Software patterns

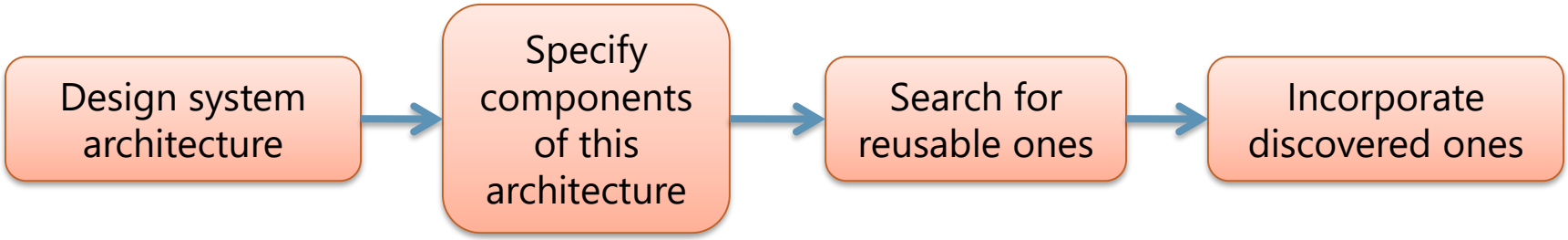
Dr. Vu Thi Huong Giang

Introduction

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems
- Software engineering has been more focused on original development
- To achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on systematic software reuse

An opportunistic reuse process

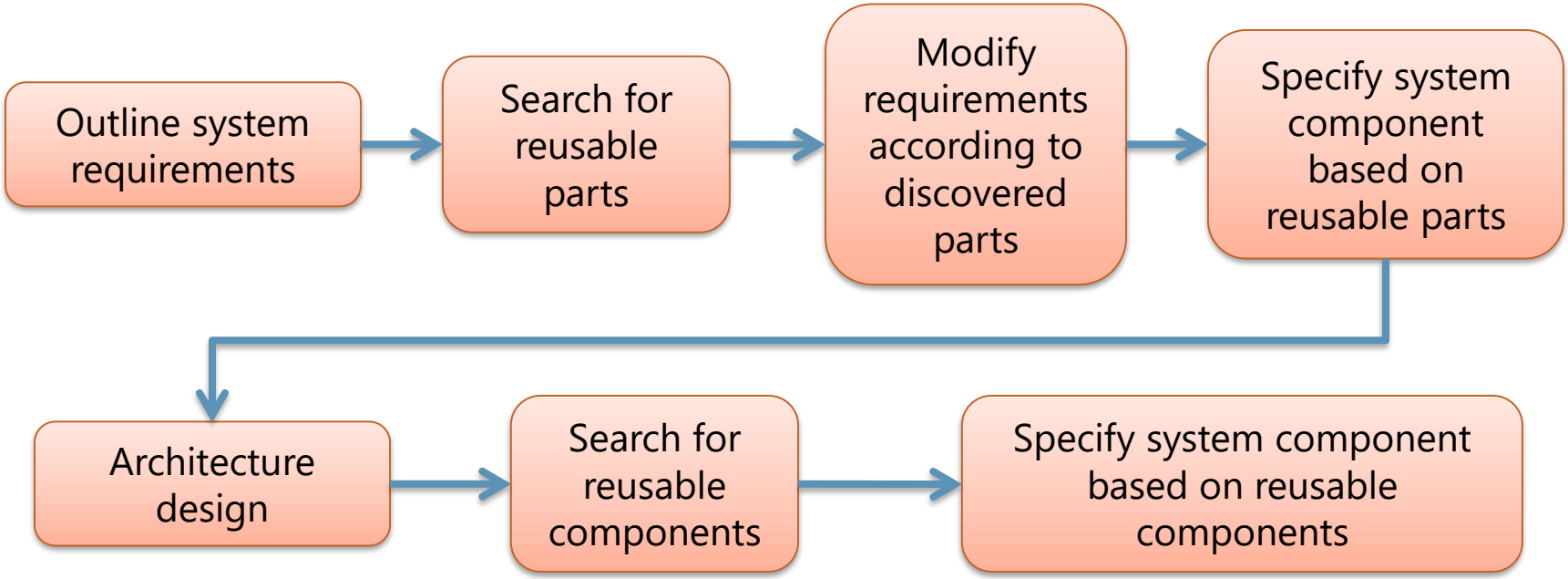
Application-driven



Problem : Specified components may be hard to find

Development with reuse

Reuse-driven



Problem : Compromised development → available components may not be the best design and implementation solution

Types of software patterns

- Design patterns : software design
  - architectural (systems design)
  - design (micro-architectures)
  - idioms (low level)
- Analysis patterns : recurring & reusable analysis models
- Organization patterns : structure of organizations/projects
- Process patterns : software process design
- Domain-specific patterns: SOA patterns, cloud patterns

Covered topics

- Reuse-based software engineering overview
- Concept reuse
- Software product lines (application families)
- Design patterns

Objectives

- After this lesson, students will be able to:
  - Recall the profiles of creational, behavioral and structural patterns.
  - Show possible usage of patterns for a specific software.

1. Reuse-based software engineering overview

- Application system reuse
  - The whole of an application system may be reused
    - by incorporating it without change into other systems
    - by developing application families
- Component reuse
  - Components of an application from sub-systems to single objects may be reused
- Function reuse
  - Software components that implement a single well-defined function may be reused

- Application system reuse example
  - COTS (commercial off-the-shelf) reuse: software services such as database management systems, firewalls, web servers, etc.
  - The same application for different environments: device drivers, file systems, network protocols, exception handling, text editor, etc.
- Component reuse example
  - CORBA components
  - OLE (Object Linking and Embedding) components
  - COM (Component Object Model) components
- Function reuse
  - domain-specific libraries of reusable functions: graphics, mathematical libraries, toolkits (e.g. Quickdraw) etc.

1.1. Reuse benefits

Increased dependability	Components exercised in working systems
Reduced process risk	Less uncertainty in development costs
Effective use of specialists	Reuse components instead of people
Standards compliance	Embed standards in reusable components (eg menus in user interface)
Accelerated development	Avoid original development and hence speed-up production

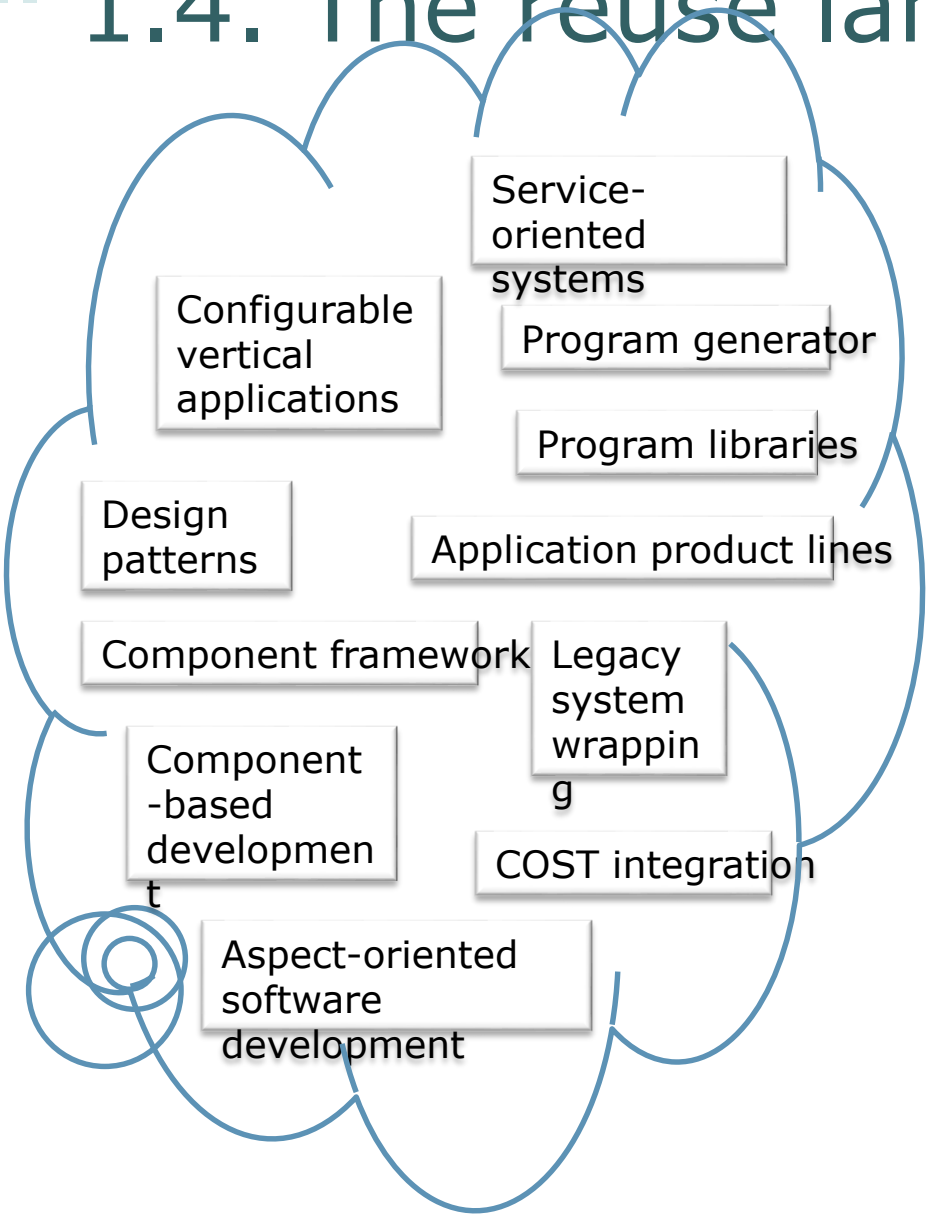
1.2. Requirements for design with reuse

- It must be possible to find appropriate reusable components
- The reuse of the component must be confident that the components will be reliable and will behave as specified
- The components must be documented so that they can be understood and, where appropriate, modified

1.3. Problems of reuse

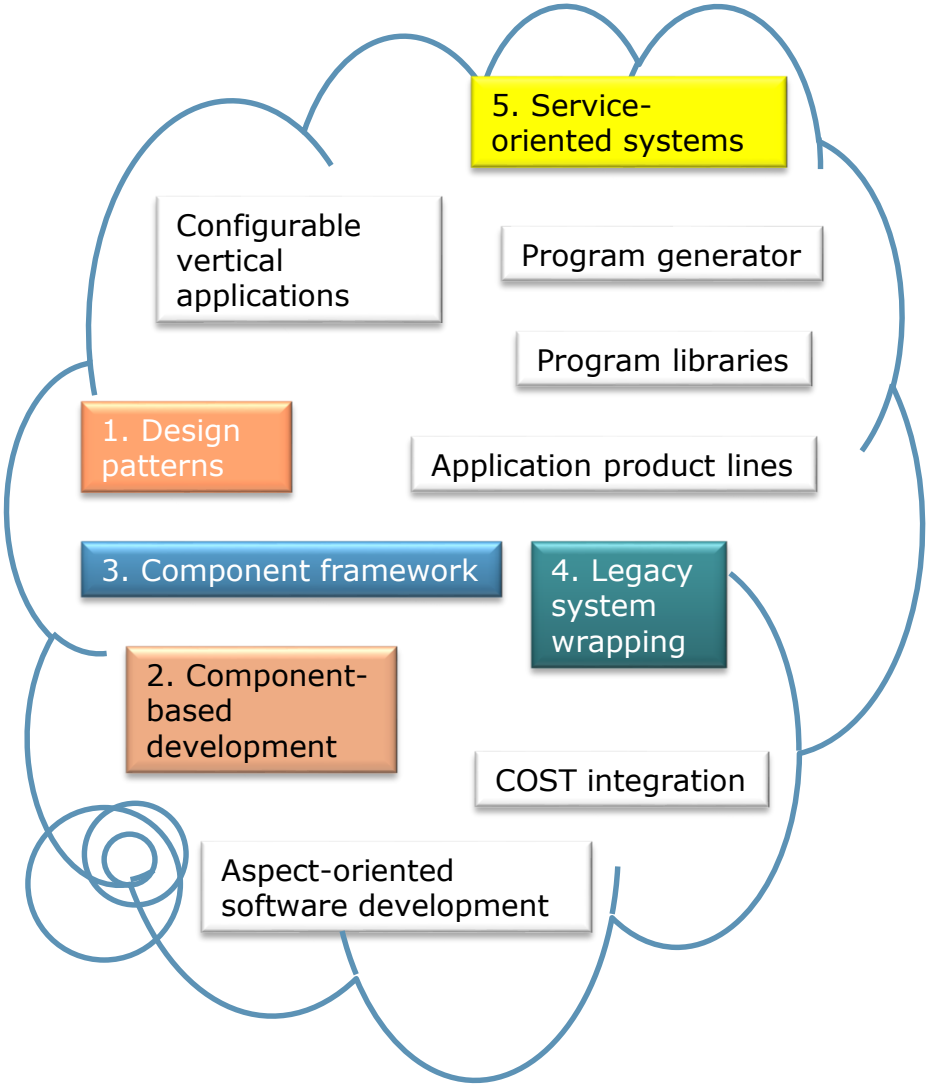
Increased maintenance costs	- Source code of a reused software system or component is not available. - Reused elements of the system may become increasingly incompatible with system changes.
Lack of tool support	No CASE toolsets for supporting development with reuse.
Not-invented-here syndrome	Preference in writing your own component due to lack of trust or challenge
Creating and maintaining a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.
Finding, understanding and adapting reusable components with confidence	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they will make routinely include a component search as part of their normal development process.

1.4. The reuse landscape



- Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used
- Reuse is possible at a range of levels from simple functions to complete application systems
- The reuse landscape covers the range of possible reuse techniques

1.4. The reuse landscape



1. Generic abstractions that occur across applications are represented as design patterns that show abstract and concrete objects and interactions.

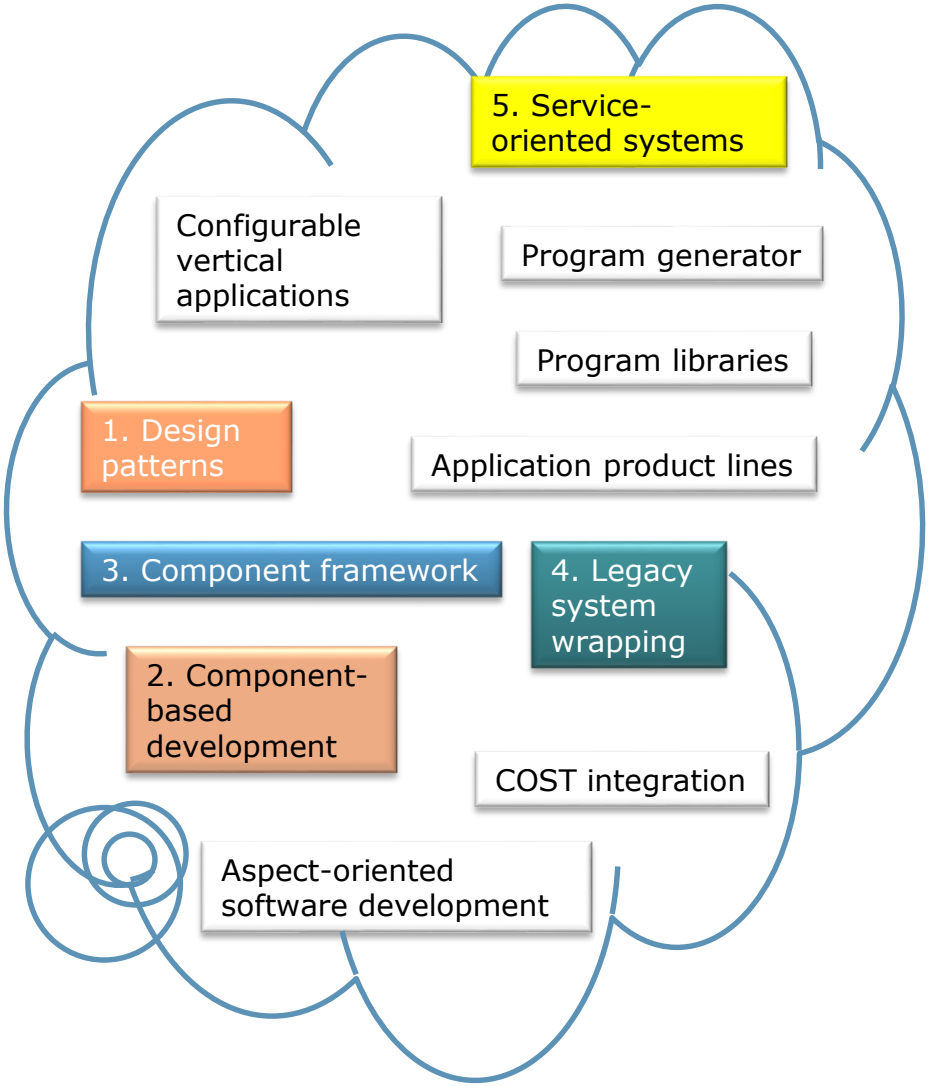
2. Systems are developed by integrating components (collections of objects) that conform to component-model standards.

3. Collections of abstract and concrete classes that can be adapted and extended to create application systems.

4. Defining a set of interfaces and providing access to these legacy systems through these interfaces

5. Systems are developed by linking shared services that may be externally provided.

1.4. The reuse landscape



6. An application type is generalised around a common architecture so that it can be adapted in different ways for different customers.

7. Systems are developed by integrating existing application systems.

8. A generic system is designed so that it can be configured to the needs of specific system customers.

9. Class and function libraries implementing commonly-used abstractions are available for reuse.

10. A generator system embeds knowledge of a particular types of application and can generate systems or system fragments in that domain.

11. Shared components are woven into an application at different places when the program is compiled.

1.5. Reuse planning factors

- The development schedule for the software
- The expected software lifetime
- The background, skills and experience of the development team
- The criticality of the software and its non-functional requirements
- The application domain
- The execution platform for the software

2. Concept reuse

- When you reuse program or design components, you have to follow the design decisions made by the original developer of the component
- This may limit the opportunities for reuse
- However, a more abstract form of reuse is concept reuse when a particular approach is described in an implementation independent way and an implementation is then developed
- The two main approaches to concept reuse are:
  - Design patterns
  - Generator-based reuse (generative programming)

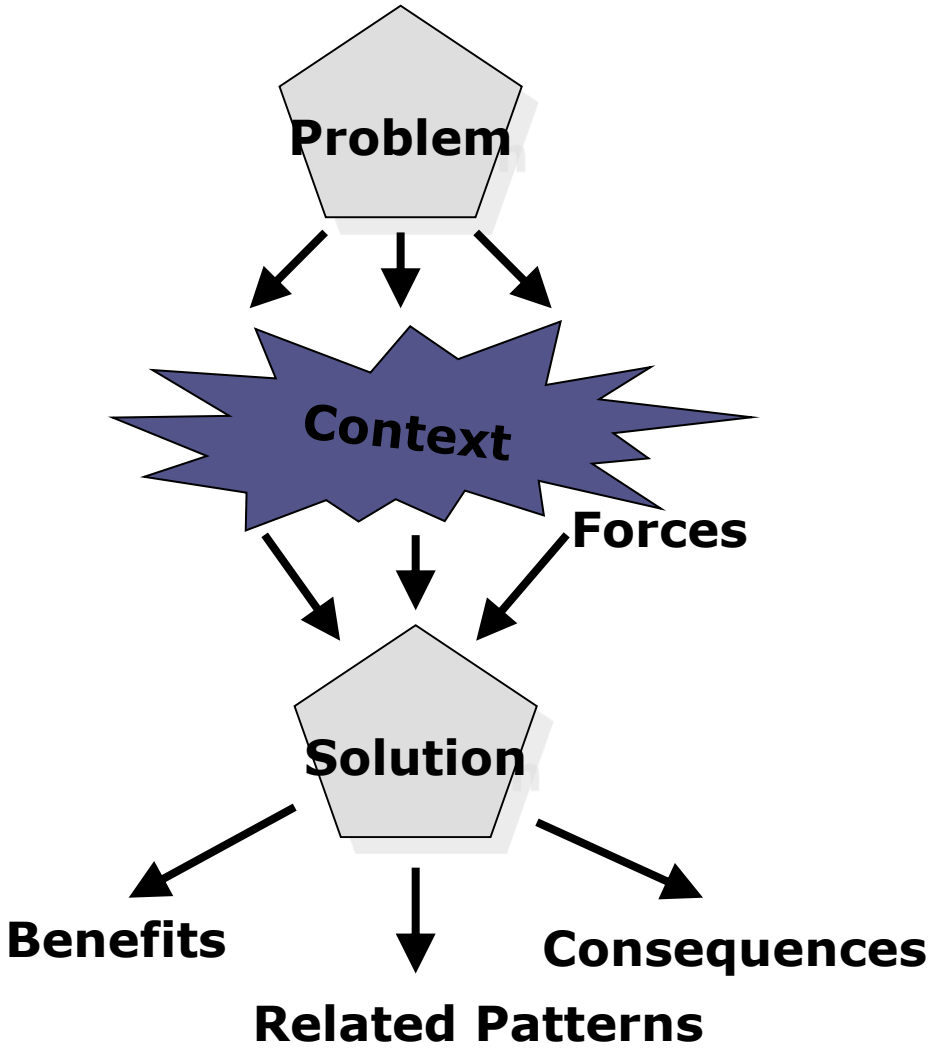
2.1. Design patterns

- A pattern is a description of the problem and the essence of its solution
- It should be sufficiently abstract to be reused in different settings
- Patterns often rely on object characteristics such as inheritance and polymorphism

→ A design pattern is a way of reusing abstract knowledge about a problem and its solution

  - Generally at a "higher level" of abstraction.
  - Not about designs such as linked lists or hash tables.
  - Generally descriptions of communicating objects and classes.

Pattern elements



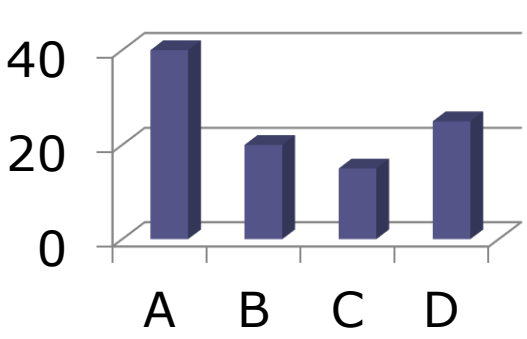
- Name
  - A meaningful pattern identifier
- Problem description
- Solution description
  - Not a concrete design but a template for a design solution that can be instantiated in different ways
- Consequences
  - The results and trade-offs of applying the pattern

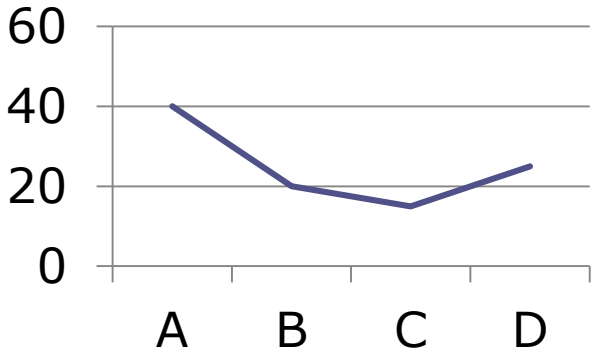
Design pattern catalog

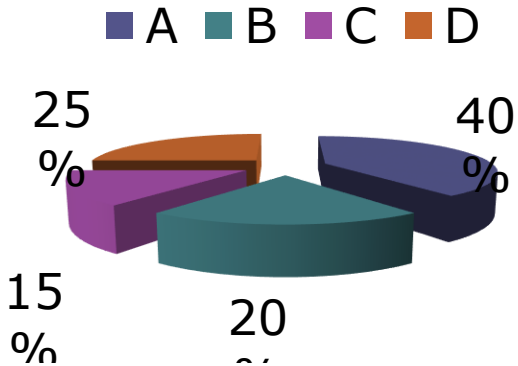
- Patterns are subdivided into small number of broad categories

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	• **Factory Method**	• **Adapter**	• **Interpreter**
	Object	• **Abstract Factory** • **Builder** • **Prototype** • **Singleton**	• **Adapter** • **Bridge** • **Composite** • **Decorator** • **Facade** • **Flyweight** • **Proxy**	• **Chain of Responsibility** • **Command** • **Iterator** • **Mediator** • **Momento** • **Observer** • **State** • **Strategy** • **Visitor**

Example: Multiple displays







Observer 1

Observer 2

Observer 3

Subject

A : 40
B : 20
C : 15
D : 25

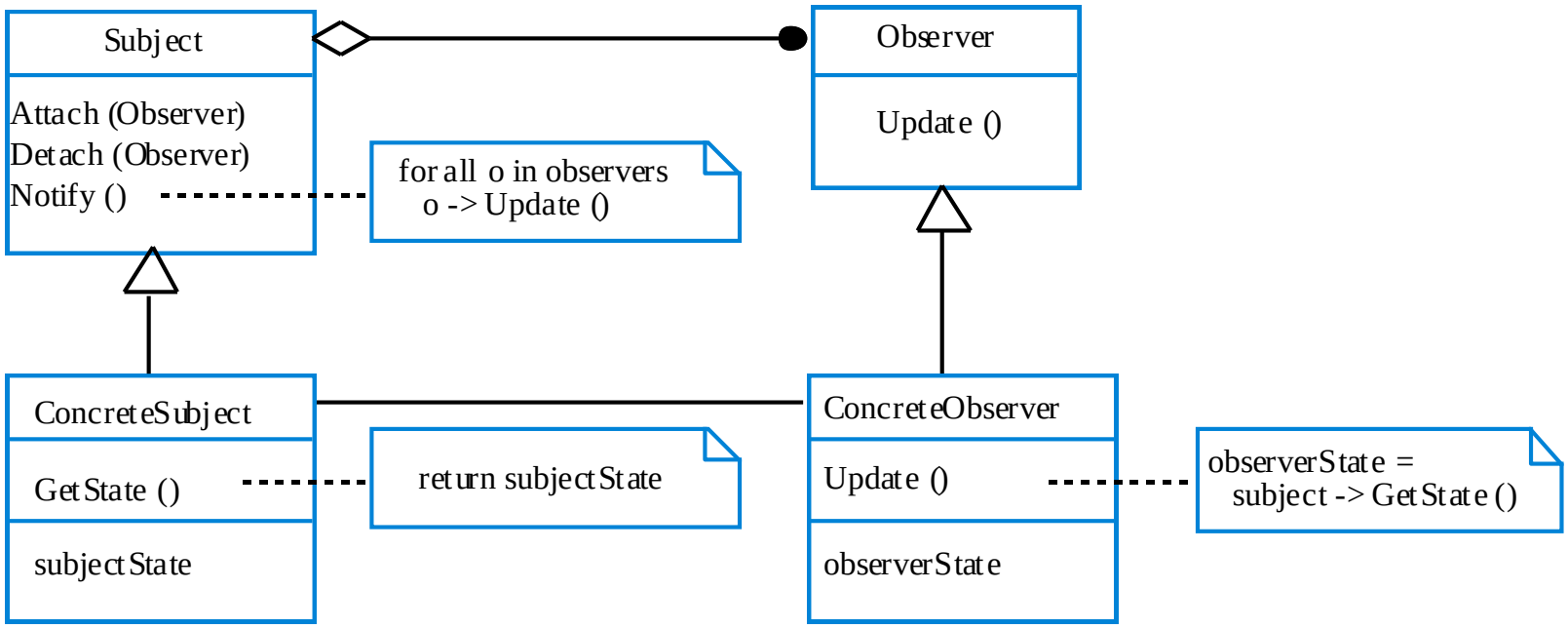
Example: Multiple displays

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

The Observer pattern

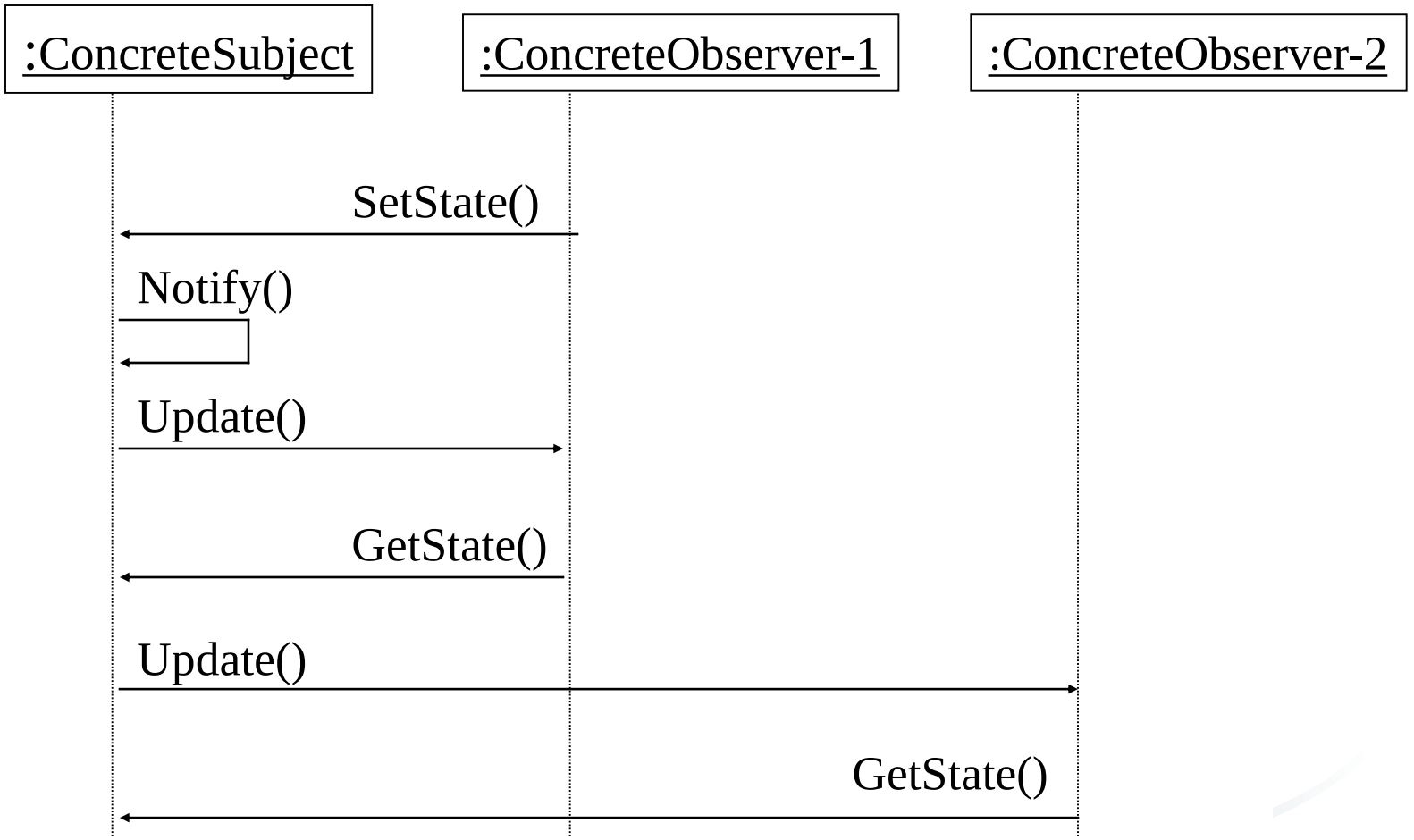
- Name
  - Observer
- Description
  - Separates the display of object state from the object itself
- Problem description
  - Used when multiple displays of state are needed
- Solution description
  - See slide with UML description
- Consequences
  - Optimisations to enhance display performance are impractical

The Observer pattern



aka “Publish and subscribe” mechanism
Choice of “push” or “pull” notification styles

Class collaboration in Observer



When to use the Observer Pattern?

- When there are two or more views on the same “data”
- When an abstraction has two aspects: one dependent on the other. Encapsulating these aspects in separate objects allows one to vary and reuse them independently.
- When a change to one object requires changing others and the number of objects to be changed is not known.
- When an object should be able to notify others without knowing who they are. Avoid tight coupling between objects.

Observer Pattern: Consequences

- Abstract coupling between subject and observer. Subject has no knowledge of concrete observer classes. (What design principle is used?)
- Support for broadcast communication. A subject need not specify the receivers; all interested objects receive the notification.

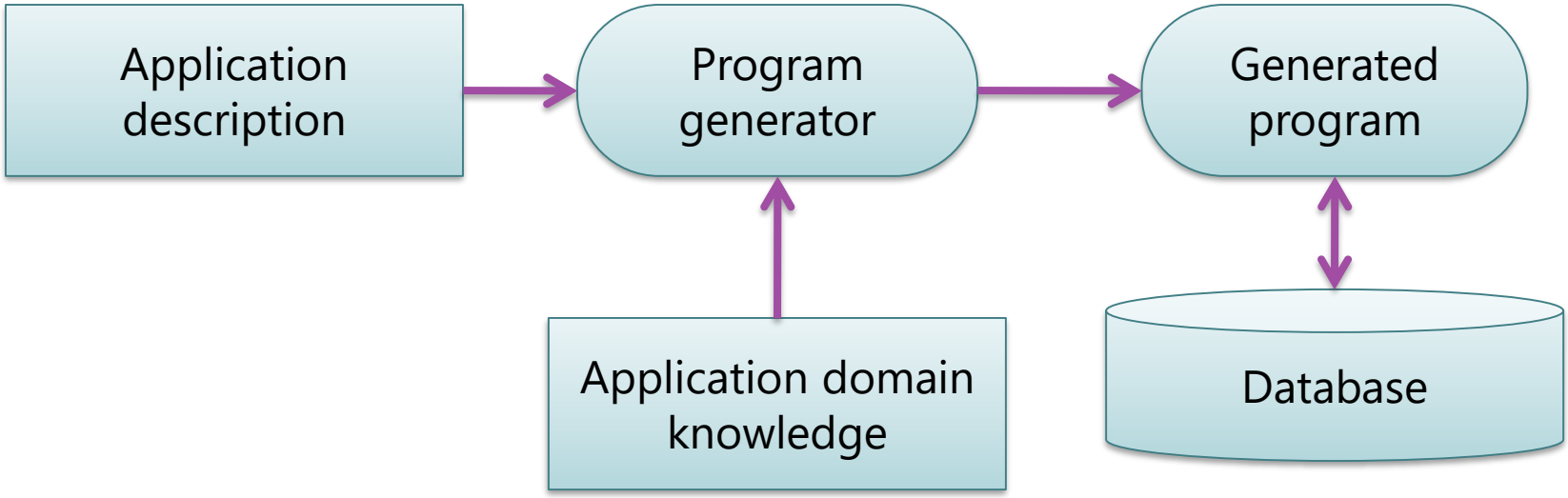
2.2. Generator-based reuse

- Program generators involve the reuse of standard patterns and algorithms
- These are embedded in the generator and parameterised by user commands
- A program is then automatically generated
- Generator-based reuse is possible when domain abstractions and their mapping to executable code can be identified
- A domain specific language is used to compose and control these abstractions (e.g. 4GL)

Types of program generator

- Types of program generator
  - Application generators for business data processing (e.g. in: 4GL, out: COBOL program)
  - Parser and lexical analyser generators for language processing (e.g. in:grammar, out: parser)
  - Code generators in CASE tools (in: SW design, out: program)

Reuse through program generation

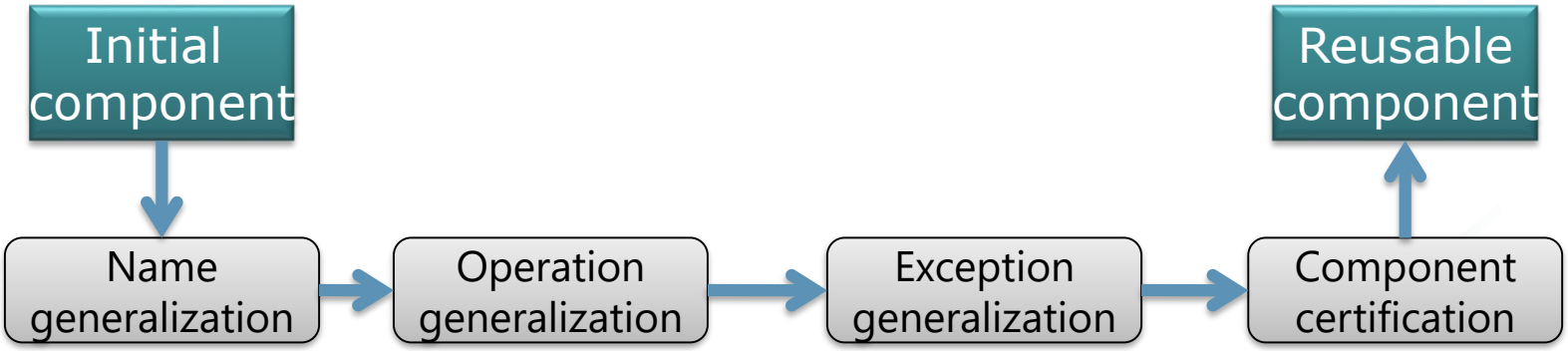


2.3. Reusability enhancement

- The development cost of reusable components is higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organization rather than a project cost
- Generic components may be larger and slower than their specific equivalents

2.3. Reusability enhancement

- Name generalisation
  - Names in a component may be modified so that they are not a direct reflection of a specific application entity
- Operation generalisation
  - Operations may be added to provide extra functionality and application specific operations may be removed
- Exception generalisation
  - Application specific exceptions are removed and exception management added to increase the robustness of the component
- Component certification
  - Component is certified as reusable



3. Software product lines

- A software product line or an application family is a related set of applications that:
  - have generic functionality that can be adapted and configured for use in a specific context
  - have a common, domain-specific architecture
  - The common core of the software product line is reused each time a new application is required.
  - Each specific application is specialised in some way (writing new components, adapting others)
- Product line development is one of application reuse approaches. This involves the reuse of entire application systems either by configuring a system for an environment or integrating two or more systems to create a new application

3.1. Product line adaptation

- Adaptation may involve:
  - Component and system configuration
  - Adding new components to the system
  - Selecting from a library of existing components
  - Modifying components to meet new requirements

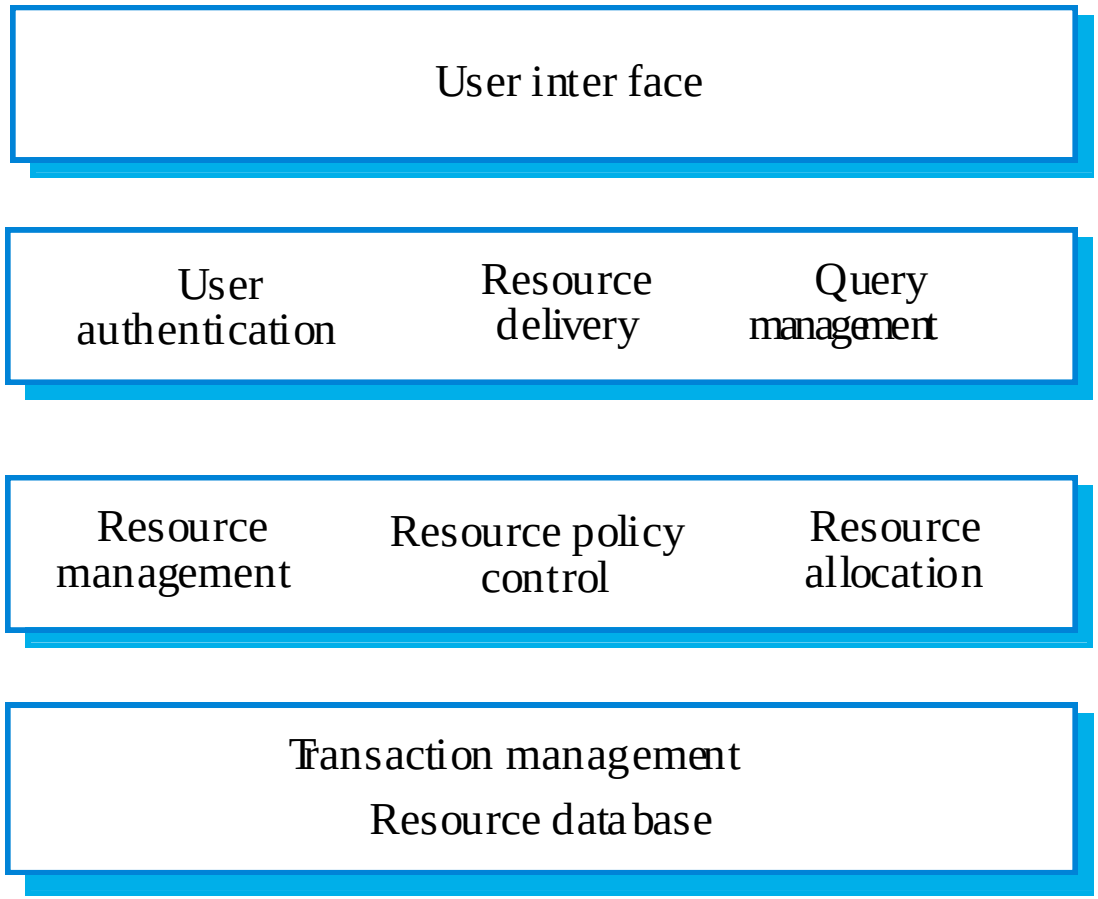
3.2. Product line specialization

- Platform specialization
  - Different versions of the application are developed for different platforms
- Configuration specialization
  - Different versions of the application are created to handle different peripheral devices
- Functional specialization
  - Different versions of the application are created for customers with different requirements

3.3. Product line architectures

- Architectures must be structured in such a way to separate different sub-systems and to allow them to be modified
- The architecture should also separate entities and their descriptions and the higher levels in the system access entities through descriptions rather than directly

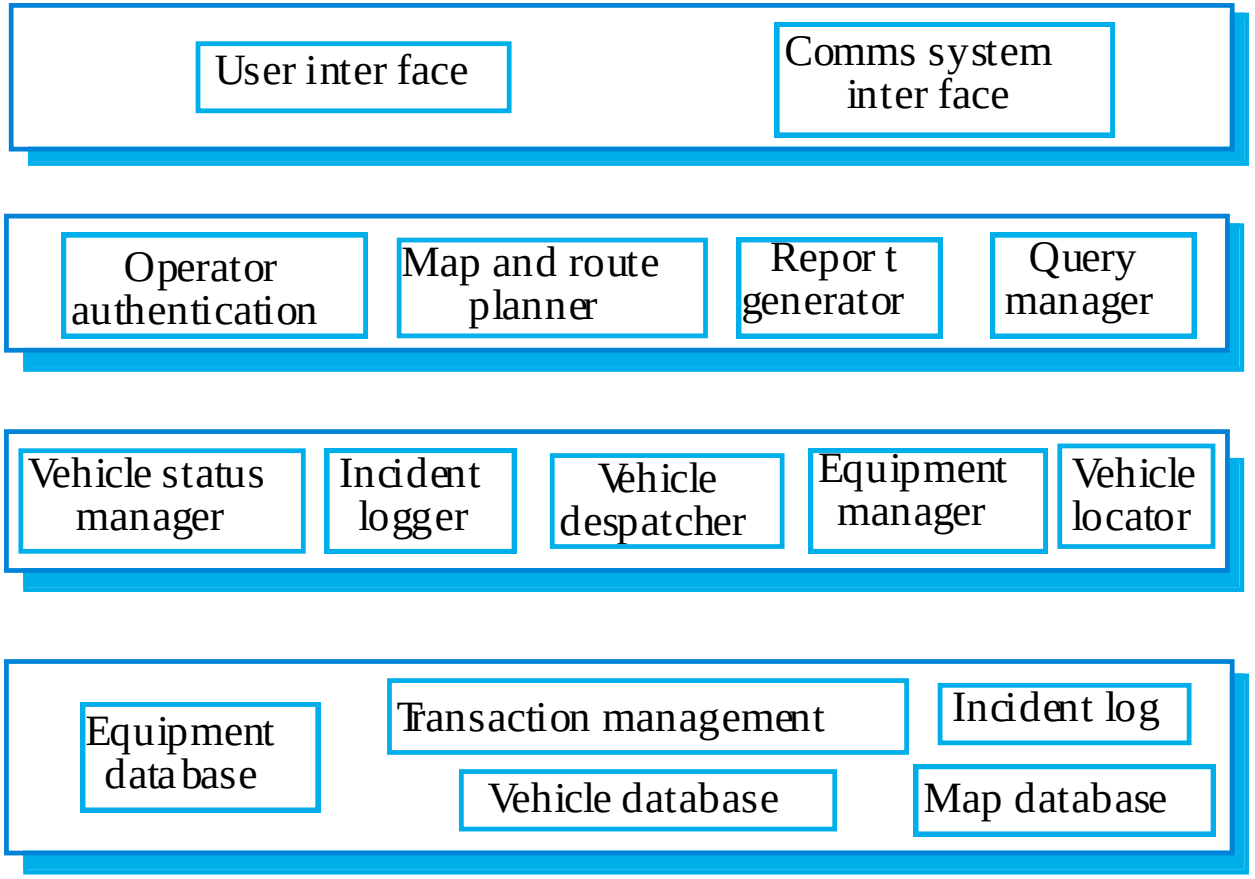
Example1 : Resource management system



Example 1: A vehicle dispatching app

- A specialised resource management system where the aim is to allocate resources (vehicles) to handle incidents
- Adaptations include:
  - At the UI level, there are components for operator display and communications;
  - At the I/O management level, there are components that handle authentication, reporting and route planning;
  - At the resource management level, there are components for vehicle location and despatch, managing vehicle status and incident logging;
  - The database includes equipment, vehicle and map databases

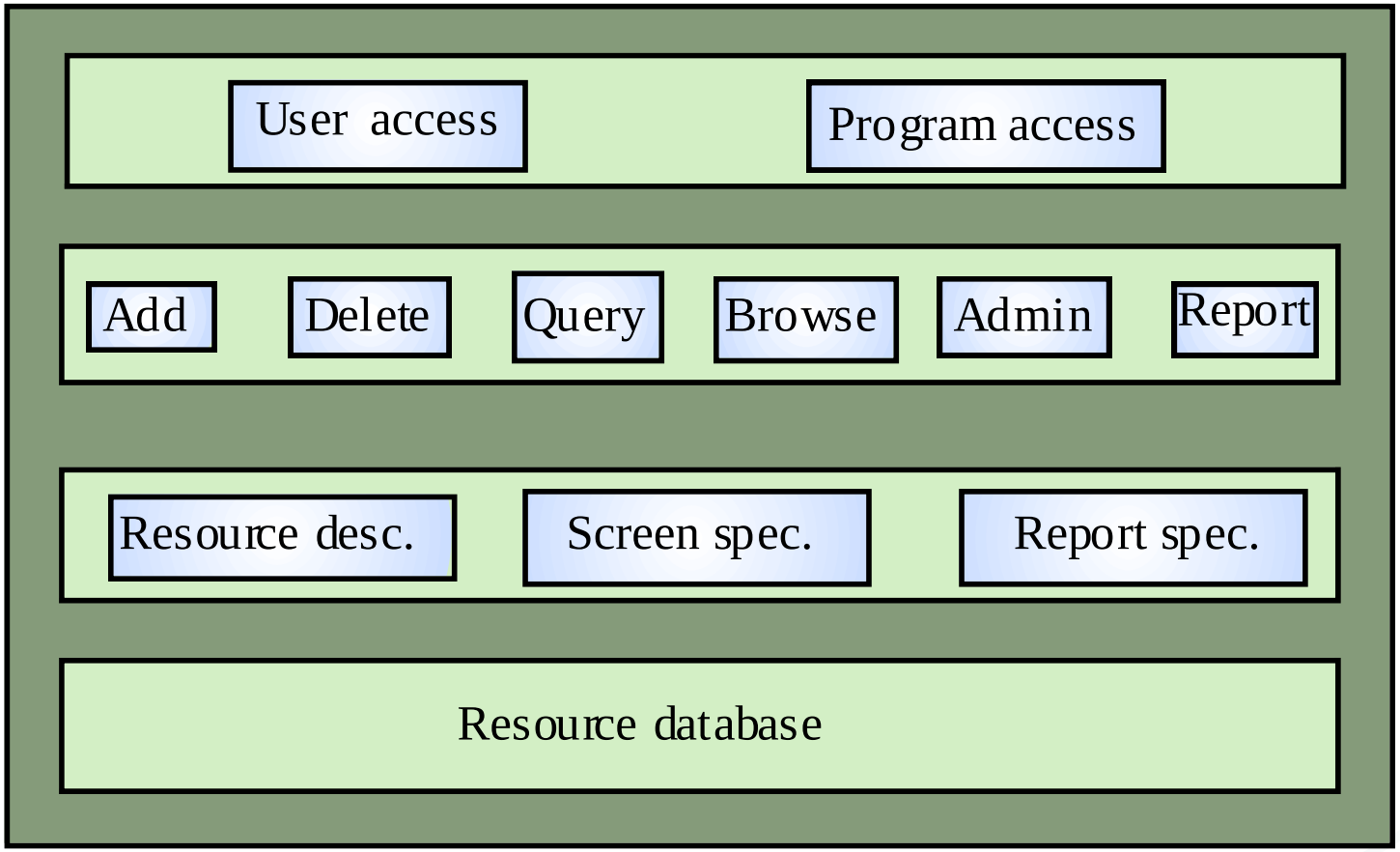
Example 1: A vehicle dispatching app



Example 2: Inventory management systems

- Resource database
  - Maintains details of the things that are being managed
- I/O descriptions
  - Describes the structures in the resource database and input and output formats that are used
- Query level
  - Provides functions implementing queries over the resources
- Access interfaces
  - A user interface and an application programming interface

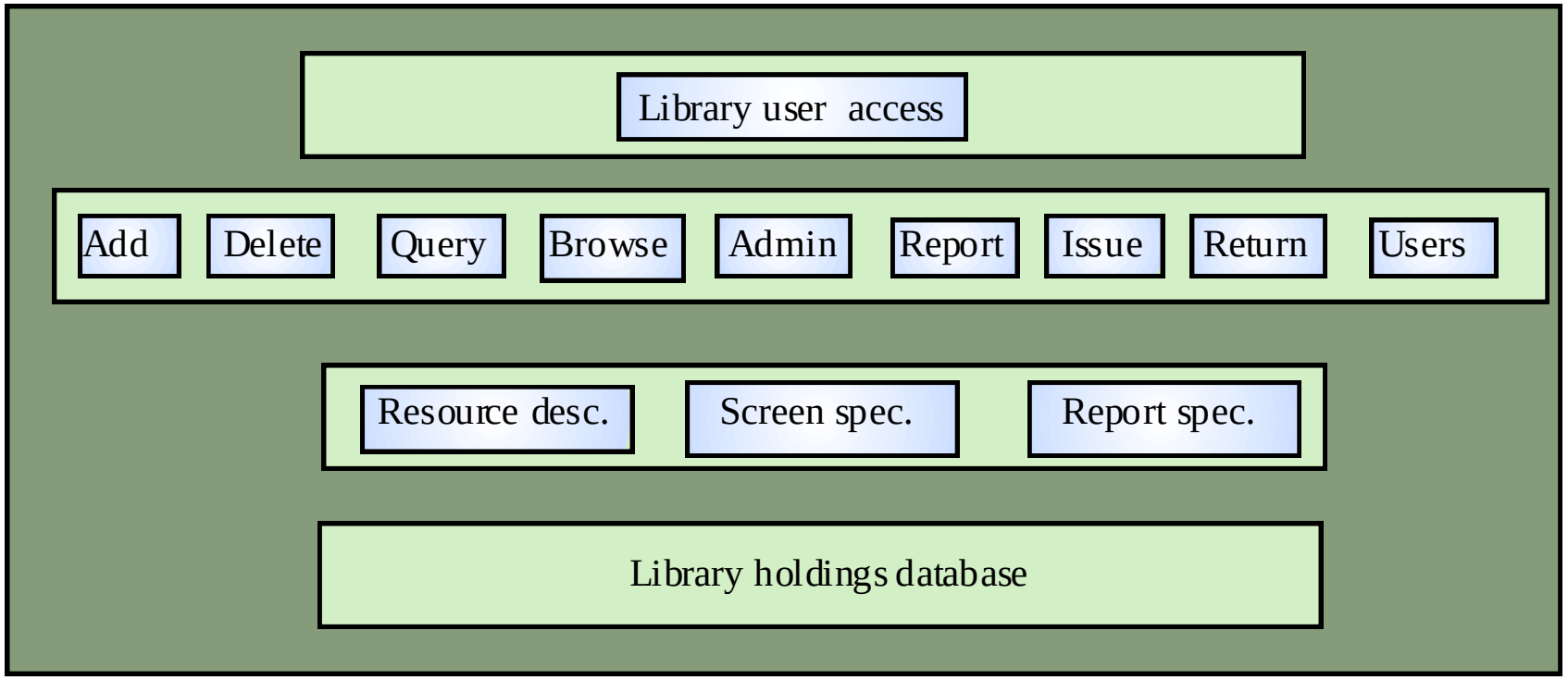
Example 2: Inventory management



Example 2: Library system

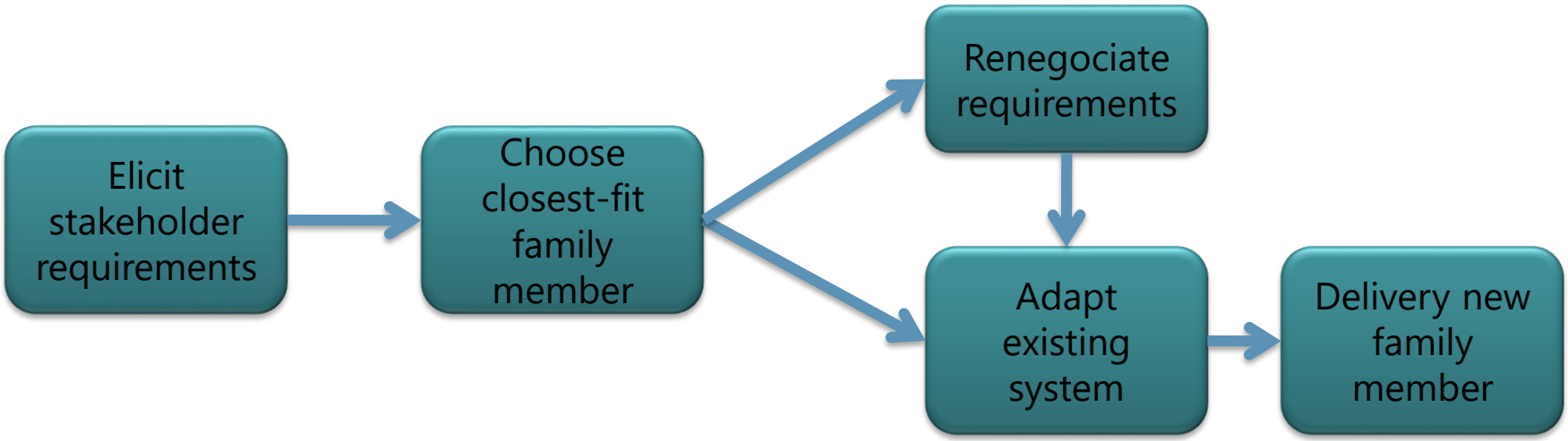
- The resources being managed are the books in the library
- Additional domain-specific functionality (issue, borrow, etc.) must be added for this application

Example 2: Library system



Adaptation of the inventory management system

3.4. Product instance development (Family member development)



Adapting an application family to create a new application

3.4. Product instance development (Family member development)

- Elicit stakeholder requirements
  - Use existing family member as a prototype
- Choose closest-fit family member
  - Find the family member that best meets the requirements
- Re-negotiate requirements
  - Adapt requirements change as necessary to capabilities of the software (minimize change)
- Adapt existing system
  - Develop new modules and make changes for family member
- Deliver new family member
  - Document key features for further member development (reuse)

Key points

- Use of patterns
  - Very effective form of reuse
  - They have high cost for introduction in the design processes
  - Can be used effectively only by experienced programmers (who can recognize generic situations where a pattern can be applied)
- Use of generators:
  - very cost-effective
  - its applicability is limited to a relatively small number of application domains
  - It is easier for end-users to develop programs using generators compared to other component-based approaches to reuse
- Software product lines
  - developed around a common core of shared functionality

Appendix: Design patterns

Patterns, Design, Framework & Architecture

vs Design

- Patterns are design
  - But: patterns transcend the "identify classes and associations" approach to design
  - Instead: learn to recognize patterns in the problem space and translate to the solution
- Patterns can capture OO design principles within a specific domain
- Patterns provide structure to "design"

vs Framework

- Patterns are lower-level than frameworks
- Frameworks typically employ many patterns:
  - Factory
  - Strategy
  - Composite
  - Observer
- Done well, patterns are the "plumbing" of a framework

vs Architecture

- Design Patterns represent a lower level of system structure than "architecture"
- Patterns can be applied to architecture
- Architectural patterns tend to be focused on middleware. They are good at capturing:
  - Concurrency
  - Distribution
  - Synchronization

Covered topics

- **Command and active object**
- Template method and strategy
- Singleton and mono-state
- Null object
- Composite
- Observer
- Abstract factory
- Adapter
- Bridge
- Proxy
- Visitor
- State

1. 1. Command

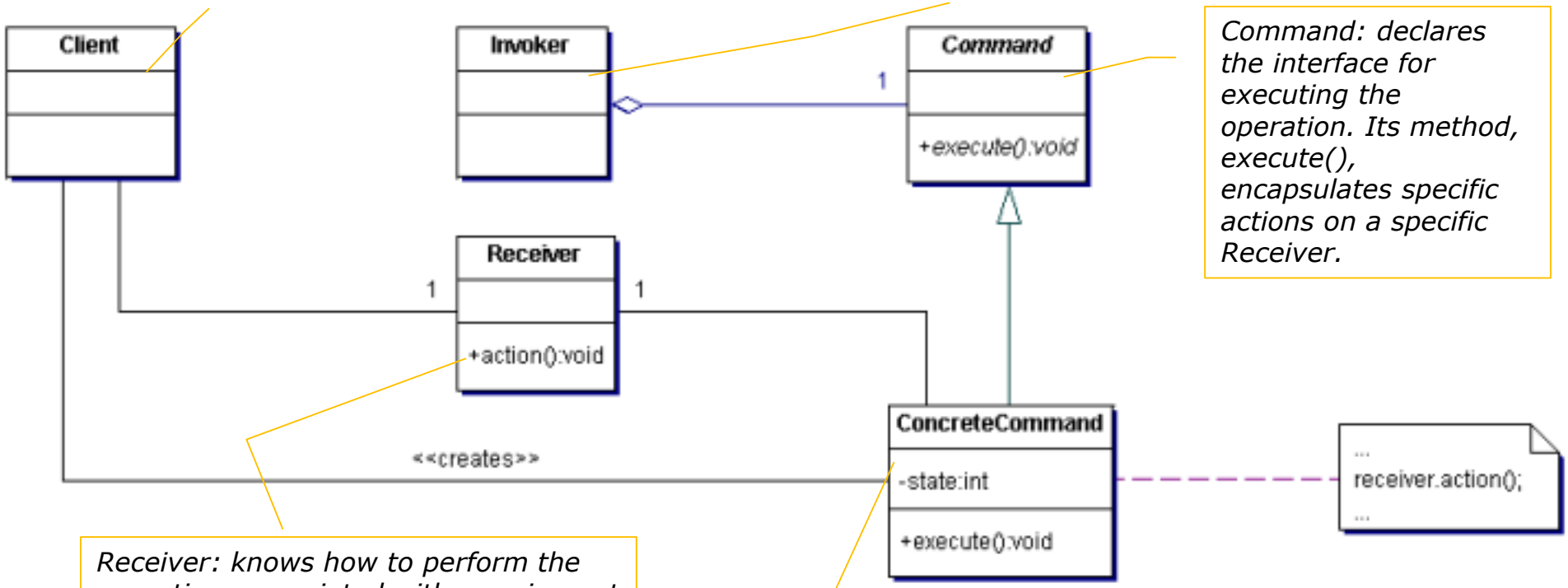
- A Command is a function object (i.e. object whose sole purpose is to encapsulate a function)
- A Command encapsulates a request as an object. By wrapping a method in an object, it can be passed to other methods or objects as a parameter, thereby allowing to parameterize other objects with different requests, queue or log requests, and support undoable operations.

→ A Command is a messenger (because its intent and use is very straightforward) that carries behavior, rather than data.

Structure

Client: creates a ConcreteCommand and sets its receiver. (Client responsible for creating command object. Said object is a set of actions on a receiver)

Invoker: asks the command to carry out the request



Command: declares the interface for executing the operation. Its method, execute(), encapsulates specific actions on a specific Receiver.

Receiver: knows how to perform the operations associated with carrying out a request (actions and Receiver are bound together in command object)

ConcreteCommand: binds a request with a concrete action.

Example: How about lightening a button?

```
public interface Command {
    public void execute();
}
```

As this is a Command, we must implement the Interface.

```
public class LightOnCommand
implements
Command {
    Light light;
    public LightOnCommand (Light
light) {
       this.light = light;
    }
   public void execute() {
     light.on();
   }
}
```

Constructor is passed a specific light
The command is to control and stores it
The Light instance variable. execute() will be received by this instance.

This method calls the on() method on the receiving object, which is light we are controlling.

Using the command object for implementing the elevator controller

```
public class SimpleController {


    Command slot;
    public SimpleController() { }
    public void setCommand (Command command) {
        slot = command;
    }
    public void buttonWasPressed() {
        slot.execute();
    }
}
```

We have one slot to hold our command (i.e., we control one device).

Here's a method for setting the command controlled by slot.

All we do here is take the current command bound to the slot and call its execute() method.

And now a simple test!

```
public class SimpleControllerTest {
    public static void main (String[] args) {
        SimpleController ctl =
         new SimpleController();
        Light light = new Light();
        LightOnCommand lightOn =
         new LightOnCommand (light);
        ctl.setCommand (lightOn);
        ctl.buttonWasPressed();
    }
}

public class DoorOpenCommand implements Command {
    Door door;
    public DoorOpenCommand (Door door) {
        this.door = door;
    }
    public void execute() {
        door.open();
    }
}
```

```
public class ElevatorController {
    public static void main
     (String[] args) {
        SimpleController ctl =
         new SimpleController();
        Light light = new Light();
        Door door = new Door();
        LightOnCommand lightOn =
         new LightOnCommand (light);
        DoorOpenCommand openCmd =
         new DoorOpenCommand (door);
        ctl.setCommand (lightOn);
        ctl.buttonWasPressed();
        ctl.setCommand (openCmd);
        ctl.buttonWasPressed();
    }
}
```

Applicability

- When you need an action as a parameter
  - Commands replace callback functions
    When you need to specify, queue, and execute requests at different times
- When you need to support undo
- When you need to support logging changes
- When you structure a system around high-level operations built on primitive operations
  - A Transactions encapsulates a set of changes to data
  - Systems that use transaction often can use the command pattern
- When you need to support a macro language

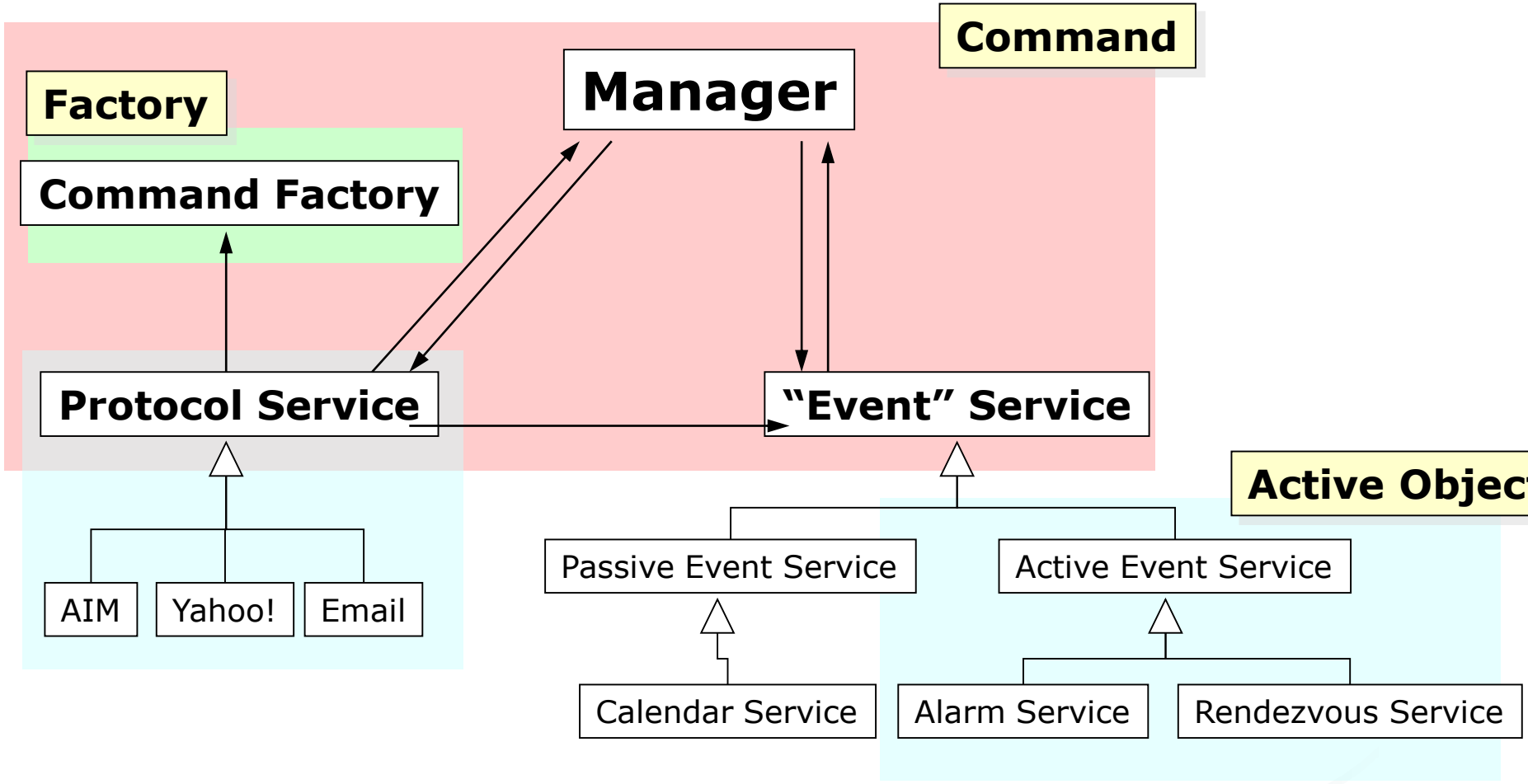
Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it
  - It is easy to add new commands, because you do not have to change existing classes
  - Different user interface elements can generate the same kind of command object
- You can assemble commands into a composite object
- Flexibility
  - in the way requests are activated
  - in the number and functionality of requests
- Allows
  - the user to configure commands performed by a user interface element
  - for the execution of commands in separate threads
- Adding new commands and providing for a macro language comes easy
- Programming execution-related services
  - Commands can be stored for later replay
  - Commands can be logged
  - Commands can be rolled back
- Testability at application level
- Concurrency

1.2. Active object

- The Active Object design pattern decouples method execution from method invocation that reside in their own thread of control.
- The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.

Modelling Collaboration



Minimizing interference between channel participants: Active Object Pattern

Fairness in Event Processing: Command Pattern

Covered topics

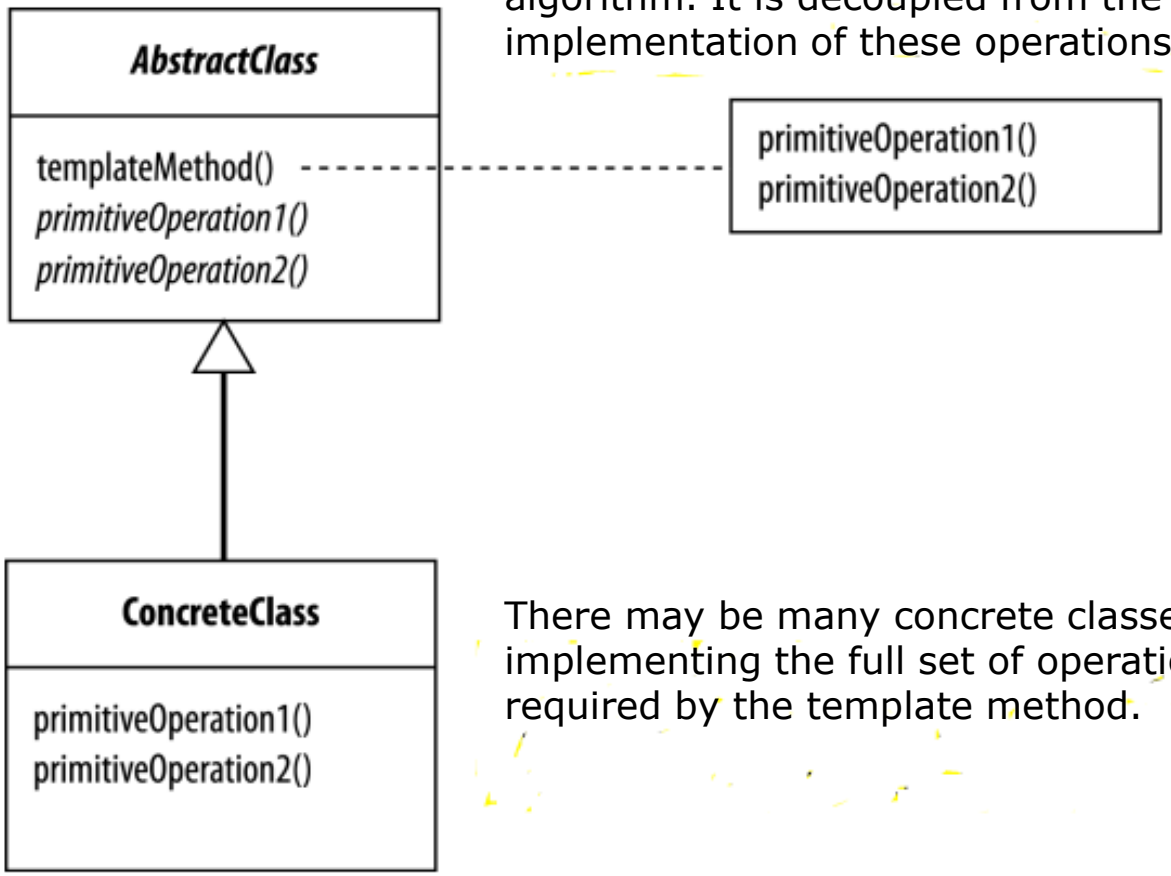
- Command and active object
- **Template method and strategy**
- Singleton and mono-state
- Null object
- Composite
- Observer
- Abstract factory
- Adapter
- Bridge
- Proxy
- Visitor
- State

2.1. Template method

- The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure. Some steps are deferred to subclasses.
- Template Method is used prominently in frameworks

Structure

The AbstractClass contains the template method…
… and abstract versions of the operations used by template method.



The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.

There may be many concrete classes, each implementing the full set of operations required by the template method.

Example: A Game Class

- We have many types of games.
- In each game there are `playersCount` players but only one may play at a time.
- Write the superclass Game, which includes method `playOneGame(int playersCount)`

```
abstract class Game {
    private int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    final void playOneGame(int
       playersCount) {
        this.playersCount =
       playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
}
```

Example: The concrete games

```
class Monopoly extends Game {
    void initializeGame()  {
       // ...
    }
    void makePlay(int player)  {
       // ...
    }
    boolean endOfGame()  {
       // ...
    }
    void printWinner()  {        //
       ...
    }

    /* Specific declarations for
     the Monopoly game. */
    // ...
}
```

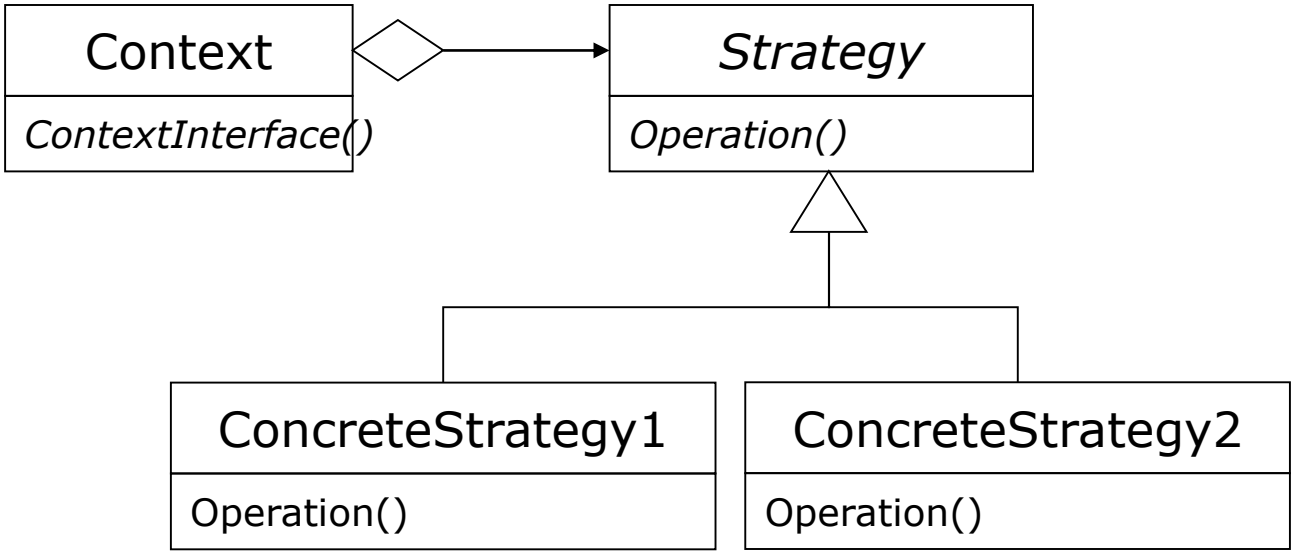
```
class Chess extends Game  {
    void initializeGame()  {
       // ...
    }
    void makePlay(int player)  {
       // ...
    }
    boolean endOfGame() {        //
       ...
    }
    void printWinner()  {        //
       ...
    }

    /* Specific declarations for
     the Chess game. */
    // ...
}
```

Usage of Template Methods

- Java Arrays class: sort()
  - The compareTo() function is abstract
- Swing JFrames
  - good old paint()!
- Also: java.awt.Component
  - update()
- In general: frameworks
  - OpenOffice: sfx2 (document persistence, multiple/ different views for documents, command dispatching infrastructure)
  - FlashMX2004 Version 2 component architecture
- FactoryMethod is a specialization

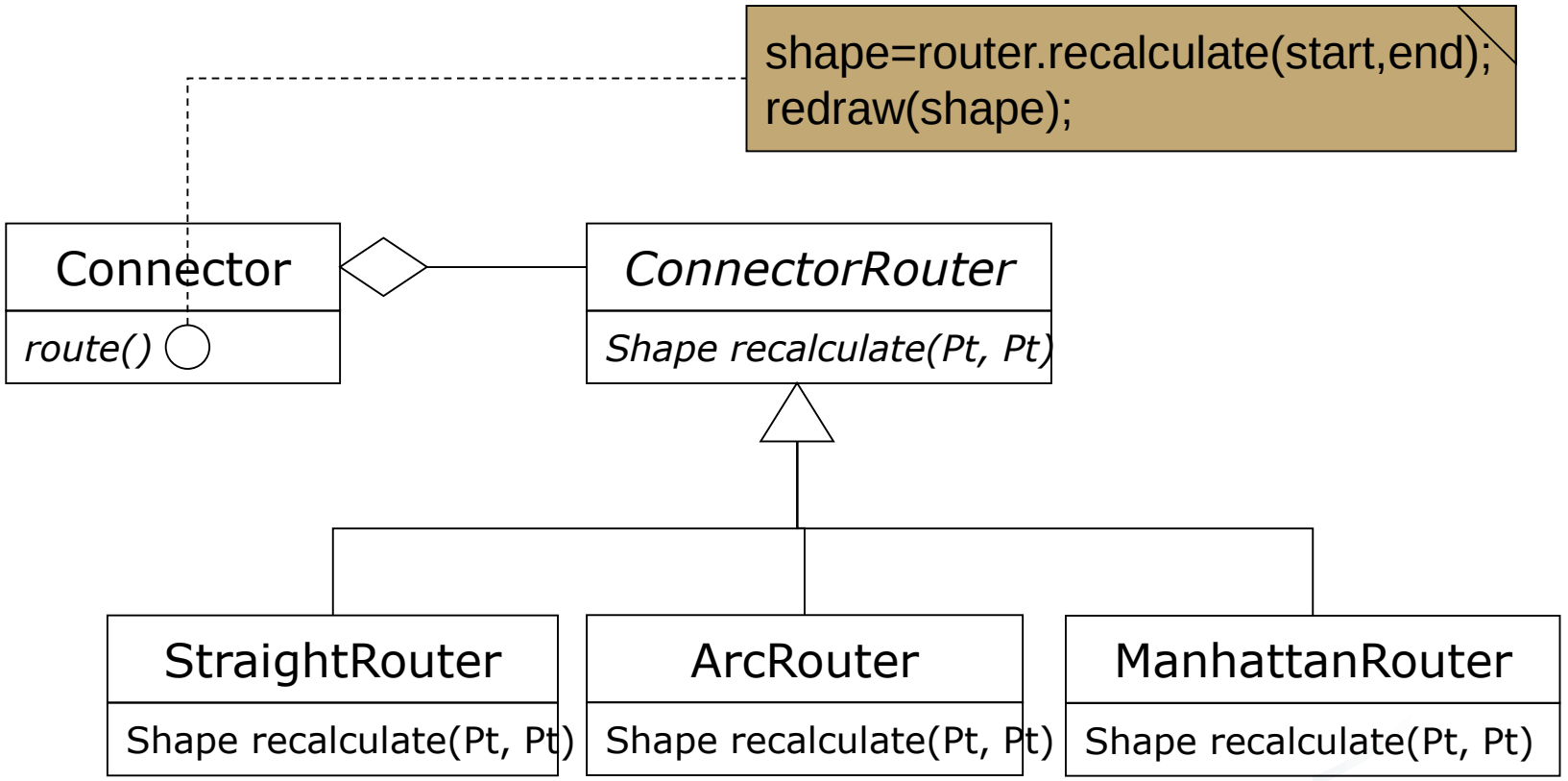
2.2. Strategy

- Defines a family of algorithms
  - Make algorithms interchangeable---”changing the guts”,
  - Encapsulate each one
- Alternative to subclassing
- Choice of implementation at run-time
- Increases run-time complexity
- Lets the algorithm vary independently from clients that use it



Example

- Drawing different connector styles



Applicability

- Use the Strategy pattern when
  - you need different variants of an algorithm
  - an algorithm uses data that clients shouldn't know about
  - a class defines many behaviors, and these appear as multiple switch statement in the classes operations
  - many related classes differ only in their behavior

Implementation

- Defining the Strategy and Context interfaces
  - How does data flow between them
  - Context pass data to Strategy
  - Strategy has point to Context, gets data from Context
- Strategies as template parameters
  - Can be used if Strategy can be selected at compile-time and does not change at runtime
- Making Strategy objects optional
  - Give Context default behavior
  - If default used no need to create Strategy object

Consequences

- Families of related algorithms
- Alternative to subclassing of Context
  - What is the big deal? You still subclass Strategy!
- Eliminates conditional statements
  - Replace in Context code like:

```
switch ( flag ) {
  case A: doA(); break;
  case B: doB(); break;
  case C: doC(); break;
}
```

  With code like:

```
strategy.do();
```

- Gives a choice of implementations
- Clients must be aware of different Strategies

```
SortedList studentRecords =
new
SortedList(new ShellSort());
```

- Communication overhead between Strategy and Context
- Increase number of objects

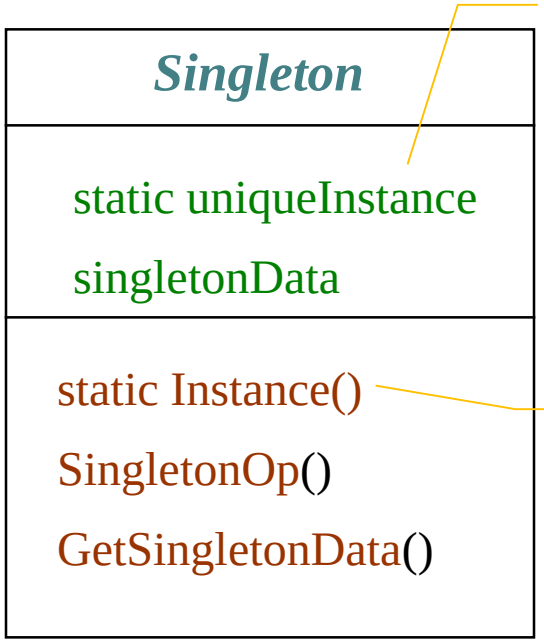
Covered topics

- Command and active object
- Template method and strategy
- **Singleton and mono-state**
- Null object
- Composite
- Observer
- Abstract factory
- Adapter
- Bridge
- Proxy
- Visitor
- State

3.1. Singleton

- The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.
  - 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.
- For example, one printer spooler object, one file system, one window manager, etc.

Structure



variable that holds our one and only instance of Singleton.

class method, you can conveniently access this method anywhere in your code using Singleton.Instance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.

Code

```
class Singleton {
// Only one instance can ever be created.
   public:
          static Singleton*
          Instance();
// Creation hidden inside Instance().

   protected:
          Singleton();


   private:
          Static Singleton*
          _instance;
// Cannot access directly.
   }
```

```
// Clients access the singleton
// exclusively via the Instance member
// function.

Singleton* Singleton::_instance=0;


Singleton* Singleton:: Instance(){

    if (_instance ==0) {
          _instance=new Singleton;
    }
return _instance;

}
```

Consequence

- Creating a singleton class is a lot of code!! Why not just use a global variable?
  - Controlled access to sole instance
  - Avoid polluting default name space with global variables
  - Allows lazy allocation (not allocated until / unless needed)
- When to implement a class as a singleton ?
  - When every application uses this class exactly the same way
  - When every application only ever needs one instance of the class, and…
  - Clients of the class are unaware of which application they belong to.

Example: Logger

- Suppose we need to implement a “Logger” class. This class records
  - activities performed by our systems
  - errors generated by system
  - “log” usually implies one file in one place
- But we also require that only one Logger operate at any one time
  - Otherwise the file associated with logger would be created every single time a Logger is instantiated!

- Does every application uses this logger class exactly the same way?
  - Yes: Applications will register the logger as a listener in the same way
- Does every application only ever need one instance of the logger class?
  - Yes: Every application should be able to work with just one instance
- Are clients of the logger class unaware of which application they belong to?
  - Yes: Any object should be able to request logging, so loggers should not be coupled to any application.

Singleton & Monostate

- applicable to any class
- lazy evaluation: if not used, not created
- not inherited: a derived class is not singleton
- can be created through derivation
- non-transparent: the user knows

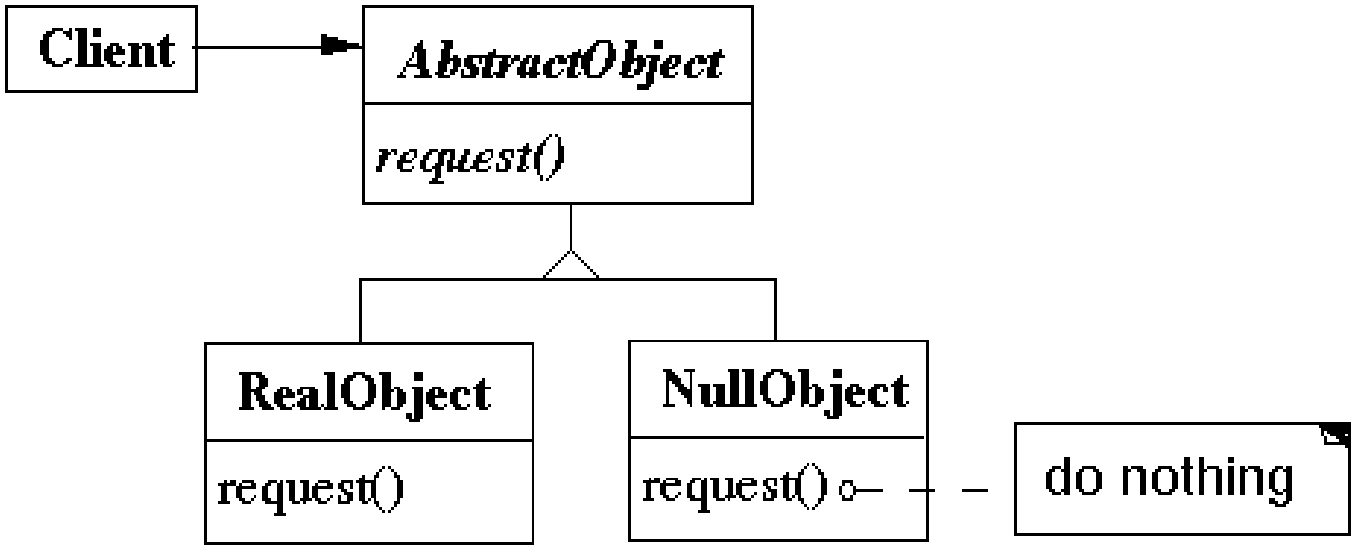
- inherited: a derived class is monostate
- polymorphism: methods can be overridden
- normal class cannot be converted through derivation
- transparent: the user does not need to know…

Covered topics

- Command and active object
- Template method and strategy
- Singleton and mono-state
- **Null object**
- Composite
- Observer
- Abstract factory
- Adapter
- Bridge
- Proxy
- Visitor
- State

4. Null Object

- Null Object implements all the operations of the real object, but these operations do nothing



Applicability

- Use the Null Object pattern when:
  - Some collaborator instances should do nothing
  - You want clients to ignore the difference between a collaborator that does something and one that does nothing - so the client does not have to explicitly check for null or some other special value
  - You want to be able to reuse the do-nothing behavior so that various clients that need this behavior will consistently work in the same way
- Use a variable containing null or some other special value instead of the Null Object pattern when:
  - Very little code actually uses the variable directly
  - The code that does use the variable is well encapsulated - at least in one class
  - The code that uses the variable can easily decide how to handle the null case and will always handle it the same way

Consequences

- Advantages
  - Uses polymorphic classes
  - Simplifies client code
  - Encapsulates do nothing behavior
  - Makes do nothing behavior reusable
- Disadvantages
  - Forces encapsulation:
    - Makes it difficult to distribute or mix into the behavior of several collaborating objects
  - May cause class explosion
  - Forces uniformity
    - Different clients may have different idea of what "do nothing" means
  - Is non-mutable
    - NullObject objects can not transform themselves into a RealObject

Implementation

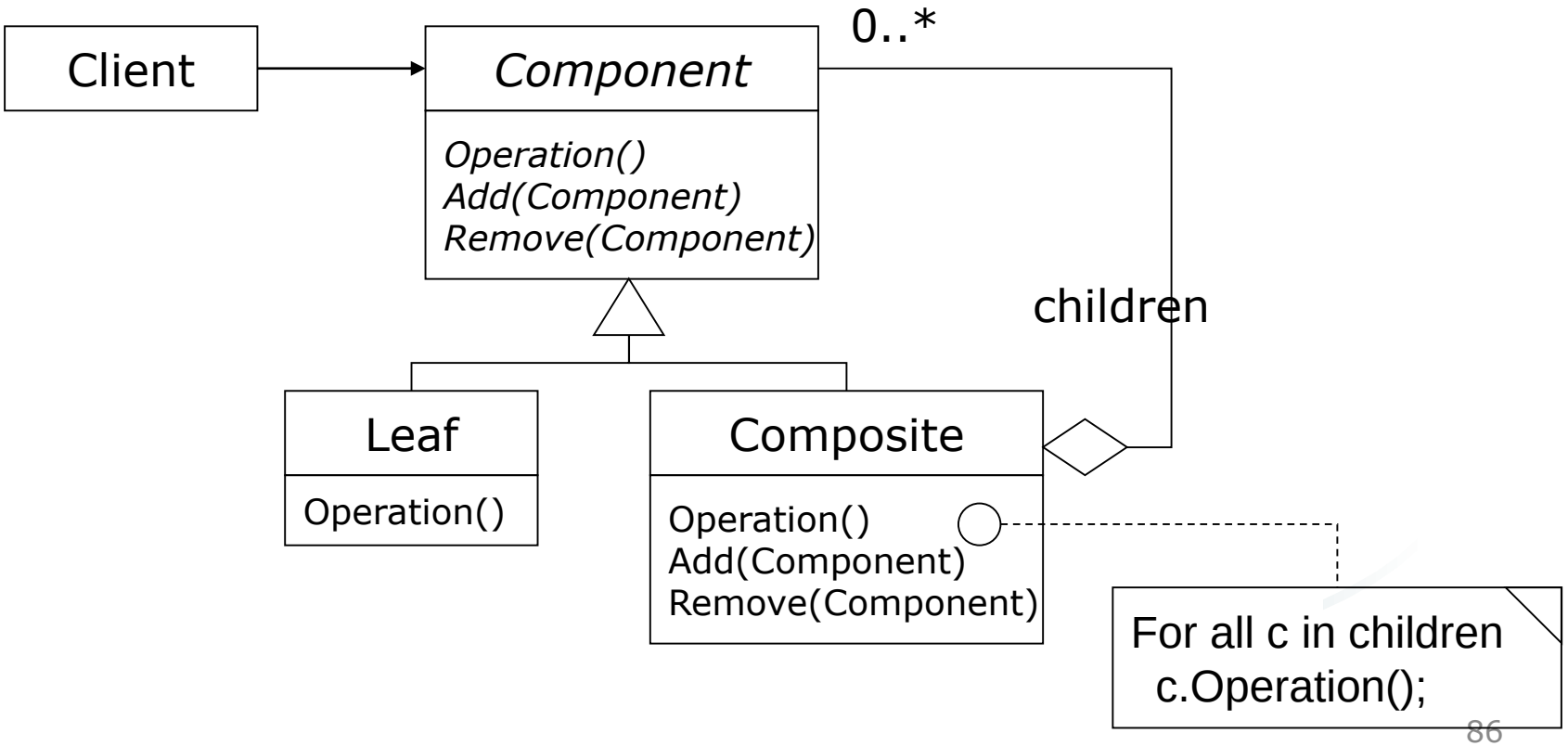
- Too Many classes
  - Eliminate one class by making NullObject a subclass of RealObject
- Multiple Do-nothing meanings
  - If different clients expect do nothing to mean different things use Adapter pattern to provide different do nothing behavior to NullObject
- Transformation to RealObject
  - In some cases a message to NullObject should transform it to a real object
  - Use the proxy pattern

Covered topics

- Command and active object
- Template method and strategy
- Singleton and mono-state
- Null object
- **Composite**
- Observer
- Abstract factory
- Adapter
- Bridge
- Proxy
- Visitor
- State

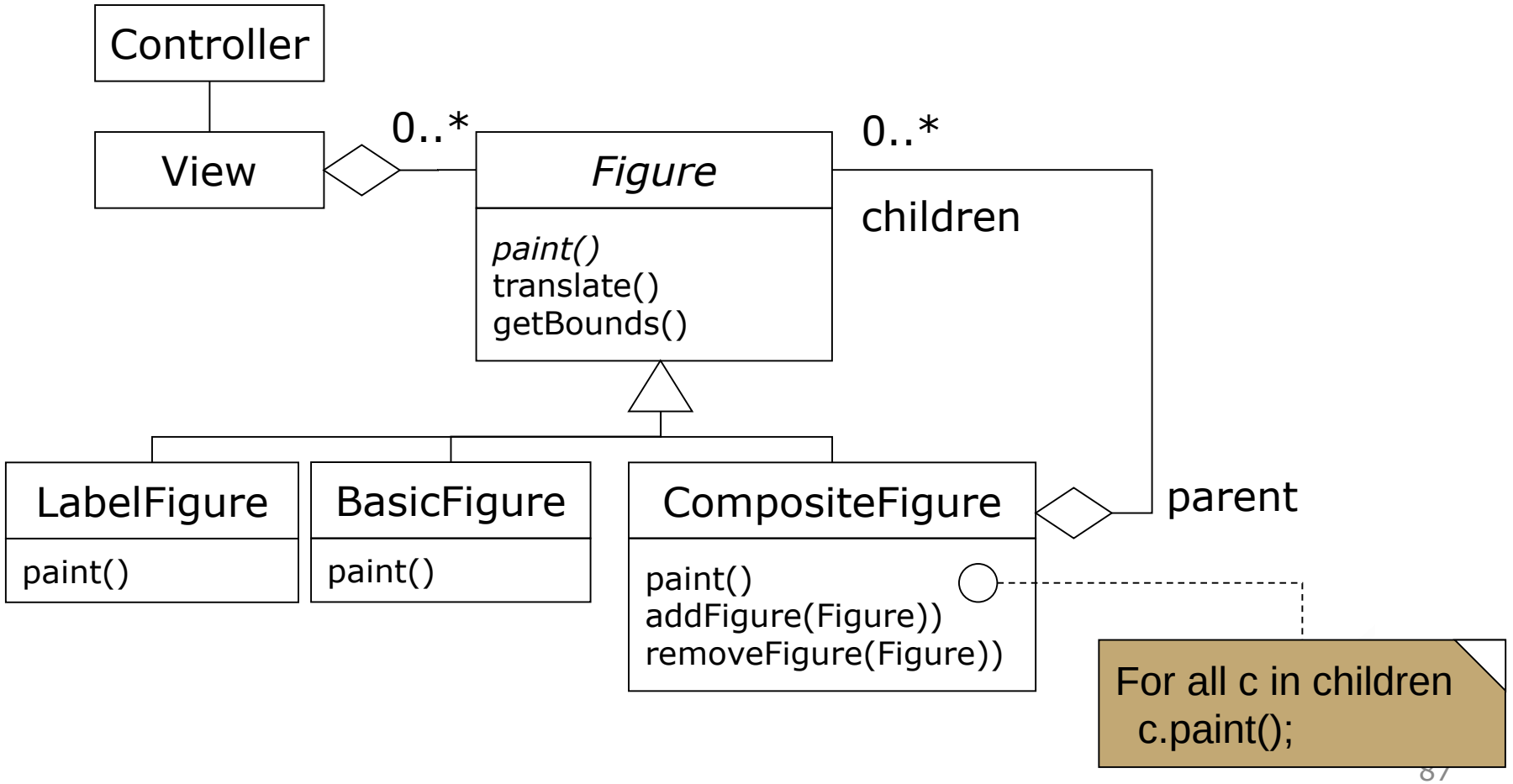
5. Composite

- Construct part-whole hierarchy
- Simplify client interface to leaves/composites
- Easier to add new kinds of components



Example

- Figures in a structured graphics toolkit



Applicability

- Use Composite pattern when
  - you want to represent part-whole hierarchies of objects
  - you want clients to be able to ignore the difference between compositions of objects and individual objects

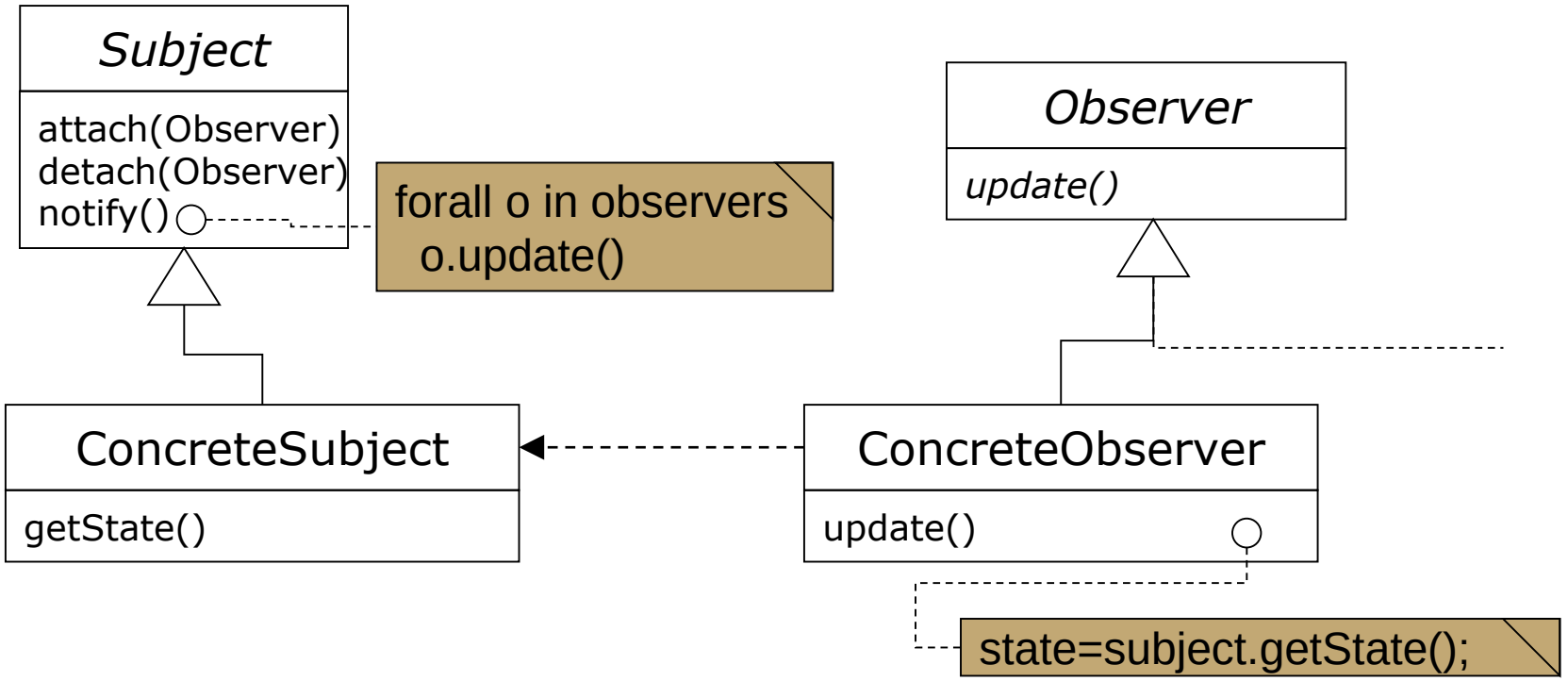
Covered topics

- Command and active object
- Template method and strategy
- Singleton and mono-state
- Null object
- Composite
- **Observer**
- Abstract factory
- Adapter
- Bridge
- Proxy
- Visitor
- State

6. Observer

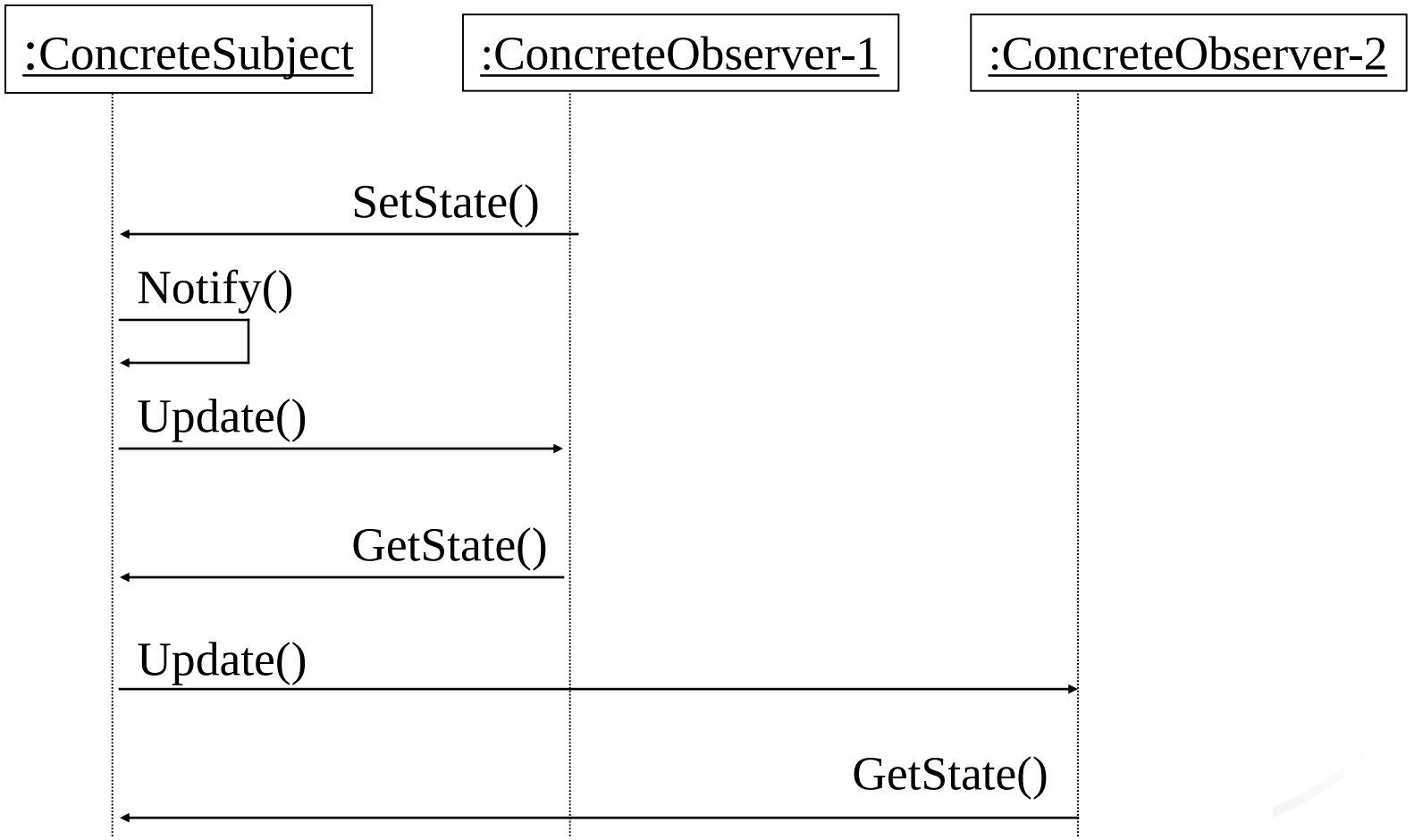
- Need to separate presentational aspects with the data, i.e. separate views and data.
- Classes defining application data and presentation can be reused.
- Change in one view automatically reflected in other views. Also, change in the application data is reflected in all views.
- Defines one-to-many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

Observer



aka “Publish and subscribe” mechanism
Choice of “push” or “pull” notification styles

Class collaboration in Observer



When to use the Observer Pattern?

- When there are two or more views on the same “data”
- When an abstraction has two aspects: one dependent on the other. Encapsulating these aspects in separate objects allows one to vary and reuse them independently.
- When a change to one object requires changing others and the number of objects to be changed is not known.
- When an object should be able to notify others without knowing who they are. Avoid tight coupling between objects.

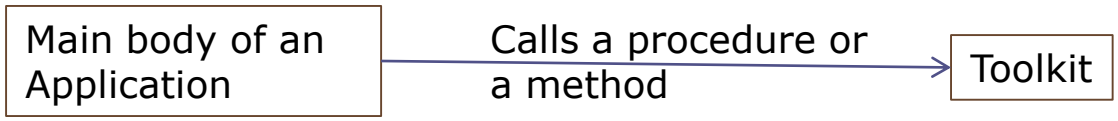
Observer Pattern: Consequences

- Abstract coupling between subject and observer. Subject has no knowledge of concrete observer classes. (What design principle is used?)
- Support for broadcast communication. A subject need not specify the receivers; all interested objects receive the notification.
- Support for broadcast communication. A subject need not specify the receivers; all interested objects receive the notification.

Covered topics

- Command and active object
- Template method and strategy
- Singleton and mono-state
- Null object
- Composite
- Observer
- **Abstract factory**
- Adapter
- Bridge
- Proxy
- Visitor
- State

7. Abstract factory



Toolkits: Collection of related and reusable classes
e.g. C++ I/O stream library

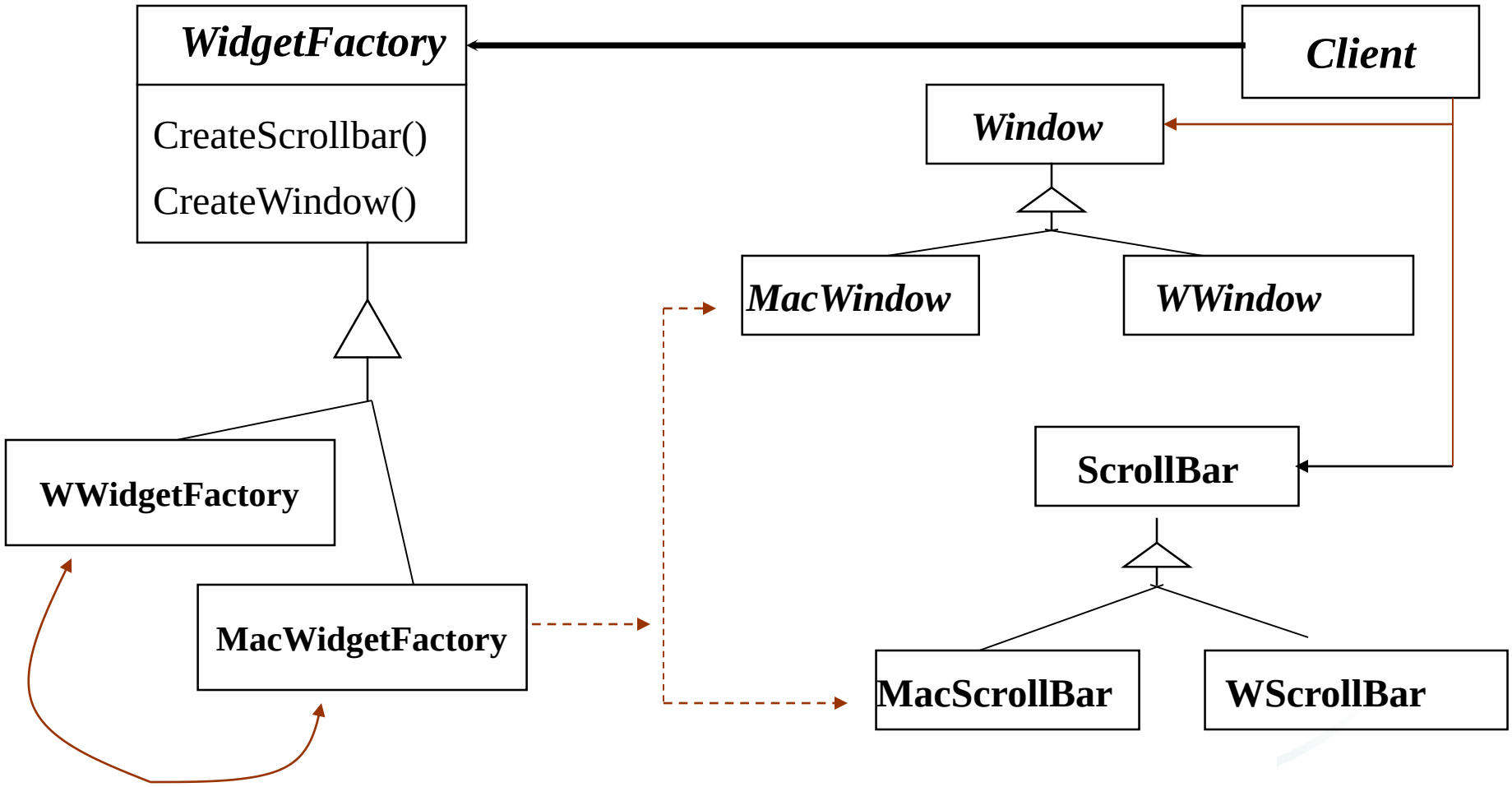
Problem

- Consider a user interface toolkit to support multiple look-and-feel standards.
- For portability an application must not hard code its widgets for one look and feel.
- →How to design the application so that incorporating new look and feel requirements will be easy?

Solution

- Define an abstract WidgetFactory class. This class declares an interface to create different kinds of widgets:
  - 1 abstract class for each kind of widget
  - Concrete subclasses implement widgets for different standards.
  - 1 operation to return a new widget object for each abstract widget class. Clients call these operations to obtain instances of widgets without being aware of the concrete classes they use.

Abstract Factory: Solution

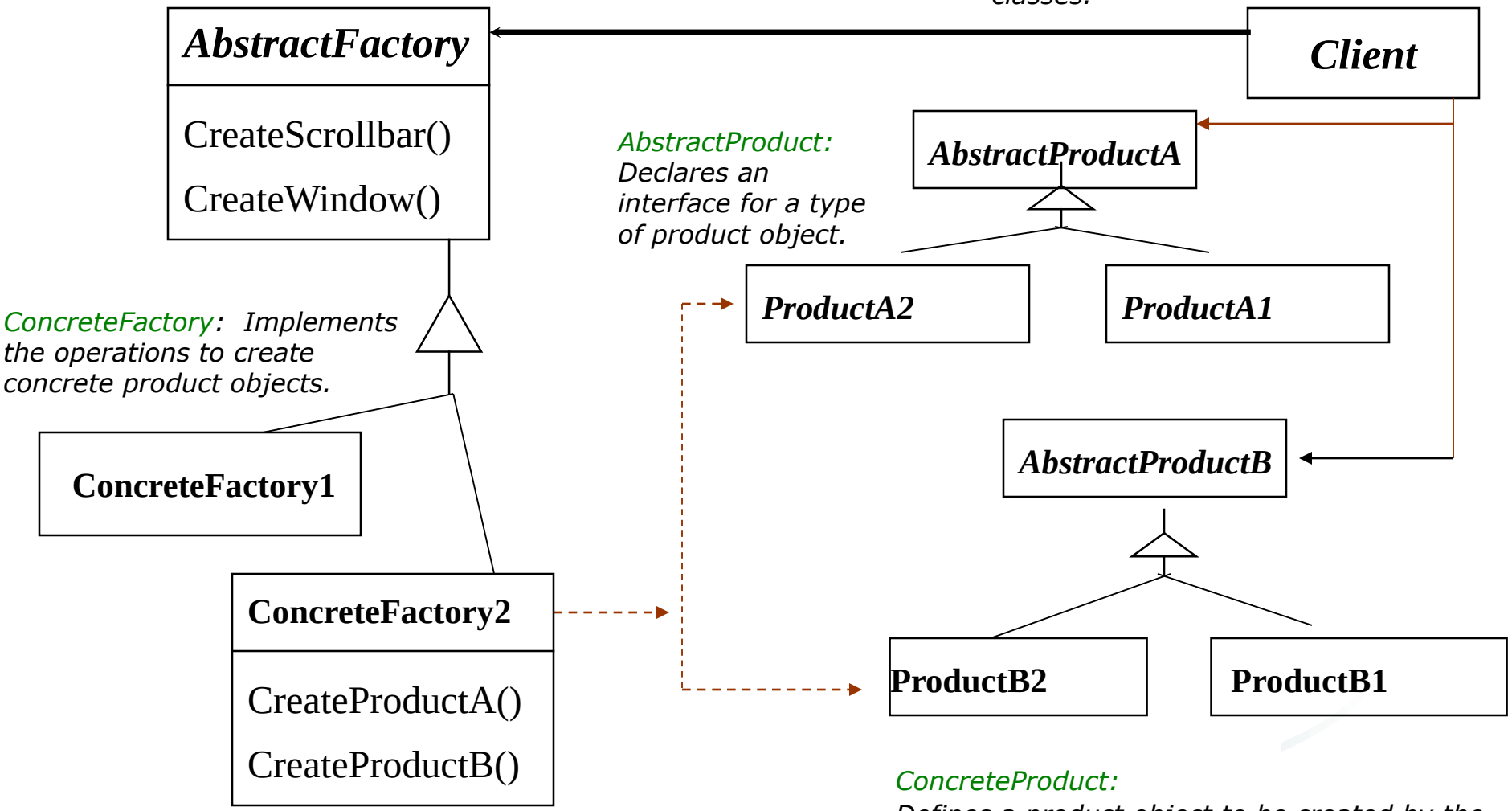


One for each standard.

Abstract Factory

AbstractFactory:
Declares the interface for operations to create abstract product objects

Client:
Uses only the interface declared by the abstractFactory and AbstractProduct classes.

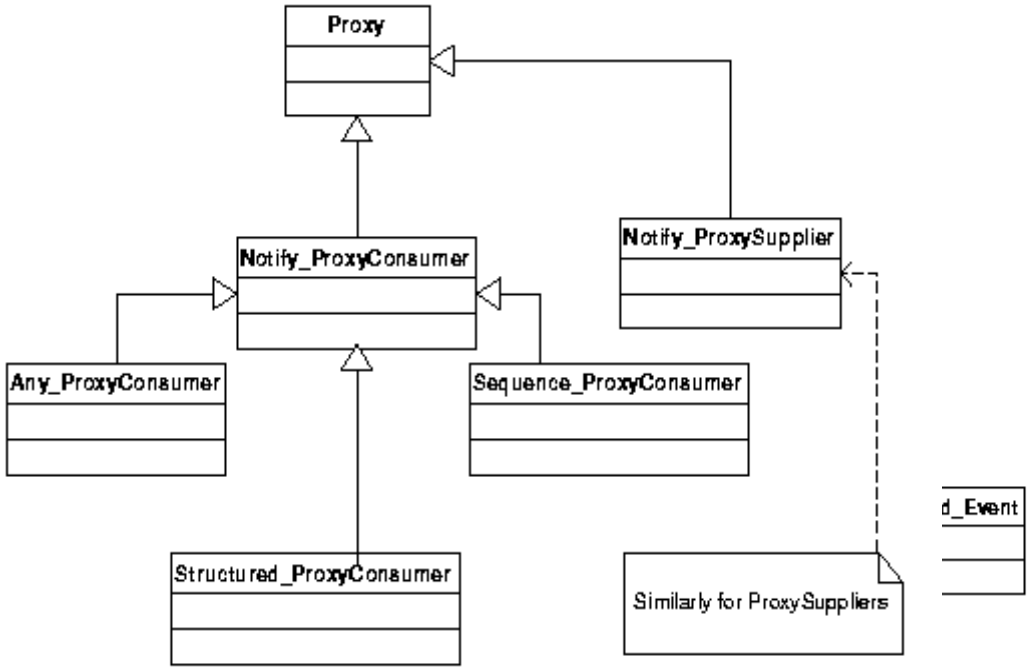


ConcreteProduct:
Defines a product object to be created by the corresponding factory.

Covered topics

- Command and active object
- Template method and strategy
- Singleton and mono-state
- Null object
- Composite
- Observer
- Abstract factory
- **Adapter**
- Bridge
- Proxy
- Visitor
- State

Example: Adapter Pattern



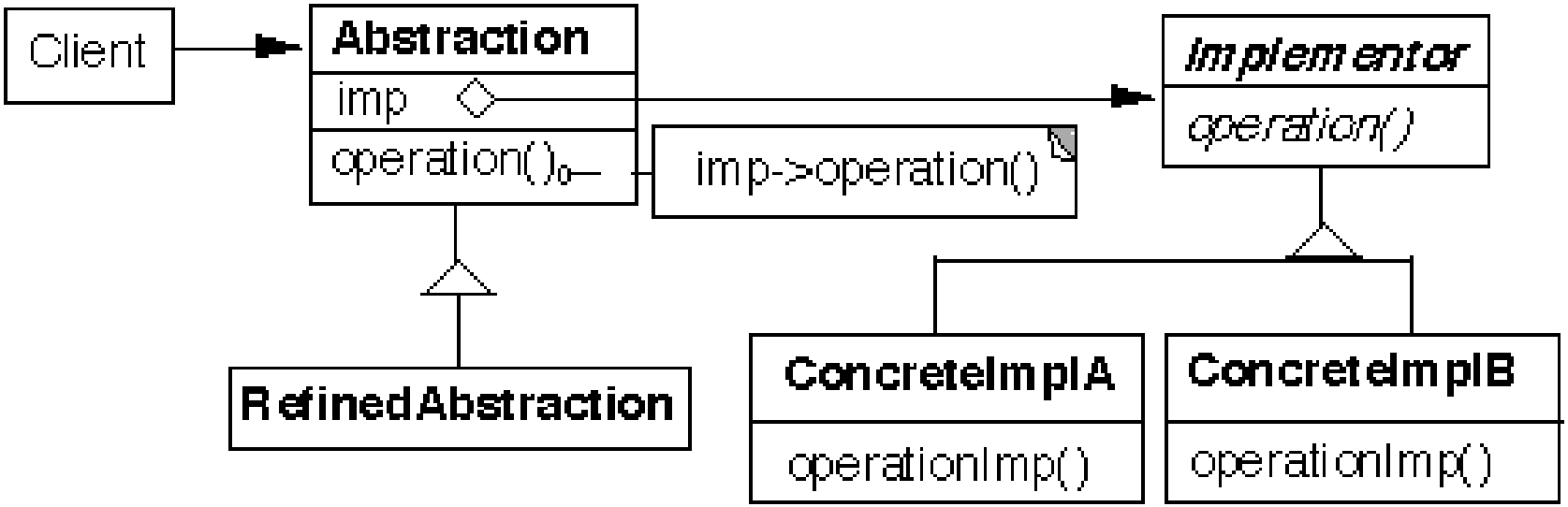
- Adapter converts the interface of a class to another interface that a client expects.
- *Adapter* object implements a *Target* interface and delegates operations to the *Adaptee*.
- *Notify_Event* = target interface
- *Any_Event*, *Structured_Event* = Adapters
- *CORBA::Any*, *CosNotification::Structured Event* = Adaptee

Covered topics

- Command and active object
- Template method and strategy
- Singleton and mono-state
- Null object
- Composite
- Observer
- Abstract factory
- Adapter
- **Bridge**
- Proxy
- Visitor
- State

9. Bridge

- Decouple the abstraction from its implementation
- This allows the implementation to vary from its abstraction
- The abstraction defines and implements the interface
- All operations in the abstraction call method(s) its implementation object

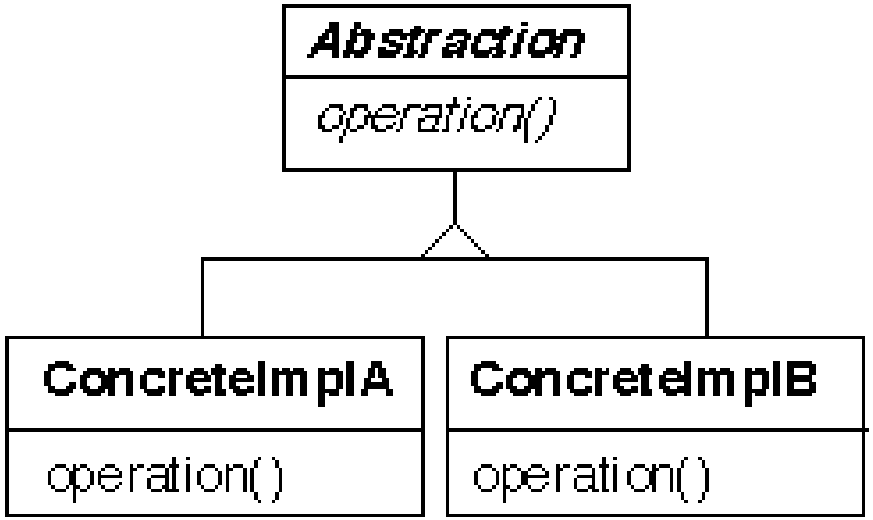


What is Wrong with Using an Interface?

- Make Abstraction a pure abstract class
- In client code:

```
Abstraction widget = new ConcreteImplA();
widget.operation();
```

- This will seperate the abstraction from the implementation
- We can vary the implementation!



Binding between abstraction & implementation

- In the Bridge pattern:
  - An abstraction can use different implementations
  - An implementation can be used in different abstraction
- Hide implementation from clients: Using just an interface the client can cheat!

```
Abstraction widget = new
ConcreteImplA();

widget.operation();

((ConcreteImplA)
widget).concreteOperation();
```

- In the Bridge pattern the client code can not access the implementation
- Java uses Bridge to prevent programmer from accessing platform specific implementations of interface widgets, etc.
- peer = implementation

```
public synchronized void
setCursor(Cursor cursor) {
      this.cursor = cursor;
      ComponentPeer peer = this.peer;
      if (peer != null) {
         peer.setCursor(cursor);
      }
}
```

Applicability

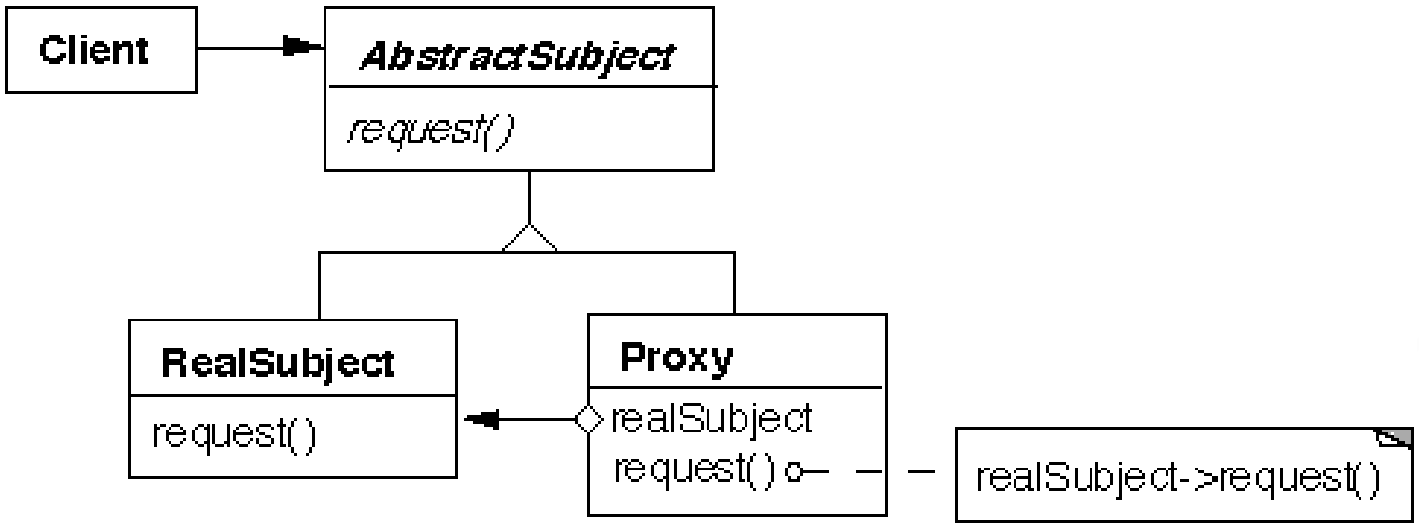
- Use the Bridge pattern when
  - you want to avoid a permanent binding between an abstraction and its implementation
  - both the abstractions and their implementations should be independently extensible by subclassing
  - changes in the implementation of an abstraction should have no impact on the clients; that is, their code should not have to be recompiled
  - you want to hide the implementation of an abstraction completely from clients (users)
  - you want to share an implementation among multiple objects (reference counting), and this fact should be hidden from the client

Covered topics

- Command and active object
- Template method and strategy
- Singleton and mono-state
- Null object
- Composite
- Observer
- Abstract factory
- Adapter
- Bridge
- **Proxy**
- Visitor
- State

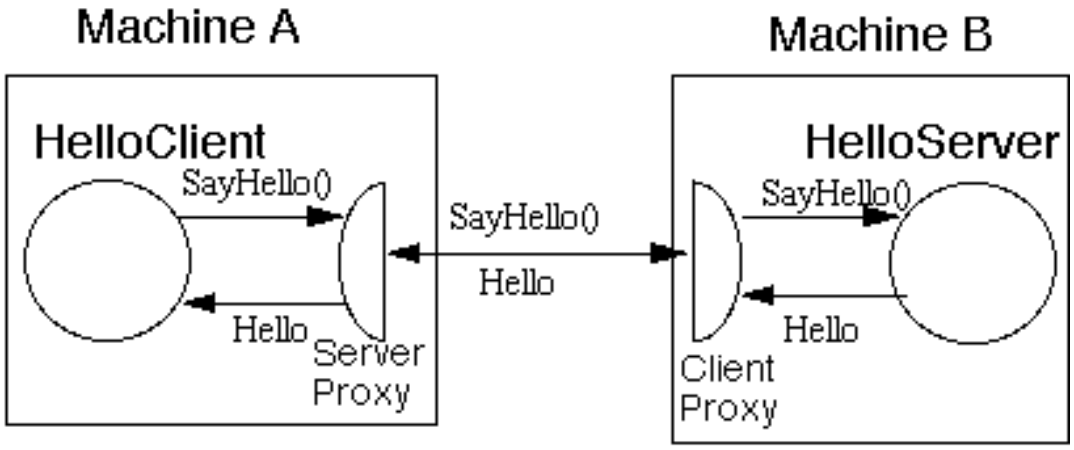
10. Proxy

- The agency for a person who acts as a substitute for another person, authority to act for another
- The proxy has the same interface as the original object
- Use common interface (or abstract class) for both the proxy and original object
- Proxy contains a reference to original object, so proxy can forward requests to the original object



Example: remote proxy

- The actual object is on a remote machine (remote address space)
- Hide real details of accessing the object
- Used in CORBA, Java RMI

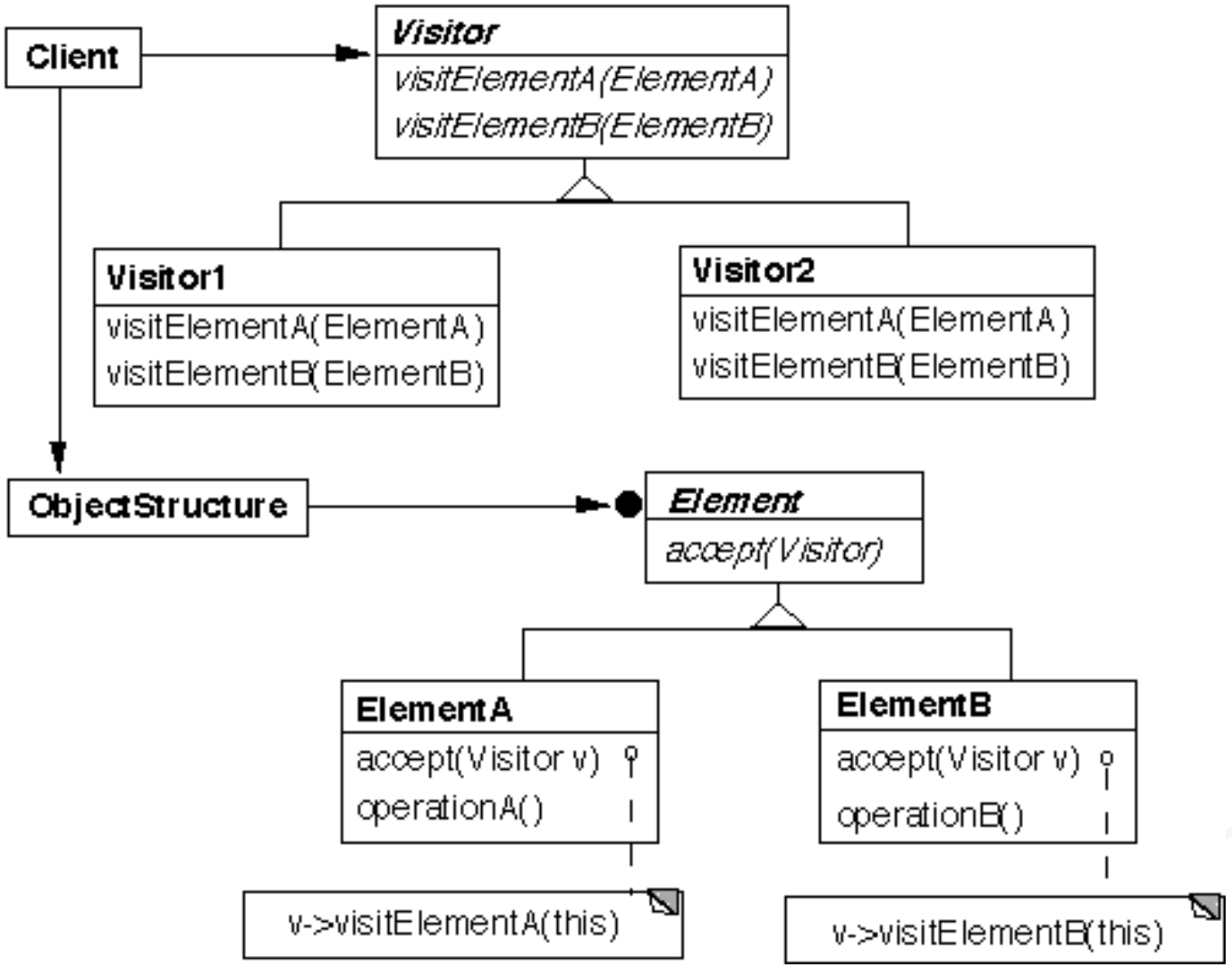



```
public class HelloClient {
    public static void main(String args[]) {
        try { String server = getHelloHostAddress( args);
                Hello proxy = (Hello) Naming.lookup( server
        );
                String message = proxy.sayHello();
                System.out.println( message );
        }
        catch ( Exception error) { error.printStackTrace(); }
    }
}
```

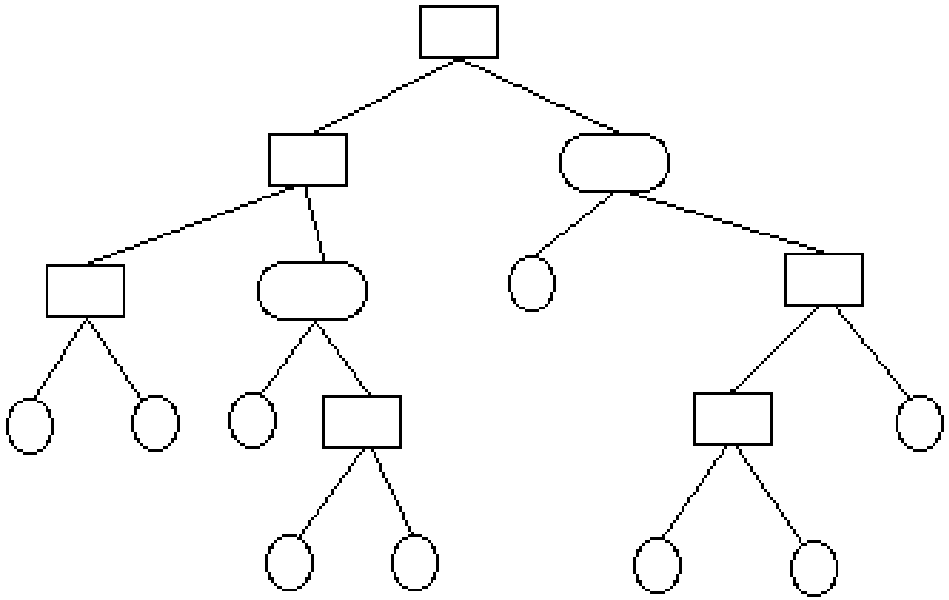
Covered topics

- Command and active object
- Template method and strategy
- Singleton and mono-state
- Null object
- Composite
- Observer
- Abstract factory
- Adapter
- Bridge
- Proxy
- **Visitor**
- State

11. Visitor


Client
Visitor
visitElementA(ElementA)
visitElementB(ElementB)
Visitor1
visitElementA(ElementA)
visitElementB(ElementB)
Visitor2
visitElementA(ElementA)
visitElementB(ElementB)
ObjectStructure
Element
accept(Visitor)
ElementA
accept(Visitor v)
operationA()
ElementB
accept(Visitor v)
operationB()
v->visitElementA(this)
v->visitElementB(this)

Example - Tree Structure



- What about preorder visit, postorder visit?
- What about an HTML print?
- What about printing a 2D representation?
- What if this was an expression tree - evaluation?
- What if this was a binary search tree - adding, deleting?
- What about balancing the binary search tree ?

Solutions

1

- Put operations into separate object (pseudo-visitor)
- Iterate through the structure
- Pass each element in the structure to operations object
- Operation object processes each element
- This helps avoid the clutter of the classes used in the structure. This works in situations where solution 1 does not
- Iteration may be difficult to do, it might be easier to let the structure to the traversal
- When the structure contains objects of different types using just iteration can be a problem
- The iterator has to be able to return different possible unrelated types, which may not be possible (C++) or difficult
- Operation objects can be passed elements it is not designed to handle

2

- Put operations into separate object
- Pass the operations object (visitor) to each element in the structure
- The element then calls the proper method in visitor for its type
- Operation object processes each element properly
- This is more complex than solutions 1 & 2The traversal can be done by the structure, an iterator, or the visitor
- Usually the traversal is done by the structure
- Having the traversal in the visitor can lead to duplicated code
- The visitor is told what type it is acting on, so using the wrong visitor will be a compile error

Applicability

- When an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes
- When many distinct and unrelated operations need to be preformed on objects in an object structure and you want to avoid cluttering the classes with these operations
- When the classes defining the structure rarely change, but you often want to define new operations over the structure

Consequences

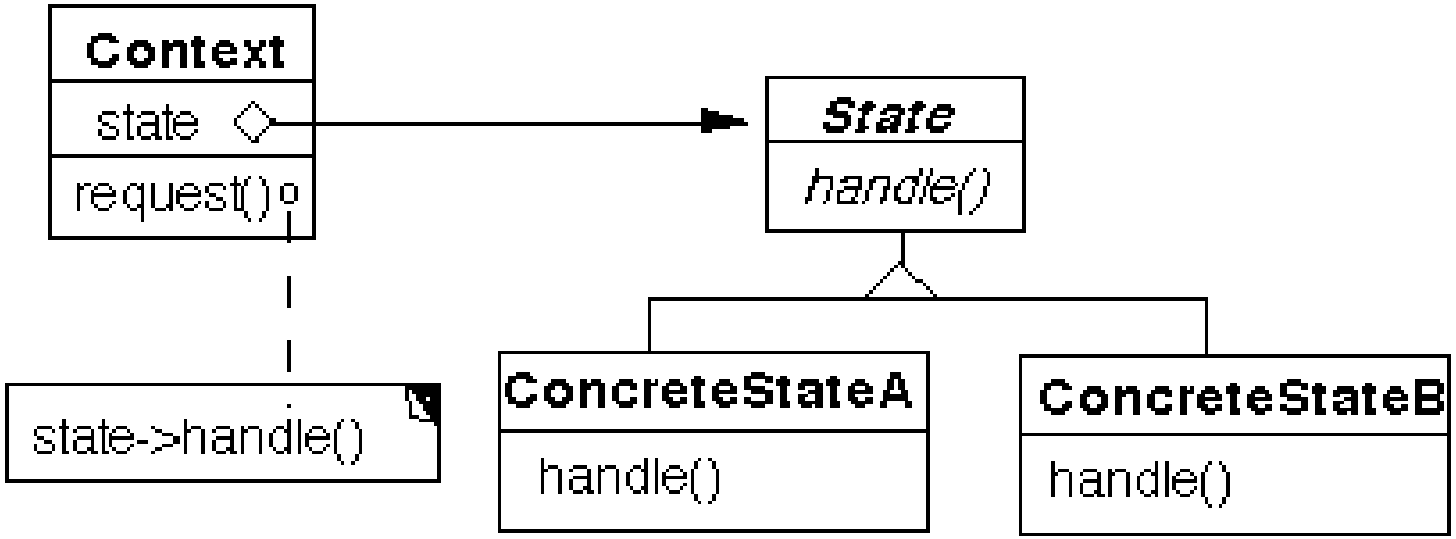
- Visitors makes adding new operations easier
  - If the structure involves many different classes then adding a new operation to the structure requires changing all those classes
- Visitors gathers related operations, separates unrelated ones
  - A print visitor could have all the different ways to print
- Adding new ConcreteElement classes is hard
  - To add a new ConcreteElement you need to change all existing visitors
- Visiting across class hierarchies
  - Text claims iterators can not iterate through structure containing unrelated classes
  - However, the visitor assumes that each element in the structure contains a visit method, which implies at least a common visit interface for all elements
- Accumulating state
  - A visitor can accumulate information as it traverses the structure
- Breaking encapsulation
  - The visitor may force you to provide public operations in the elements that you would not otherwise make public
  - C++ friends are useful here

Covered topics

- Command and active object
- Template method and strategy
- Singleton and mono-state
- Null object
- Composite
- Observer
- Abstract factory
- Adapter
- Bridge
- Proxy
- Visitor
- State

12. State

- Allow an object to alter its behavior when its internal state changes. The object will appear to change it class.

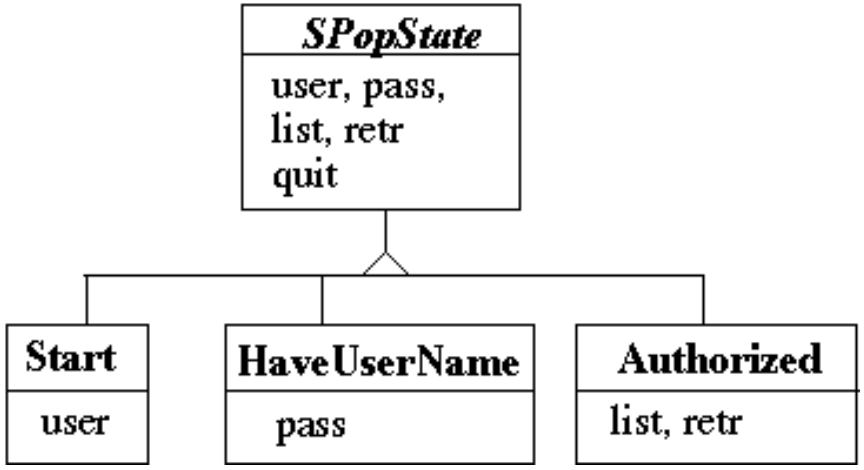


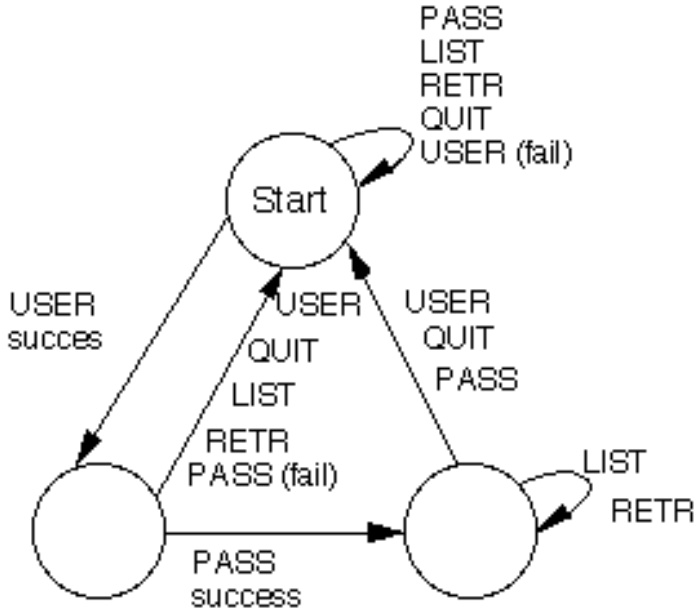
Example : SPOP (Simple Post Office Protocol).
SPOP is used to download e-mail from a server using the following commands

- USER <username>
  - with a username must come first
- PASS <password>
  - with a password
  - come after USER
- QUIT
  - come after USER
  - Updates mail box to reflect transactions taken during the transaction state, then logs user out. If session ends by any method except the QUIT command, the updates are not done
- The user can use the following commands if the username and password are valid:
- LIST :
  - with a message-number (optional)
  - If it contains an optional message number then it returns the size of that message. Otherwise it return size of all mail messages in the mail box
- RETR <message number>:
  - with a message-number
  - The mail message indicated by the number

Example : SPOP (Simple Post Office Protocol)

```
abstract class SPopState {
    public SPopState user( String userName ) {
           //put default action here        }
    public SPopState pass( String password ) {
           //put default action here        }
    public SPopState list( int messageNumber ) {
           //put default action here        }
    public SPopState retr( int messageNumber ) {
           //put default action here        }
    public SPopState quit( ) {
           //put default action here        }
}
```





Example : SPOP (Simple Post Office Protocol)

```
class Start extends SPopState {
     public SPopState user( String userName ) {
            return new HaveUserName( userName );
     }
}
```

```
class SPop {
     private SPopState state = new Start();
     public void user( String userName ) {
            state = state.user( userName );
     }
     public void pass( String password ) {
            state = state.pass( password );
     }
     public void list( int messageNumber ) {
            state = state.list( massageNumber );
     }
     …
}
```

```
class HaveUserName extends SPopState {
     String userName;
     public HaveUserName( String userName ) {
            this.userName = userName;
     }
     public SPopState pass( String password ) {
            if ( validateUser( userName, password )
                        return new Authorized( userName );
            else return new Start();
     }
}
```

Consequence

- It localize state-specific behavior and partitions for different states
- It makes state transitions explicit
- State objects can be shared

Concluding remarks

- Design Patterns provide a foundation for further understanding of:
  - Object-Oriented design
  - Software Architecture
- Understanding patterns can take some time
  - Re-reading them over time helps
  - As does applying them in your own designs!

Quiz and Exercises

- Now let’s go over what you have learned through this lesson by taking a quiz.
- When you’re ready, press Start button to take the quiz