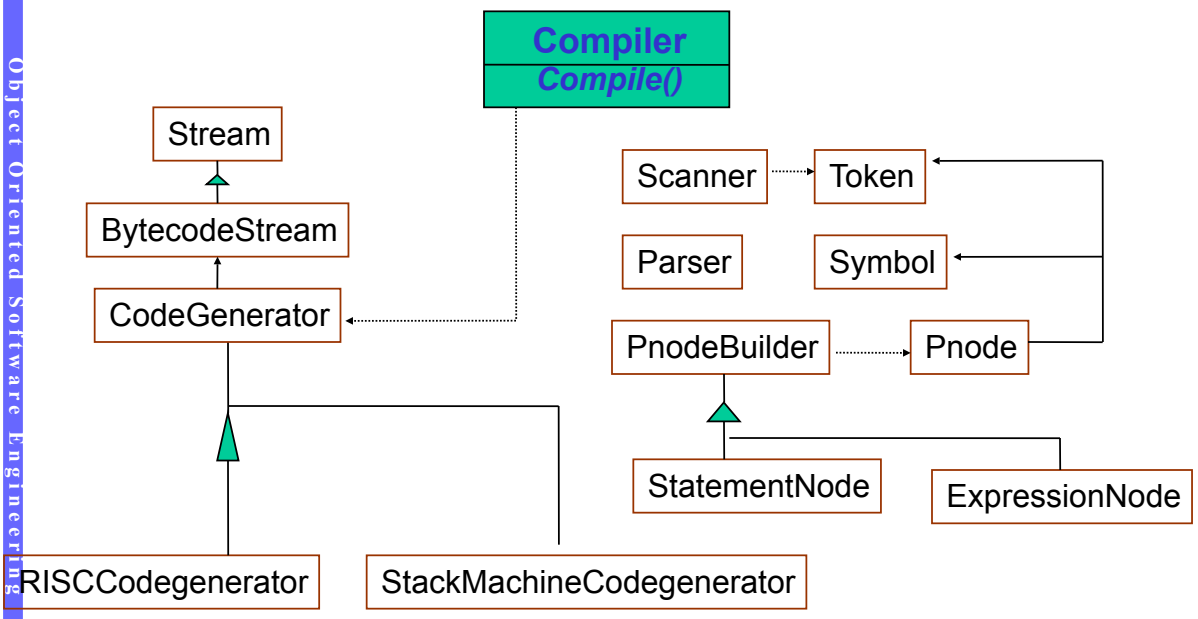
Example: A compiler



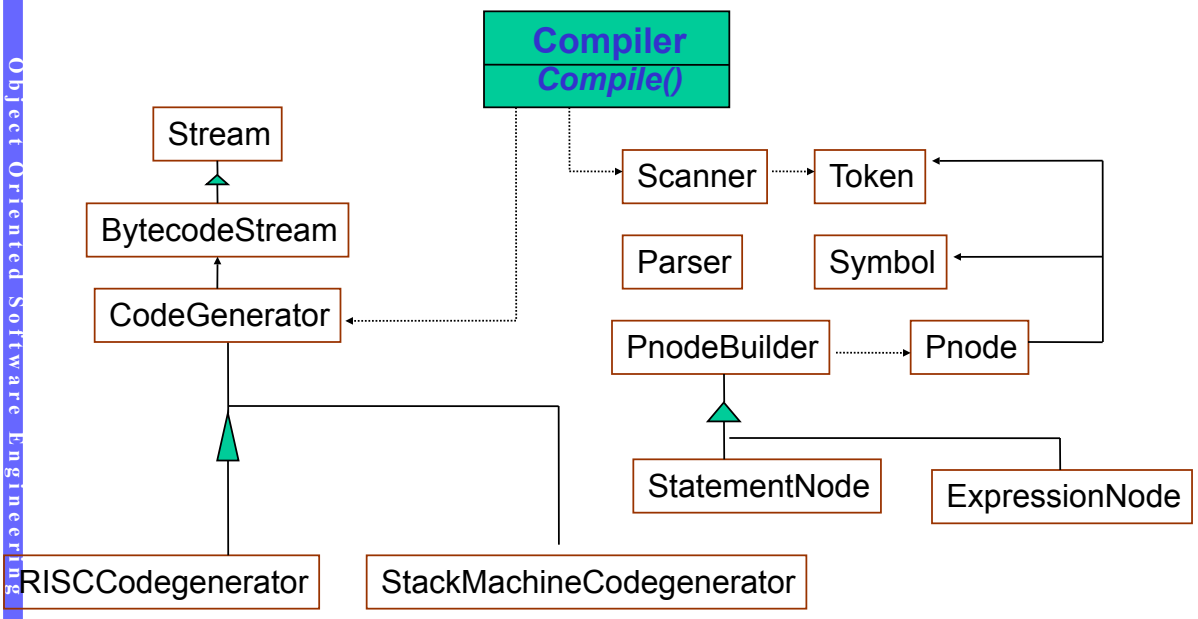
Example: A compiler



Example: A compiler

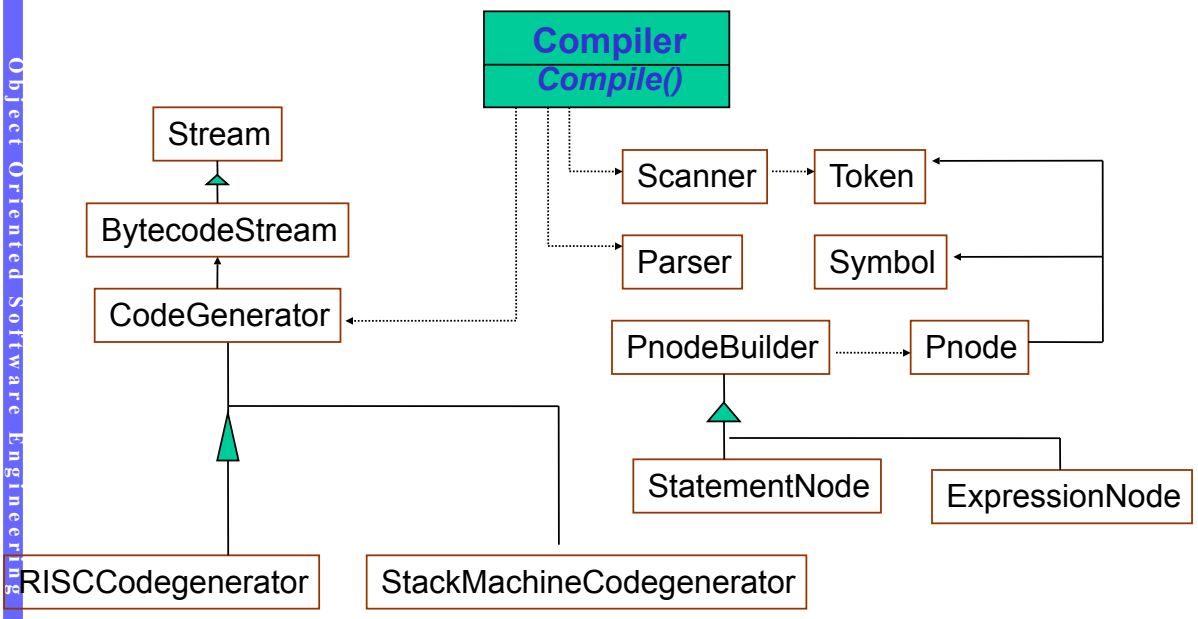


Example: A compiler

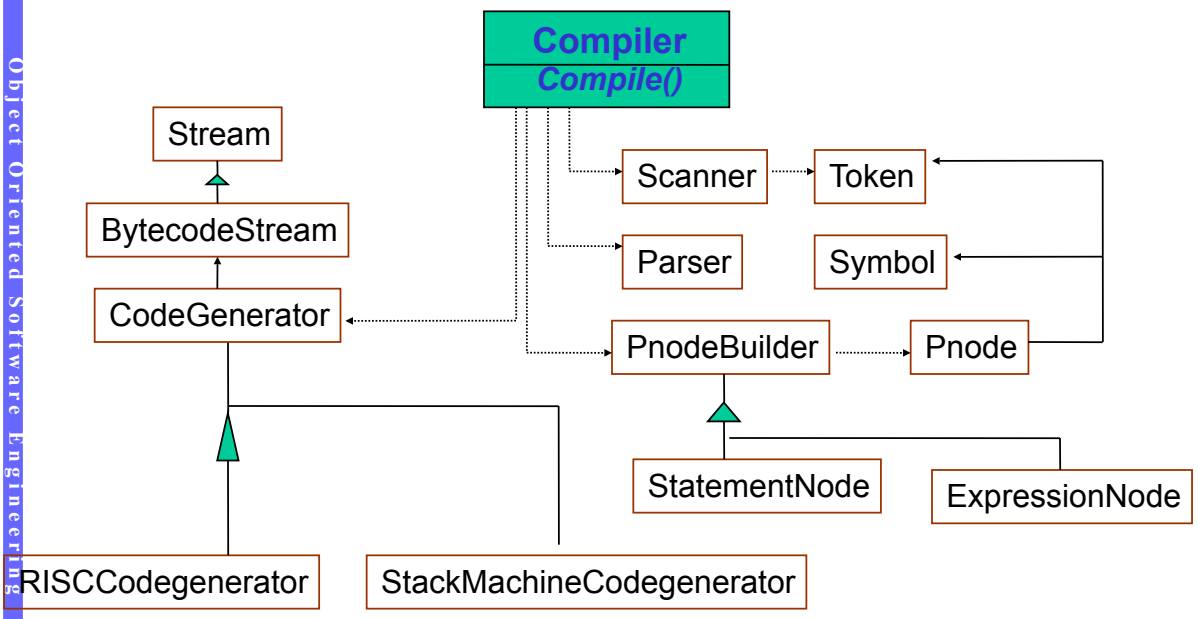


Object Oriented Software Engineering

Example: A compiler

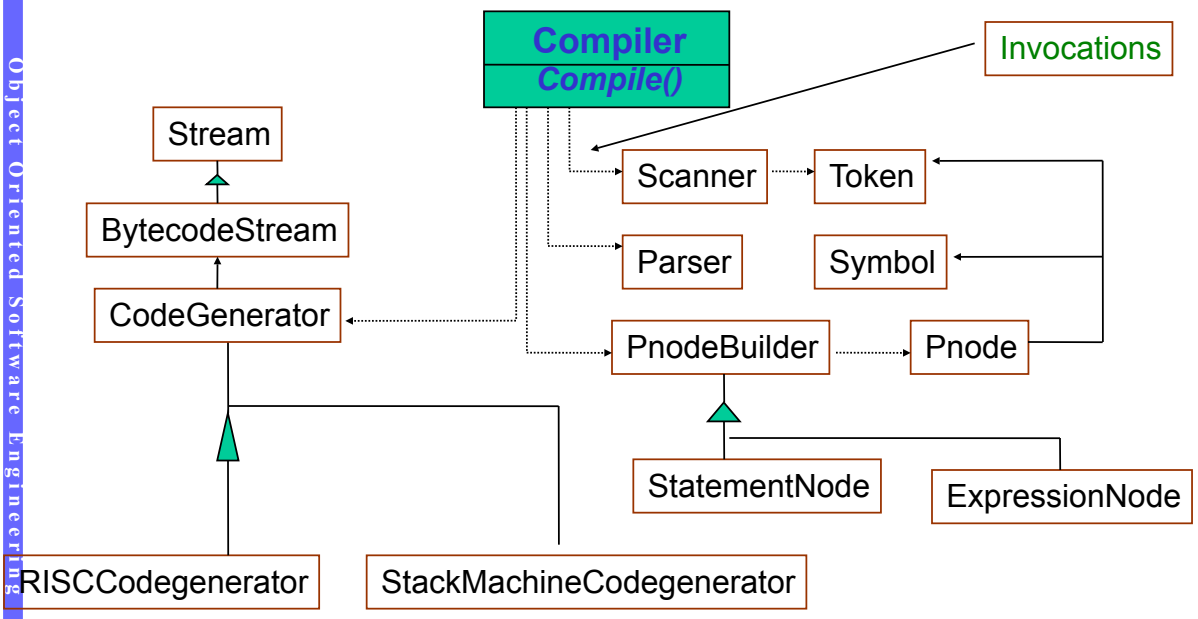


Example: A compiler



Object Oriented Software Engineering

Example: A compiler



Façade Pattern: Code

Façade Pattern: Code

```
class Scanner {
```


Object Oriented Software Engineering

Façade Pattern: Code

```
class Scanner {

   private Stream input_;
```

January 02, 2009 341

Façade Pattern: Code

```
class Scanner {

   private Stream input_;

   public Scanner(Stream);
```


Object Oriented Software Engineering

Façade Pattern: Code

```
class Scanner {

   private Stream input_;

   public Scanner(Stream);
   public Token scan();
```

Façade Pattern: Code

```
class Scanner {
                // Takes a stream of characters and produces a stream of tokens.
   private Stream input_;

   public Scanner(Stream);
   public Token scan();
```


Façade Pattern: Code

```
class Parser {


       public Parser();
       public void parse(Scanner, PNodeBuilder);
```

Façade Pattern: Code


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

        public Parser();
        public void parse(Scanner, PNodeBuilder);
```


Façade Pattern: Code

Object Oriented Software Engineering

Façade Pattern: Code

```
class PNodeBuilder {
```


Façade Pattern: Code

```
class PNodeBuilder {
```

Object Oriented Software Engineering

```
   public PNodeBuilder();
```

January 02, 2009 343

Façade Pattern: Code

```
class PNodeBuilder {

   public PNodeBuilder();

   Pnode makeVariable(String name);
```


Object Oriented Software Engineering

Façade Pattern: Code

```
class PNodeBuilder {

   public PNodeBuilder();

   Pnode makeVariable(String name);


   Pnode makeAssignment(Pnode variable, Pnode exp);
```

Façade Pattern: Code

```
class PNodeBuilder {

   public PNodeBuilder();

   Pnode makeVariable(String name);


   Pnode makeAssignment(Pnode variable, Pnode exp);


   private PNode node_;
```


Object Oriented Software Engineering

Façade Pattern: Code

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

   Pnode makeVariable(String name);


   Pnode makeAssignment(Pnode variable, Pnode exp);


   private PNode node_;
```

January 02, 2009 343

Façade Pattern: Code

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

   Pnode makeVariable(String name);
                                    // Node for a variable.


   Pnode makeAssignment(Pnode variable, Pnode exp);



   private PNode node_;
```


Façade Pattern: Code

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

   Pnode makeVariable(String name);
                                    // Node for a variable.


   Pnode makeAssignment(Pnode variable, Pnode exp);

                                                  // Node for an assignment.

   private PNode node_;
```

Object Oriented Software Engineering

January 02, 2009 343

Façade Pattern: Code

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

   Pnode makeVariable(String name);
                                    // Node for a variable.


   Pnode makeAssignment(Pnode variable, Pnode exp);

                                                  // Node for an assignment.

   private PNode node_;

                                                // Similarly...more nodes.
```


Façade Pattern: Code

```
abstract class PNode {

        protected PNode();


        public int getSourceLine();
        public int getSourceIndex();

        public void add(Pnode);
        public void remove(Pnode);


        public traverse(CodeGen);

        private PNode node_;
```

Façade Pattern: Code

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

                    protected PNode();


                    public int getSourceLine();
                    public int getSourceIndex();

                    public void add(Pnode);
                    public void remove(Pnode);


                    public traverse(CodeGen);

                    private PNode node_;
```


Façade Pattern: Code

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

                    protected PNode();

                     // Manipulate program node.

                    public int getSourceLine();
                    public int getSourceIndex();

                    public void add(Pnode);
                    public void remove(Pnode);


                    public traverse(CodeGen);

                    private PNode node_;
```

Object Oriented Software Engineering

January 02, 2009 344

Façade Pattern: Code


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

                    protected PNode();

                     // Manipulate program node.

                    public int getSourceLine();
                    public int getSourceIndex();

                    // Manipulate child node.
                    public void add(Pnode);
                    public void remove(Pnode);

                      // ….
                    public traverse(CodeGen);

                    private PNode node_;
```


Façade Pattern: Code


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

                    protected PNode();

                     // Manipulate program node.

                    public int getSourceLine();
                    public int getSourceIndex();

                    // Manipulate child node.
                    public void add(Pnode);
                    public void remove(Pnode);

                      // ….
                    public traverse(CodeGen); // Traverse tree to generate code.

                    private PNode node_;
```

Façade Pattern: Code


```
Abstract class CodeGen {

                public void
visit(StatementNode n);
                public void
visit(StatementNode n);


                private ByteCodeStream
output_;
```


Façade Pattern: Code

Object Oriented Software Engineering

```
Abstract class CodeGen {

                public void          // Generate bytecode.
visit(StatementNode n);
                public void
visit(StatementNode n);


                private ByteCodeStream
output_;
```

January 02, 2009 345

Façade Pattern: Code

```
Abstract class CodeGen {

                public void          // Generate bytecode.
visit(StatementNode n);
                public void
visit(StatementNode n);
                          // Manipulate program node.

                private ByteCodeStream
output_;
                               // ….
```


Façade Pattern: Code

```
class ExpressionNode {
       public void traverse(CodeGen g)  {
              cg.visit(this);
              ListIterator it =  children.iterator();
              while (it.hasNext())
              ((PNode)it.next()).traverse(g);
 }
```

Façade Pattern: Code


```
class Compiler {

public Compiler();

public void compile(Stream input, ByteCodeStream output) {

                Scanner scanner = new Scanner(input);
                PNodeBuilder builder = new PNodeBuilder();
                parser.Parse(scanner, builder);
                RISCCodeGen genrator = new RISCCodeGen(output);
                Pnode parseTree = builder.getRootNode();
                parseTree.traverse(generator);
```


Façade Pattern: Code

Object Oriented Software Engineering

```
class Compiler {        // Façade. Offers a simple interface to compile and
                        // Generate code.

public Compiler();

public void compile(Stream input, ByteCodeStream output) {

                Scanner scanner = new Scanner(input);
                PNodeBuilder builder = new PNodeBuilder();
                parser.Parse(scanner, builder);
                RISCCodeGen genrator = new RISCCodeGen(output);
                Pnode parseTree = builder.getRootNode();
                parseTree.traverse(generator);
```

Façade Pattern: Code

```
class Compiler {        // Façade. Offers a simple interface to compile and
                        // Generate code.

public Compiler();
```

Could also take a CodeGenerator Parameter for increased generality.

```
public void compile(Stream input, ByteCodeStream output) {

        Scanner scanner = new Scanner(input);
        PNodeBuilder builder = new PNodeBuilder();
        parser.Parse(scanner, builder);
        RISCCodeGen genrator = new RISCCodeGen(output);
        Pnode parseTree = builder.getRootNode();
        parseTree.traverse(generator);
```


Singleton

Singleton

- Used to ensure that a class has only one instance. For example, one printer spooler object, one file system, one window manager, etc.

Object Oriented Software Engineering

Singleton

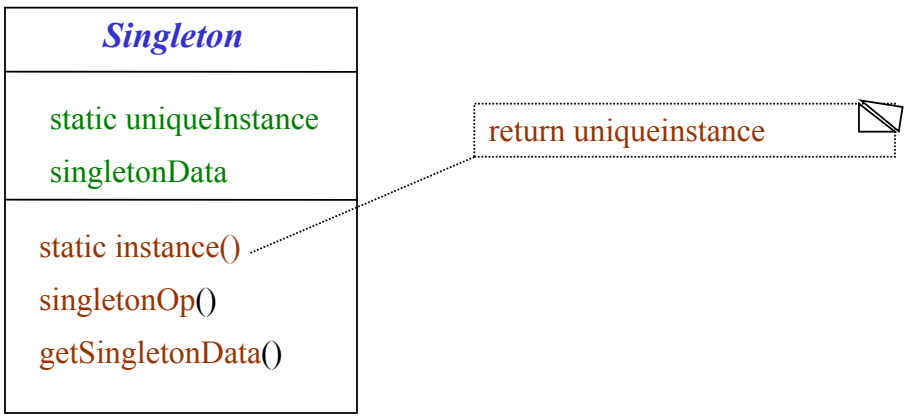
- Used to ensure that a class has only one instance. For example, one printer spooler object, one file system, one window manager, etc.

- One may use a global variable to access an object but it does not prevent one from creating more than one instance.

Singleton

- Used to ensure that a class has only one instance. For example, one printer spooler object, one file system, one window manager, etc.
- One may use a global variable to access an object but it does not prevent one from creating more than one instance.
- Instead the class itself is made responsible for keeping track of its instance. It can thus ensure that no more than one instance is created. This is the singleton pattern.

Singleton Structure



Singleton Code

```
public class Singleton {
   private static Singleton instance_;
   private Data data_;
   private Singleton() { …init data… }
   public Data getData() { return data_; }

   public synchronized static Singleton instance() {
       if (instance_ == null) instance_ = new
   Singleton();
       return instance_;
   }
```


Object Oriented Software Engineering

Singleton Code

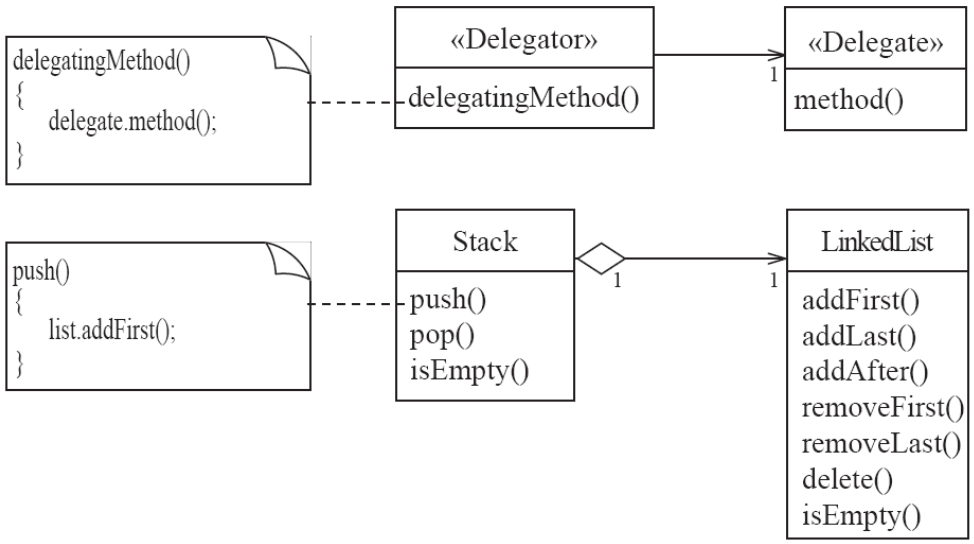
```
public class Singleton {
   private static Singleton instance_;
   private Data data_;     // Only one instance can ever be created.
   private Singleton() { …init data… }
   public Data getData() { return data_; }

   public synchronized static Singleton instance() {
       if (instance_ == null) instance_ = new
   Singleton();
       return instance_;
   }
```

6.7 The Delegation Pattern

- ***Context***:
  - —You are designing a method in a class
  - —You realize that another class has a method which provides the required service
  - —Inheritance is not appropriate
    - E.g. because the isa rule does not apply
- ***Problem***:
  - —How can you most effectively make use of a method that already exists in the other class?
- ***Forces***:
  - —You want to minimize development cost by reusing methods

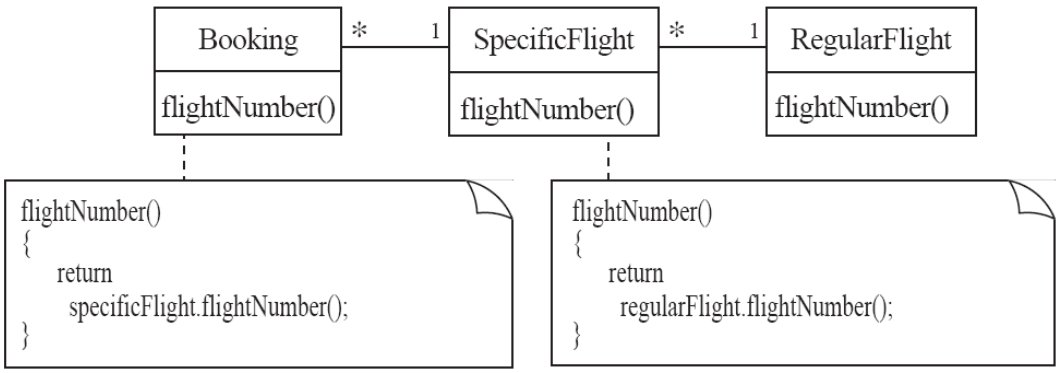
Delegation

- ***Solution:***



Delegation

Example:



Delegation

Antipatterns

- Overuse generalization and *inherit* the method that is to be reused
- Instead of creating a *single* method in the «Delegator» that does nothing other than call a method in the «Delegate
  - —consider having many different methods in the «Delegator» call the delegate's method
- Access non-neighboring classes

```
return specificFlight.regularFlight.flightNumber();

return getRegularFlight().flightNumber();
```

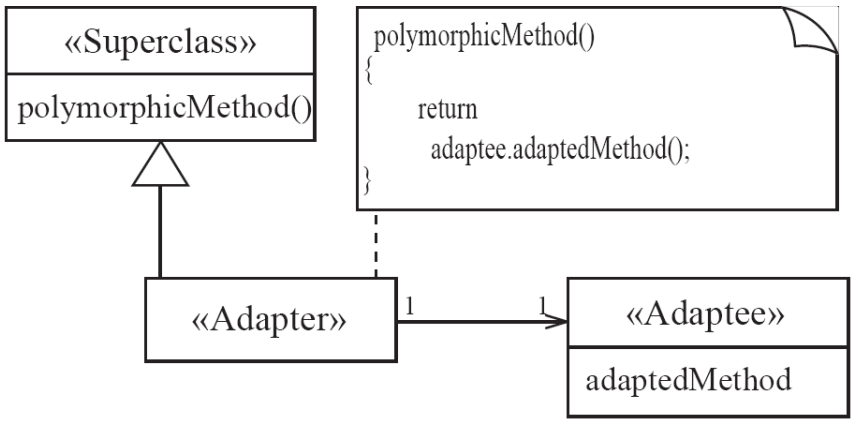
6.8 The Adapter Pattern

- ***Context***:
  - —You are building an inheritance hierarchy and want to incorporate it into an existing class.
  - —The reused class is also often already part of its own inheritance hierarchy.
- ***Problem***:
  - —How to obtain the power of polymorphism when reusing a class whose methods
    - have the same function
    - but *not* the same signature

    as the other methods in the hierarchy?
- ***Forces***:
  - —You do not have access to multiple inheritance or you do not want to use it.

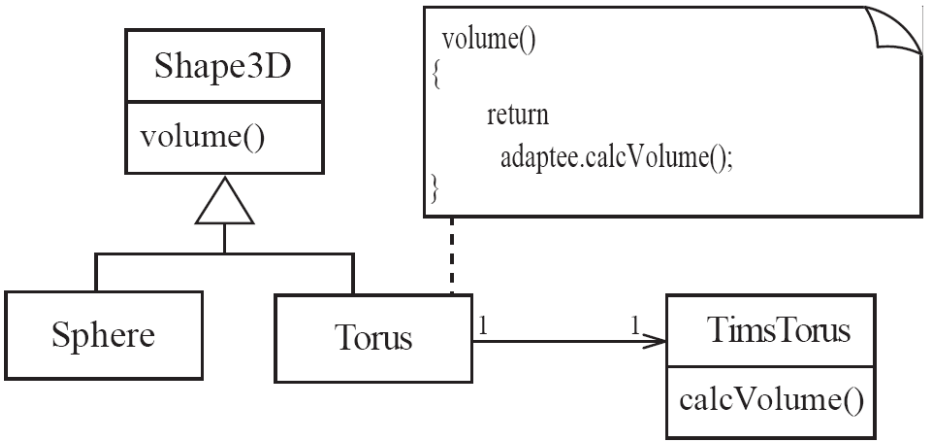
Adapter

- ***Solution:***



Adapter

Example:



6.10 The Immutable Pattern

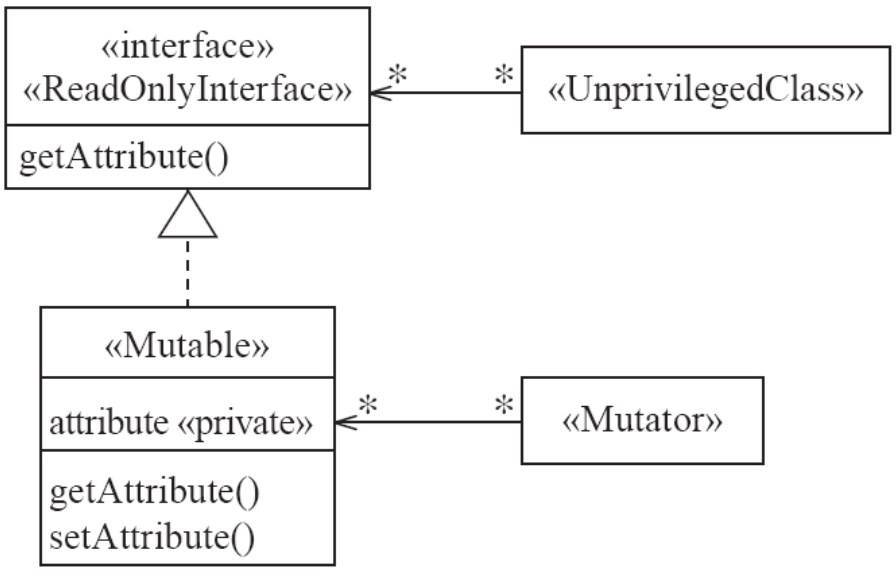
- ***Context***:
  - —An immutable object is an object that has a state that never changes after creation
- ***Problem***:
  - —How do you create a class whose instances are immutable?
- ***Forces***:
  - —There must be no loopholes that would allow 'illegal' modification of an immutable object
- ***Solution***:
  - —Ensure that the constructor of the immutable class is the *only* place where the values of instance variables are set or modified.
  - —Instance methods which access properties must not have side effects.
  - —If a method that would otherwise modify an instance variable is required, then it has to return a *new* instance of the class.

6.11 The Read-only Interface Pattern

- ***Context***:
  - —You sometimes want certain privileged classes to be able to modify attributes of objects that are otherwise immutable
- ***Problem***:
  - —How do you create a situation where some classes see a class as read-only whereas others are able to make modifications?
- ***Forces***:
  - —Restricting access by using the **`public`**, **`protected`** and **`private`** keywords is not adequately selective.
  - —Making access **`public`** makes it public for both reading and writing

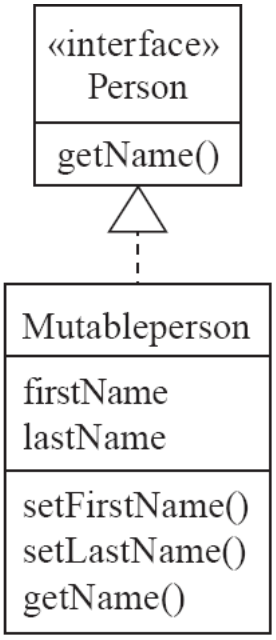
Read-only Interface

- ***Solution:***



Read-only Interface

Example:



Read-only Interface

Antipatterns:

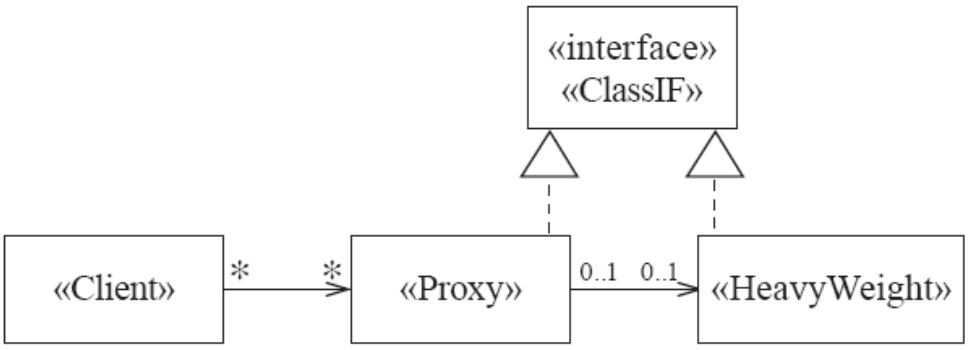
- Make the read-only class a *subclass* of the «Mutable» class
- Override all methods that modify properties
  - —such that they throw an exception

6.12 The Proxy Pattern

- ***Context***:
  - —Often, it is time-consuming and complicated to create instances of a class (*heavyweight* classes).
  - —There is a time delay and a complex mechanism involved in creating the object in memory
- ***Problem***:
  - —How to reduce the need to create instances of a heavyweight class?
- ***Forces***:
  - —We want all the objects in a domain model to be available for programs to use when they execute a system's various responsibilities.
  - —It is also important for many objects to persist from run to run of the same program

Proxy

- ***Solution:***



Proxy

Examples: