Sinclair Community College

Exploring Core Design Patterns & Frameworks

The Java Developer Education Series

Educate. Collaborate. Accelerate!

Workshop Overview

- This is a three day workshop introducing Object Oriented Design Patterns
- By the end of this course you should understand why Design Patterns are needed, and you will be able to write programs that use the following Patterns in the process of Software Development:
- We will explore the following Creational Design Patterns
  - AbstractFactory (full)
  - Singleton (full)
  - Builder (brief)
  - Factory Method (full)
  - Prototype (brief)
  - Comparison and Summary (Survey of Creational Design Patterns)

Workshop Overview (cont.)

- Structural Design Patterns
  - Composite (full)
  - Adapter (full)
  - Proxy (full)
  - Bridge (brief)
  - Facade (brief)
  - Decorator (brief)
  - Comparison and Summary (Survey of Structural Design Patterns)
- Behavioral Design Patterns
  - Observer (full)
  - Strategy (full)
  - Iterator (full)
  - Visitor (brief)
  - Interpreter (brief)
  - Chain of Responsibility (brief)
  - Command (brief)
  - Mediator (brief)
  - State (brief)
  - Comparison and Summary

Student Pre-Requisites

- A working knowledge of Java™ programming
- A solid understanding of Object Oriented concepts
- A reading knowledge of UML

Student Introductions

- Name
- Company
- Type of developer
- Previous projects of interest
- Current project
- Languages/Areas of expertise
- Why are you taking this workshop?

- My background

Workshop Agenda

- Day 1
  - Session 1 – Introduction to Design Patterns
  - Session 2 – Creational Patterns
- Day 2
  - Session 2 – Creational Patterns (cont.)
  - Session 3 – Structural Patterns
- Day 3
  - Session 3 – Structural Patterns (cont.)
  - Session 4 – Behavioral Patterns
- Day 4
  - Session 4 – Behavioral Patterns (cont.)
  - Session 5 – Patterns and Frameworks

Workshop Structure

Session X

- Lesson A
  - Objectives
  - Lesson Pages
  - Class Review
  - Lesson Summary
  - Lab Exercises (optional)
  - Lab Review (optional)
- Lesson B
  - ...

Session Y

- ...

Workshop Conventions

- Command Prompt Example

```
Prompt> java MyMainClass
```

- Programming Example

```
void setSalary(int newSalary) {
  salary = newSalary; }
```

- HTML / JSP / XML Example

```
<jsp:include page="catalog.jsp"/>
```

- Incorrect code

```
class MyClass {

  public static void main(String args[]) {
    new MyClass(); // this won't work
  }

  MyClass(String someArg) {
  }

}
```

Classroom Environment

- Student workstations
  - Pentium II 128MB Workstation or better
  - Windows NT or Windows 2000
- Software
  - JDK 1.4
  - A suitable text editor or development environment

Workshop Table of Contents

Workshop Table of Contents (cont.)

Sinclair Community College

Questions before we begin ?

Sinclair Community College

Session 1: Introduction to Core Design Patterns

Introduction to Design Patterns

Educate. Collaborate. Accelerate!

Lesson Objectives

- Understand why we use Patterns
- Develop a crisp understanding of Patterns
- Discuss Design Patterns in particular
- Understand the sound design principles implemented by Patterns

The Problem Definition

- Software development can become very complex
  - The crisis is not over yet!
- Disciplined Development Processes:
  - When to perform which task and What is delivered
  - (Possibly) The qualities of deliverables
- How to perform the task is the engineer's responsibility!
- How much of the solution is "invented"?
  - Dejà-vu! Haven't I done this before?
  - Has any one solved this problem before?
- Can knowledge and skill be transferred efficiently and effectively?
  - Common vocabulary of terms
  - Common Semantics of relations

Patterns: Basics

- The meanings of the word Pattern in the English language
  - A decorative design
  - A natural configuration or compound
  - A combination of actions, qualities and idea's that form a consistent or characteristic regime
  - All that is formed or designed in order to serve as a model or suggestion for building things
- First used in the field of Architecture
- A new philosophy and approach to problem solving
- The Quality, The Gate and the Way!

Patterns: Software Community

- Adopted by Software community in 1980's
- Develop a higher level language for communicating and discussing software problems and solutions
- Document and communicate proven and successful practices

Patterns: Definitions

- The Alexandrian Definition:
  - A three part Rule
  - Both the process and the result
- Solves a recurring Problem
- Concept behind the solution is proven
- Provides generic solutions
- Describes basic components of the solution
  - Their associations
  - Their collaborations
- The generated result is both functional and appealing (the quality!)

Patterns: Crucial Elements

- An Element is an “attribute” of a Pattern
- Pattern Name:
  - Uniquely identifies pattern
- Context:
  - A specific situation where the problem may occur
  - It determines the relevance of the problem

Patterns: Problem

- A recurring set of forces
- Operational nature
- Determine the requirements to the solution
- Could be Contradictory

Patterns: Solution

- Solutions are generic
- Describes how the forces may be balanced
- Has two major parts
  - Static Structure
  - Dynamic Behavior

Patterns: Crucial Qualities (1)

- Patterns focus on recurring problems in a specific context, and present a generic solution
- Patterns facilitate the documentation and communication of proven software architecture and design
- Patterns implement sound architectural and design principles
- Patterns identify and specify abstractions that surpass single classes and components

Patterns: Crucial Qualities (2)

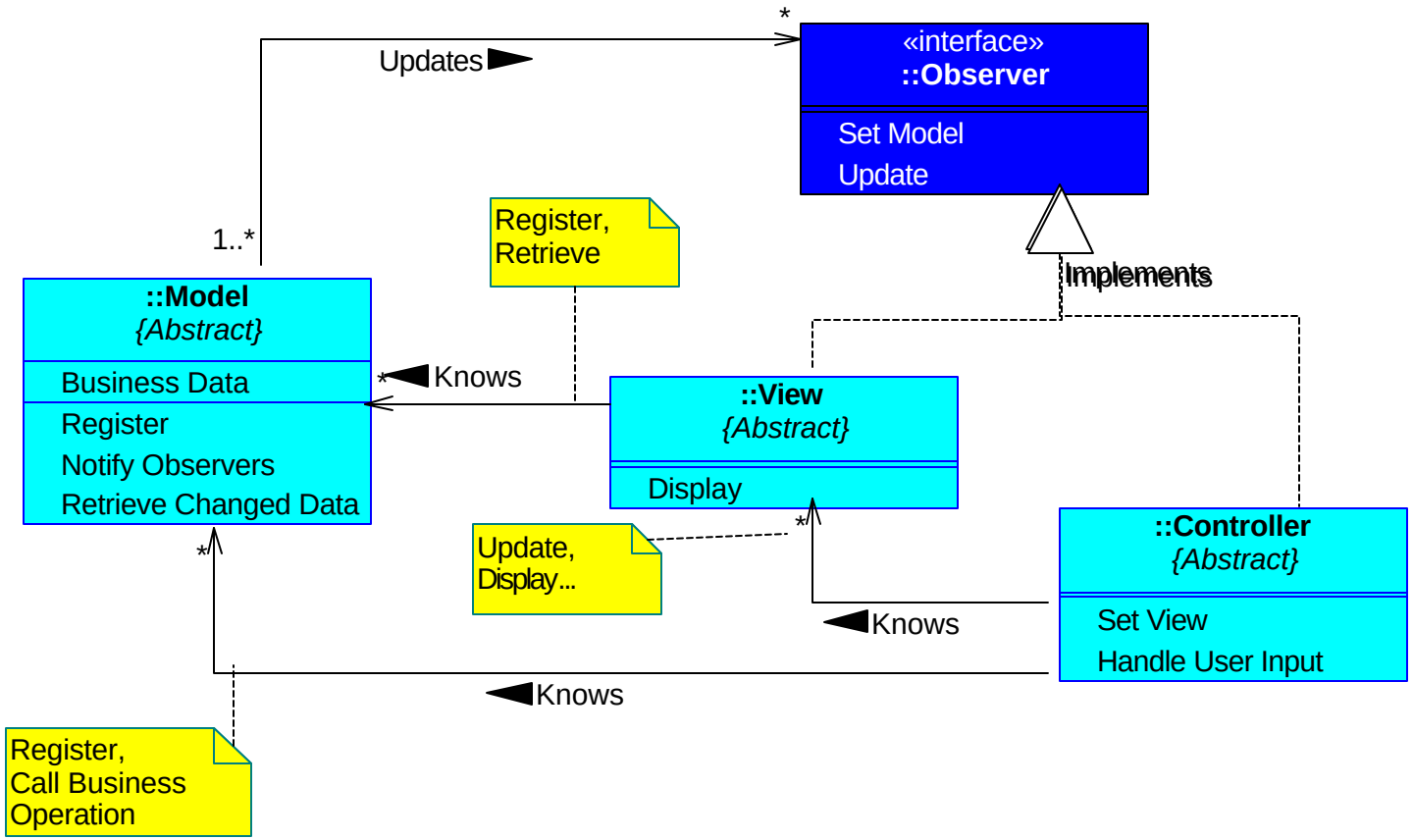
- Patterns provide a paradigm for documenting software development tasks and deliverables
- Patterns provide a common understanding, terminology and semantics for discussion and communication of insight, idea's and solutions
- Patterns provide us with an alternative paradigm for solving software development problems:
  - Thinking in problem-solution pairs in specific contexts
  - Higher abstraction level, therefore easier to recognize, understand and apply than e.g. code
  - More concrete than design principles

Patterns: Example (1)

- Context: Highly Interactive Applications with UI
- Main Problem:
  - User Interfaces are modified often
  - A modification in the UI must not affect other modules
- Secondary Forces
  - Consistency of data presentation
  - Visibility of changes in the state of business data
  - Modifying UI must be easy
  - Platform changes must not affect business logic

Patterns: Example (2)

- Solution: Partition the application in three different stereotypes of modules
  - **Model**: Encapsulates core business logic and data
  - **View:** Presentation of the output of Model
  - **Controller**: Handles input for the Model



Patterns: Example (3)

- The MVC Pattern has all crucial qualities of a Pattern:
- It addresses a recurring problem; Changing UI without affecting the functional core
- The context for this problem is “interactive applications”

Patterns: Example (4)

- The solution is to realize decoupling between UI and Business Logic
- The approach has been used in SmallTalk, Swing, AWT, MFC
- There are three collaborating components, each with its own responsibilities The term “MVC” is commonly known by many. The term MVC alone is enough to communicate many idea’s, concept and solutions

Patterns: What is Not a Pattern?

- A solution for a problem in a context
  - Recurrence of the problem is a must
- Professional Jargon
  - Patterns are more than terms and phrases
- Data Structures
  - Patterns describe an architecture including behavior
- Algorithms
  - Patterns are more generic, and address software problems
- Code re-usability techniques
  - The emphasis on architecture and design
- Complete Solution
  - Patterns are used to generate solutions

Patterns: Relating to Methodologies

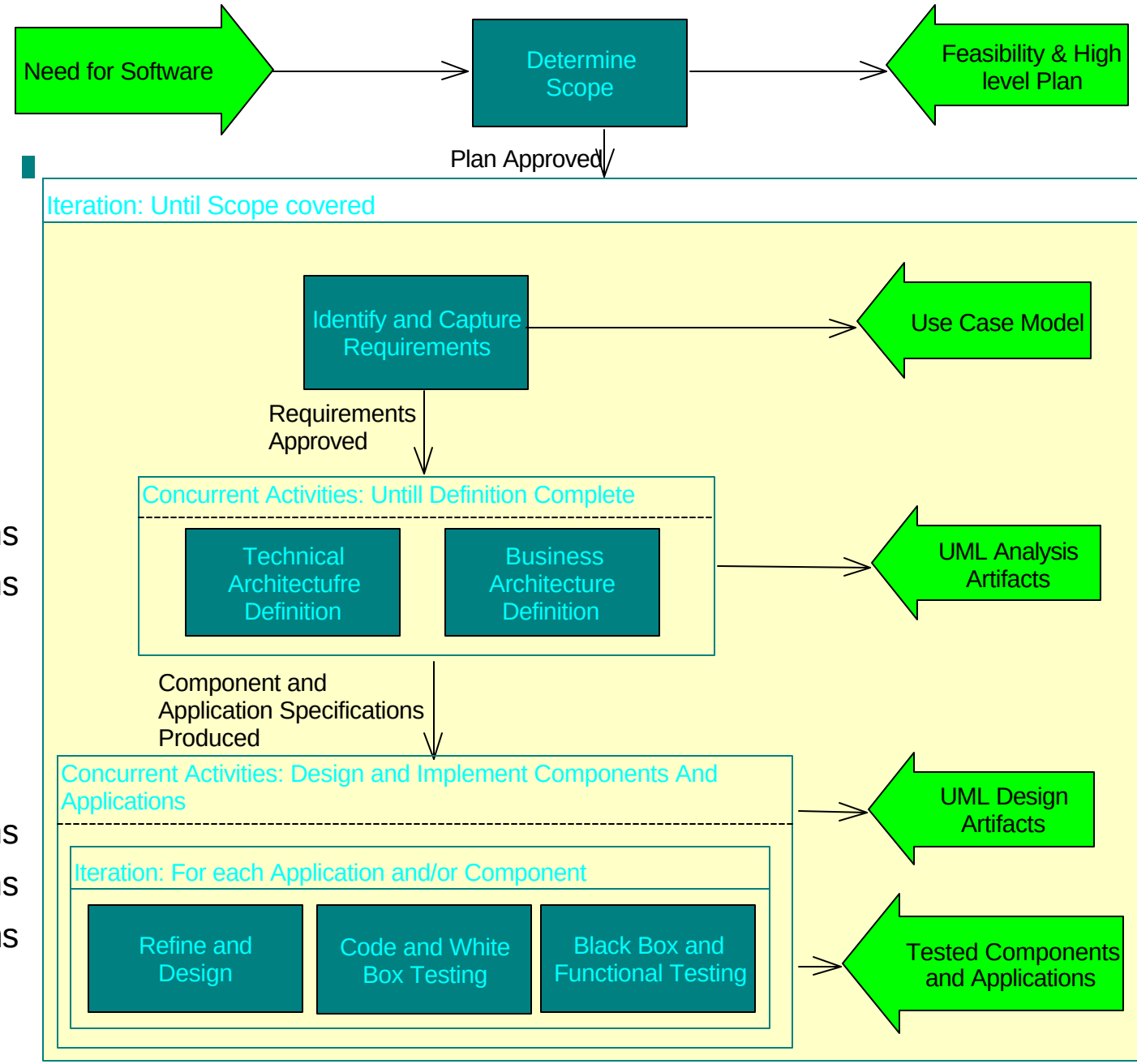
- It is essential to use the right methodology
  - A well defined and complete transformation process
  - Must Allow iterative development & incremental delivery
- Methods are Generic, Abstract and cover multiple phases
  - They address WHAT need to be done and when
- Patterns Complement Methodologies
  - by telling us HOW to perform specific Analysis, Design and Implementation Tasks
  - In order to generate high quality software

Patterns: Aligned with Process

Analysis Patterns

Analysis Patterns
Architectural Patterns

Architectural Patterns
Design Patterns
Idioms



Design Patterns

- Tactical, medium-level and domain-independent
  - Must not affect the domain model or the high-level architecture
- Programming Language Independent
  - Must not impose the use of a specific language or environment
- Focus on non-functional forces
  - The resulting solutions addresses issues such as adaptability, changeability and issues regarding time & space
- Applicable during (detailed) design
  - Smaller scale than Architectural Patterns
- Used for refining coarse-grained component
  - Internal design of bigger components

Design Patterns: Classification Criteria

- Classified by two criteria:
- Purpose
  - The Type of Design Problem
  - Creational
  - Structural
  - Behavioral
- Scope:
  - Class : Static Problems (compile time)
  - Instance: Dynamic in runtime

Design Patterns: Creational

- Hiding Complexity of object creation
  - Instantiation of objects
  - Initialisation of created objects
  - Configuration
- Unknown classes
  - In compile time
- Controlling Number of instances

Design Patterns: Structural

- Composition of classes and objects
- Variant implementations
- Extending functionality through composition
- Enable sharing of instances

Design Patterns: Behavioral

- The address issues regarding Complex, variant run-time behavior:
  - Delegation of responsibility
  - Undo, redo
  - Notification
  - State management
  - Variant algorithms
  - Layering responsibility

Design Patterns: Scope Criteria

- The Scope of a Pattern
  - Leveraging use of Static and Dynamic relationships
- Class
  - Inheritance
  - Associations
  - Aggregation by value
- Object
  - Varying object relationships
  - Varying Object Behavior

Design Patterns: Description

- Pattern Name
  - Mandatory
  - the name of the pattern by which it is referred to
- Pattern Summary
  - Mandatory
  - Briefly describes what it does
- Also known as
  - Optional
- Illustrative Example
  - Optional, although highly desired
  - A real-world example demonstrating the need for this pattern

Design Patterns: Description (cont.)

- Context
  - Mandatory
  - The specific situation in which the Problems may rise
- Problem
  - Mandatory
  - A full description of the generic problem and the forces that will influence the Solution

Design Patterns: Description (cont.)

- The Solution:
  - Mandatory
- Abstract Approach
- A textual description of the generic approach
- A static model
- A dynamic model of the behavior of the pattern

Design Patterns: Description (cont.)

- Implementation
  - Optional, But highly Desirable
  - A step wise description of how to implement the solution, accompanied by code examples
- Alternative
  - Optional
  - Specialized or Variant scenario's

Design Patterns: Description (cont.)

- Known uses
  - Optional
  - Real world examples of where the pattern has been applied
- Consequences
  - Optional, but highly desirable
  - Lists and discusses the benefits and liabilities of applying the pattern
- Related Patterns
  - Optional
  - Lists and discusses other patterns that may be used in combination with this pattern.

Design Patterns: Design Principles

- Design Patterns implement many proven design principles. Some of these are:
  - Abstraction of a problem into its essence, thus allowing us to cope with complexity
  - Encapsulates related behavior and data in the solution
  - Design for flexibility
  - Separation of Interface and Implementation
  - Composition and Aggregation
  - Responsibility Driven Design
  - Coupling
  - Cohesion

Design Patterns: The Selection Process

- Formulate the problem:
  - General description
  - The individual Forces
  - Other operational Aspects
- Identify the Category of Design Patterns
  - Assuming it is a design problem
  - Use Design Patterns Classifications
- Identify the Candidate Pattern
  - Start with Summary
  - Then compare your problem description with that of the pattern
  - Use the provided examples
- Study the consequences
  - This will determine the implementation
- Identify possible other required Patterns

Lesson Summary

- Developing software is hard, developing quality software is harder;
- The “not invented here” syndrome
- Patterns document successful solutions to known recurring problems, in a specific context
- The Crucial elements and Qualities of Patterns
- There are various types of Patterns
- Design Patterns are tactical and language independent

Sinclair Community College

Session 1: Exploring Creational Design Patterns

AbstractFactory Design Pattern
Singleton Design Pattern
Builder Design Pattern
Factory Method Design Pattern
Prototype Design Pattern
Survey of Creational Design Patterns

Educate. Collaborate. Accelerate!

Sinclair Community College

Lesson 1: AbstractFactory Design Pattern

AbstractFactory Design Pattern

Singleton Design Pattern

Builder Design Pattern

Factory Method Design Pattern

Prototype Design Pattern

Survey of Creational Design Patterns

Educate. Collaborate. Accelerate!

Lesson Objectives

- Introduce and Explain `AbstractFactory` in Details
- write systems independent of how its objects are created
- make design decisions based on the advantages and disadvantages of the `AbstractFactory` design pattern

AbstractFactory: Introduction

- Summary
  - provides an interface for creating families of related or dependent objects without specifying their concrete classes
  - allows a program to work with a variety of complex external entities such as different windowing systems with similar functionality.
- Also Known As
  - Kit

AbstractFactory: Illustrative Example

- Toasters work the same way across the globe
  - They have the same interface and are used the same way
- However …
  - When you buy a toaster in Los Angeles it has a US three prong plug and draws 110 volts
  - When you buy a toaster in Paris, France it has a different three prong plug and draws 240 volts
  - These toasters are not interchangeable
  - They do serve the same purpose and have the same interface
- Who makes this decision?
- What would it be like if the decision was not made for you?

AbstractFactory: Context

- We have a series of dependant objects that can appear in two or more different styles. For example, GUI components may have a Border or an Icon.
- There could be a different "Look and Feel" for a GUI.
- Used in windowing systems where "Look and Feel" may need to change
- Suitable for GUI systems that must be portable

AbstractFactory: Problem

- Consider the following example from a GUI program:
- Each class represent a Button component with a different "look and feel"

```
class Button_WinLAF {
  public Button_WinLAF() {
    System.out.println("New WinLAF Button";
  }
}

class Button_MacLAF {
  public Button_MacLAF() {
    System.out.println("New Button with Icon");
  }
}
```

AbstractFactory: Problem (cont.)

- Each class represent a different style of Choice component

```
class Choice_WinLAF {
  public Choice_WinLAF() {
    System.out.println("New WinLAF Choice");

  }
}
class Choice_MacLAF {
  public Choice_MacLAF() {
    System.out.println("New MacLAF Choice with Border");

  }
}
```

AbstractFactory: Problem (cont.)

- create() method creates the appropriate objects

```
class CreateObjects {
  public static void create() {
    Button_WinLAF buttonB = new Button_WinLAF();
    Choice_WinLAF choiceB = new Choice_WinLAF();
  }
}

class AbstractFactory_example1 {
  public static void main(String argv[]) {
    CreateObjects.create();
  }
}
```

AbstractFactory: Problem (cont.)

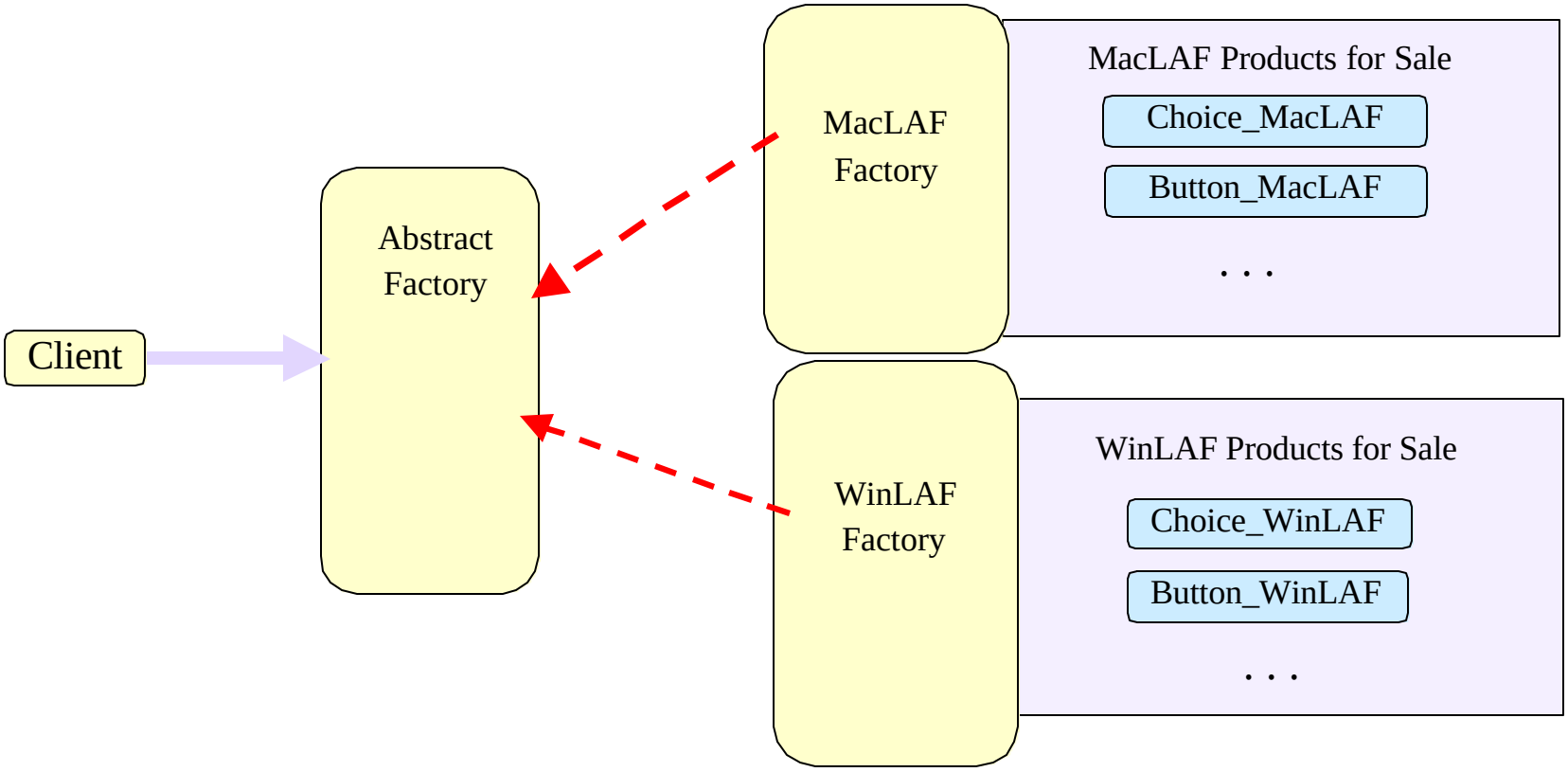
- Here we have a definition of two types of products, Button and Choice and two different styles, WinLAF and MacLAF
- Method `create()` is responsible for instantiating two WinLAF objects
- Imagine that this method `create()` now creates and configures dozens of WinLAF objects

AbstractFactory: Problem (cont.)

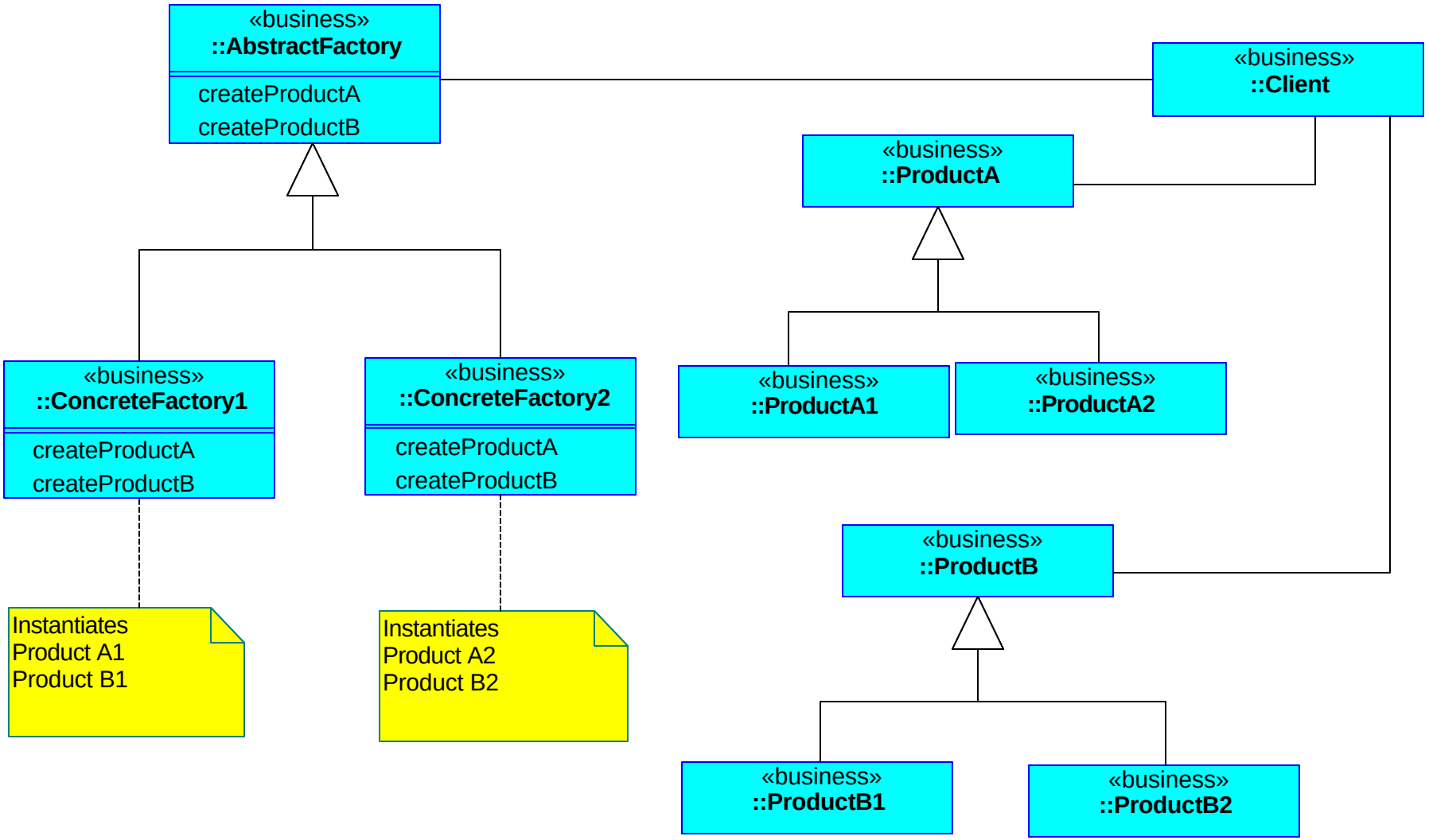
- Consider that we want to do the same for MacLAF objects
- We want to create similar objects but for a different "look and feel"
- We can either re-write `create()` function
- AbstractFactory design pattern allows us to
  - Write function `create()` to be independent of the "look and feel"
  - Make code more flexible

AbstractFactory: Solution Description

- Client only sees an `AbstractFactory`
- Which actual factory it is (MacLAF or WinLAF) is immaterial to the client



AbstractFactory: Structure



AbstractFactory: Structure (cont.)

- AbstractFactory (Factory) defines an interface for creating products
- ConcreteFactory (FactoryWinLAF and FactoryMacLAF) implements all of the operations defined in the AbstractFactory class.
- AbstractProduct (Button and Choice) declares an interface for each type of the product
- ConcreteProduct (four product classes) implements the AbstractProduct interface for all different products and styles
- Client (CreateObjects) uses interfaces defined by the AbstractFactory class to create objects

`AbstractFactory`: Implementation

- The diagram represents the design of the `AbstractFactory`
- For each product we define an abstract superclass
- Each abstract superclass has two or more concrete subclasses for each different style of the product
- WinLAF and MacLAF in this case
- The factory classes are used to create all of the objects in a "look and feel" independent manner

AbstractFactory: Implementation (cont.)

- How does the factory know which of the concrete factory classes to actually implement?
- Client can inform the factory when it is created
- Factory can make its decision for the client based on some other information

AbstractFactory: Code Example

- The following is the source code for this design pattern

```
abstract class Button {

}

class Button_WinLAF extends Button {
  public Button_WinLAF() {
    System.out.println("New Button WinLAF");
  }
}

class Button_MacLAF extends Button {
  public Button_MacLAF() {
    System.out.println("New Button MacLAF");
  }
}
```

AbstractFactory: Example Code (cont.)

```
abstract class Choice {

}

class Choice_WinLAF extends Choice {
  public Choice_WinLAF() {
    System.out.println("New Choice WinLAF");
  }
}

class Choice_MacLAF extends Choice {
  public Choice_MacLAF() {
    System.out.println("New Choice MacLAF");
  }
}
```

AbstractFactory: Example Code (cont.)

```
abstract class Factory {
  abstract Button createButton();
  abstract Choice createChoice();
}

class FactoryWinLAF extends Factory {
  Button createButton() { return new Button_WinLAF(); }
  Choice createChoice() { return new Choice_WinLAF(); }
}


class FactoryMacLAF extends Factory {
  Button createButton() { return new Button_MacLAF(); }
  Choice createChoice() { return new Choice_MacLAF(); }
}
```

AbstractFactory: Example Code (cont.)

```
class CreateObjects{
  public static void create(Factory factory) {
    Button button = factory.createButton();
    Choice choice = factory.createChoice();
  }
}

class AbstractFactory_example2 {
  public static void main (String argv[]) {
    FactoryWinLAF factoryWinLAF = new FactoryWinLAF();
    FactoryMacLAF factoryMacLAF = new FactoryMacLAF();

    CreateObjects.create(factoryWinLAF);
    CreateObjects.create(factoryMacLAF);
  }
}
```

AbstractFactory: Consequences

- Benefits
  - The system is independent of how the objects are created
  - The system can easily be changed to be configured with a different style (family) of objects
  - Abstract classes can be grouped together to provide a library that reveals interfaces but not implementation
  - Enforces that client uses objects from only one family of objects at a time
- Liabilities
  - It is not well suited for adding new types of products since this involves changing the interface of the `AbstractFactory` class and all of its subclasses
  - To work well all the methods must be polymorphic

AbstractFactory: Relating Patterns

- You can use a Singleton Design pattern to ensure that only one object of each concrete factory exists
- Factory Method

AbstractFactory: Known Uses

- java.awt.Toolkit
- java.io.Socket
- Java Foundation Classes

Class Review

- `AbstractFactory` provides a level of abstraction for instantiating objects. Is this true?
- There is a one to one relationship between each `ConcreteFactory` and a product style. Is this true?
- The client is responsible for the instantiation of Product objects. Is this true?
- Adding new products to the implementation is easy (i.e. Product C). Is this true?

Lesson Summary

- The `AbstractFactory` pattern allows you to change the type of product you create
- The type of product you create does not affect the application code
- All product types must implement the same inheritance tree
- `AbstractFactory` allows the product type to be changed at run time
- `AbstractFactory` is easily extendable for new "look and feel" variations

Exercise 1 – `AbstractFactory` Pattern

- Follow the instructions in the Student Guide

Sinclair Community College

Lesson 2: Singleton Design Pattern

AbstractFactory Design Pattern

Singleton Design Pattern

Builder Design Pattern

Factory Method Design Pattern

Prototype Design Pattern

Survey of Creational Design Patterns

Educate. Collaborate. Accelerate!

Lesson Objectives

- Become familiar with the `Singleton` Design Pattern
- Learn how to ensure that one and only one instance of a class Can ever be created

Singleton: Introduction

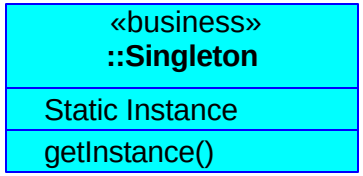
- Summary:
  - Ensures that there is one and only one instance of a class in existence at any point in time
  - All clients see only that instance

Singleton: Context

- We want to ensure that only one instance of the class can be created
- The Problem:
  - consider the `AbstractFactory` design pattern
  - only one instance of each of the Concrete Factories is needed
  - each concrete factory class is a perfect target for a Singleton design pattern

Singleton: Solution

- Make the Class responsible
  - For managing the number of its instances
- Define an Instance operation
- This operation is the sole point of accessing the instances of the class for all clients
- Structure



- Instance is a static class method

Singleton: Implementation

- The Singleton class must provide a static method to create the single instance (p.e. getInstance())
- This makes the class responsible for its own creation
- The Singleton object may not always be needed. The actual creation of the object can be delayed until the first call
- Dynamic loading of objects that contain references to a Singleton may incorrectly create more than one instance of the Singleton
- Client classes should not have responsibility of creating the Singleton object

Singleton: Code Example

```
class TooManyObjects extends Exception {

  public TooManyObjects() {
    super();
  }
}

class Singleton {
  public Singleton() throws TooManyObjects {
    if (numObjects > 0)
      throw new TooManyObjects();
    }
    numObjects++;

  }
  private static int numObjects = 0;
}
```

Singleton: Example 1 (cont.)

```
public class Singleton_version1 {

  public static void main(String argv[]) {
    try {
      Singleton a = new Singleton();
    }
    catch (TooManyObjects e){
      System.out.println("Exception in first try block.");
    }
    try {
      Singleton b = new Singleton();
    }
    catch (TooManyObjects e) {
      System.out.println("Exception in second try block.");
    }
  }
}
```

Singleton: Example 1 (cont.)

- Example: The static variable `numObjects` keeps track of how many objects are created
- It is declared static so that it exists before any objects of class Singleton are instantiated
- This class throws an exception if a program attempts to create more then one instance of the class
- An exception is caught in the second try-catch block when the second instance of the class `Singleton` is created

Exercise 2 – Instance counting

- Follow the instructions in the Student Guide

Singleton: Example 2 (cont.)

- Code Example 2

```
class Singleton {

  private static Singleton ptr = null;

  public static Singleton instance() {
    if (ptr == null) ptr = new Singleton();
    System.out.println("Returning an instance");
    return ptr;
  }

  private Singleton() {
    System.out.println("Creating an instance");
  }
}

public class Singleton_version2{
  public static void main(String argv[]){
    Singleton a = Singleton.instance();
    Singleton b = Singleton.instance();
  }
}
```

Singleton: Example 2 (cont.)

- In this example a user is not allowed to create an object of the `Singleton` class directly
- This is achieved by making the constructor of the `Singleton` class private
- A user gets an object of the class `Singleton` by invoking `instance()` method. This method is responsible for the creation of the object. `instance()` calls the `private` constructor of the class.

Exercise 3 – Permanent instance

- Follow the instructions in the Student Guide

Singleton: Consequences

- Benefits
  - A Singleton Design Pattern is superior to global variables
  - It provides single point of access to instances
  - It allows configurable number of instances
- Liabilities
  - Imperfect encapsulation
  - Subclassing the Singleton class may cause multiple problems

Singleton: Supplementary Information

- Alternatives Scenarios
  - More than one instance
- Relating Patterns:
  - `AbstractFactory`, `Builder` and `Prototype` all use `Singleton` to ensure that only one instance of them may exist.

Class Review

- A `Singleton Design Pattern` must use `static` elements to accomplish its goal. Is this true?
- A `Singleton Design Pattern` lets you control how many objects of the class are created. Is this true?
- Think of 4 situations where a `Singleton Design Pattern` can be used.
- Compare the implementation examples 1 and 2. What are the advantages and the disadvantages of each one?

Lesson Summary

- The `Singleton` Design Pattern is used to ensure that only one object can be created at any given time
- The Concrete Factory class from the `AbstractFactory` Design Pattern is an example of a situation where a `Singleton` can be useful

Sinclair Community College

Lesson 3: Builder Design Pattern

AbstractFactory Design Pattern
Singleton Design Pattern
Builder Design Pattern
Factory Method Design Pattern
Prototype Design Pattern
Survey of Creational Design Patterns

Educate. Collaborate. Accelerate!

Lesson Objectives

- Introduce and Explain the `Builder` Pattern briefly
- Demonstrate how the process of creating (complex) objects can be separated from their constituent components

Builder: Introduction

- Summary
  - Separates the construction of a complex object from its representation
  - Allows the client to create an object by specifying only its type and content
  - Allows a flexible process of creating a set of related objects

Builder: Context

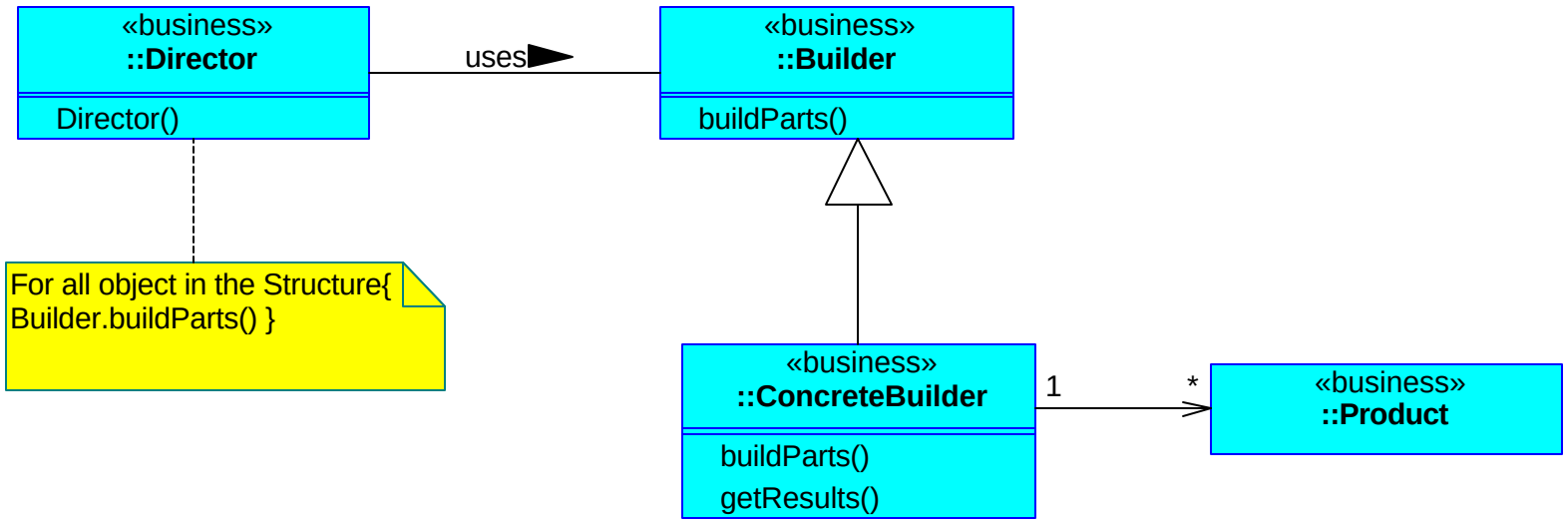
- The client must not know how the complex product is created
- The algorithm for creating a complex product must be decoupled from the concrete parts that are used to build the product
  - `Builder` is often used to produce multiple representations of the same data, e.g. - an email message

Builder: The Problem

- Many different objects may be required
- The order in which they are required can vary
- Classes that build external data representations need to be independent of classes that provides the content

Builder: The Solution

- Separate “How” to build complex Objects from “what” they are made of.
- Class Diagram

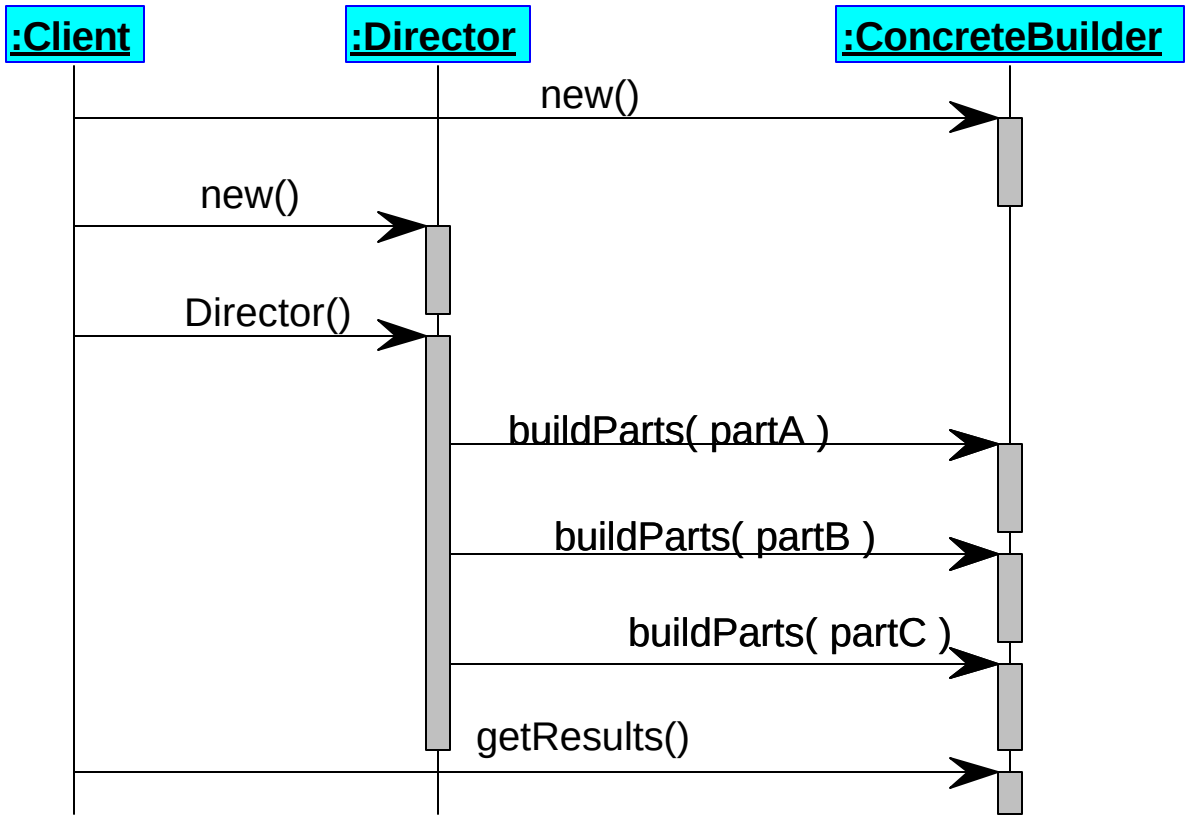


Builder Pattern: Static Structure (cont.)

- Builder
  - Defines an abstract interface for instantiating all parts of the product
- ConcreteBuilder
  - Implements the abstract interface defined by the Builder
  - Creates parts of the final Product object
  - Keeps track of the representation (Product) that it creates
  - Implements the interface to retrieve the final product
- Director
  - Uses Builder interface to construct an object - one part at a time
- Product
  - Represents the object we are trying to build

Builder Pattern: Dynamic Model

- UML Object Sequence Diagram for the `Builder` pattern



Builder: Implementation

- Create an abstract `Builder`
  - To allow various concrete `Builder` classes to be added
  - `ConcreteBuilder` contains all of the code necessary to assemble a `Product`
- Define one operation for each concrete object
  - They are all abstract
- Sub-class `AbstractBuilder` for each concrete component that may be required
- Design the `Director` so that it completely encapsulates the building algorithm
- Consider the requirements for returning and assembling objects
  - Appending, Inserting

Builder: Consequences

- Benefits:
  - Lets you vary the product's internal representation
  - The `Director` does not need to know how the `Product` gets assembled
  - The code for construction is isolated from the code for representation
  - Gives you fine control of the construction instead of just returning every thing at once

Builder: Supplementary Information

- Alternative Scenario
  - Different `Directors` can be used to build different `Products` using the same `ConcreteBuilder`
- Known uses
  - Parsers
  - Tokenizers
- Related Patterns
  - Similar to `AbstractFactory` in that they both return complex objects, but different because `Builder` returns parts of the object incrementally
  - `Composite` pattern is used to store the returned objects

Lesson Summary

- Builder Pattern: separates the creation of complex objects from the attributing parts and the way they are assembled
- This separation results in two collaborating objects:
  - The Director: Knows the process
  - The Builder: Builds one component of the complex objects

Sinclair Community College

Lesson 4: Factory Method Design Pattern

AbstractFactory Design Pattern
Singleton Design Pattern
Builder Design Pattern
Factory Method Design Pattern
Prototype Design Pattern
Survey of Creational Design Patterns

Lesson Objectives

- Introduce the `Factory Method`
- Demonstrate how to dynamically create objects without knowing their concrete classes

Factory Method: Introduction

- Summary
  - Allow concrete subclasses of an abstract class determine which classes to instantiate, but use the same interface defined in the abstract super-class.
- Also Known as
  - Virtual Constructor

Factory Method: Context

- A class can't anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to know which helper subclass is the delegate.

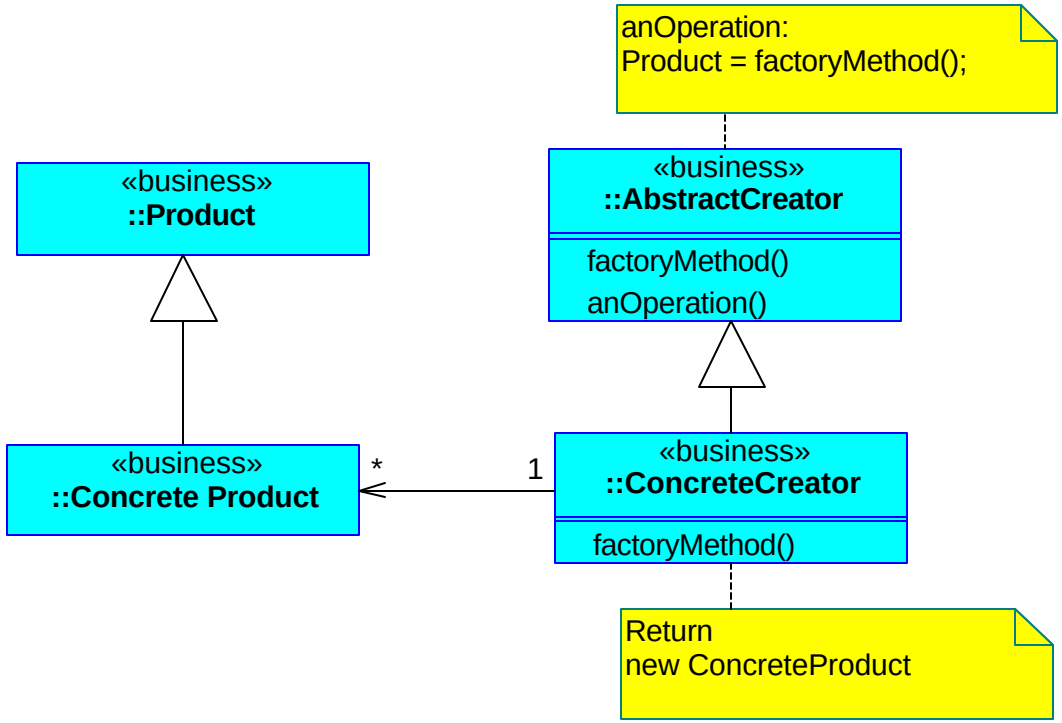
Factory Method: Problem

- A Client knows when it needs an object
  - In the JFC a `Swing` component needs to install or change a specific presentation for itself
- A Client does not know the specific class
  - A Swing component only knows the abstract class `UIManager`
- A Client can only provide information about itself
  - The `UIManager` uses this information to eventually install the default presentation for the component.

Factory Method: Solution

- Enforce the same interface for all clients
- Create an abstraction that decides which of the possible (pre-determined) sub-classes to instantiate and return.

Factory Method: Solution



Factory Method: Components

- Product
  - Defines the interface that a ConceteProduct must implement
  - Is returned by the Factory Method of the Creator
- ConcreteProduct
  - Implements the Product interface
- Creator
  - Defines the Factory Method. This method returns an object of type Product.
  - May also implement the default Factory Method
- ConcreteCreator
  - May override the default Factory Method to return an instance of the ConcreteProduct

Factory Method: Implementation

- Concrete objects need to be created in a common way
- Lets a class defer instantiation to its subclasses
- Subclasses decide which class to instantiate a reusable class Instantiates other classes
- The reusable class remains independent of the classes it Instantiates
- Delegates the choice to another object that supports a Common interface
- Based on parameters, it chooses which kind of concrete Object to create

Factory Method: Implementation (cont.)

- Factory methods deal only with an interface defined by the abstract `Creator` class
- A `ConcreteCreator` for each concrete product
- Creating products within a class is more powerful than direct instantiation
- The `Factory Method` can be extended to provide an extended version of the product

Factory Method: Implementation (cont.)

- Parameterized factory methods
- Allows a single factory method to create multiple concrete Products
- Factory Method accepts a parameter that identifies which Concrete product to instantiate

Factory Method: Consequences

- Benefits
  - provides hooks for subclasses
  - more flexible than creating objects directly
  - connects parallel hierarchies of `Creator` and `Product`
  - allows classes to create objects in a separate operation so that subclasses can override them
  - allows factories to create multiple kinds of products
  - the user does not have to hardcode application specific classes

Factory Method: Supplementary Info

- Alternative implementations:
  - two types of Factory Method exist:
  - Creator is an abstract class
  - Creator provides a default implementation
- Known uses
  - Various GUI Frameworks, Java RMI, …
- Relating Patterns:
  - AbstractFactory is implemented using Factory Method
  - Template pattern uses the Factory Method

Lesson Summary

- The Factory Method Pattern provides a high level mechanism for returning instances of a predetermined set of classes
- The Factory Method Pattern uses client context information to determine the concrete product

Sinclair Community College

Lesson 5: Prototype Design Pattern

AbstractFactory Design Pattern
Singleton Design Pattern
Builder Design Pattern
Factory Method Design Pattern
Prototype Design Pattern
Survey of Creational Design Patterns

Educate. Collaborate. Accelerate!

Lesson Objectives

- Introduce and explain `Prototype` Pattern briefly
- Demonstrate how we can reduce complexity by cloning objects and reusing their data and state information, rather than creating new objects

Prototype: Introduction

- Summary
  - Instead of creating new objects or subclasses for similar tasks or creating sets of data, copy or clone the relevant existing instances and re-use their contents in order to perform tasks or create new objects if necessary.

Prototype: Context

- Potentially many similar objects are required to perform similar tasks
- The data and state of each potential object are equally important to the application

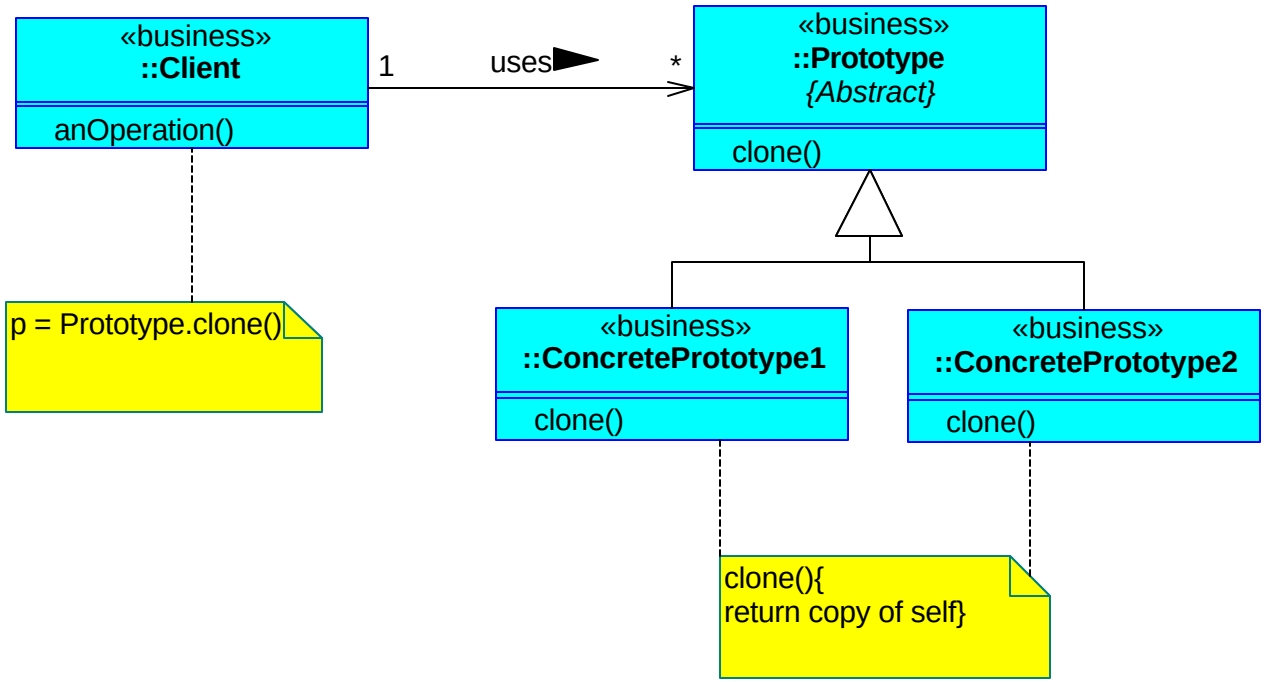
Prototype: The Problem

- Many similar requirements may lead to many similar objects
  - Too many
- Too many objects can lead to complex class hierarchies (of factories)
- Not all tasks are known in design time
  - Various clients may require new tasks from our objects
  - Therefore we can't hard-code these tasks in the objects
- Separate factory and object trees are hard to maintain

Prototype: The Solution

- Identify and design Objects that can be re-used in run-time to perform slightly different tasks
- These Objects must allow themselves to be copied
  - Cloneable
- These objects must allow access to their internal state and data after they have been cloned
- This allows an object to create a customized object
- A newly created object is created by a prototype making a copy of itself

Prototype: Solution, the Structure



Prototype: Components

- Prototype
  - declares an interface for cloning
- ConcretePrototype
  - implements the interface defined by the Prototype
- Client
  - creates a new object by asking a prototype to clone itself

Prototype: Implementation

- Uses a prototypical instance to define what kind of objects to instantiate
- Prototypes are created ahead of time
- Object creates new objects by copying these prototypes
  - suitable if new object types need to be added over time without the client having to know any details
- Objects created may be different classes or the same class with a different state
- Product class hierarchy defines a clone() operation for instantiating new objects
- Use a prototype manager if the number of prototypes is high

Prototype: Consequences

- Liabilities
  - It is difficult to implement the clone() method in each subclass
  - Shallow copies?
  - Circular references?
  - How do you initialize a clone if the client wants different values in different instances of the same prototype?
- Benefits
  - Add and remove products at runtime
  - Very efficient - actual prototypes can be created ahead of time
  - Allows the specification of new objects by varying structure
  - Configure an application at runtime
  - Greatly reduces the number of classes required

Prototype: Supplementary Info

- Known uses
  - A Connection Pool to a database creates multiple connections when started instances are passed to clients as required
  - JavaBeans
- Related patterns
  - Composite: Often used with the Prototype pattern
  - AbstractFactory: A good alternative to the prototype
  - Facade: The client often is an implementation of the Facade pattern and hides the classes that collaborate in the Prototype pattern
  - Factory Method: It is an alternative to the Prototype
  - Decorator: Often used with the Prototype pattern

Lesson Summary

- Requirements can expose the need for many similar tasks and information
- It is sometimes overkill to declare new classes for each task
- Instead we can re-use instances of the same class, by copying them
- This greatly reduces complexity
- Enhances flexibility

Sinclair Community College

Lesson 6: Survey of Creational Design Patterns

AbstractFactory Design Pattern
Singleton Design Pattern
Builder Design Pattern
Factory Method Design Pattern
Prototype Design Pattern
Survey of Creational Design Patterns

Lesson Objectives

- Describe how all Creational Patterns compare to each other
- Know when to use one of these patterns over the other

Comparison of Creational Patterns

- 5 Creational Patterns were introduced
  - AbstractFactory
  - Factory Method
  - Builder
  - Prototype
  - Singleton
- One approach is to subclass the class responsible for creating objects
  - the Factory Method takes this approach
- A second approach is to define an object that knows the class of the product objects and makes it the parameter of the system
  - AbstractFactory, Builder and Prototype follow this approach

Comparison of Creational Patterns (cont.)

- Factory Method
  - Easiest to use at first
  - Easy to define a new subclass
  - Requires you to create a new subclass to change a product
  - There is one subclass in the Factory Method hierarchy for each subclass in the product hierarchy
  - If the product creator is created by a Factory Method you need to subclass two tree hierarchies
- AbstractFactory
  - Does not provide an improvement over the Factory Method
  - Requires a class hierarchy as large as the Factory Method class hierarchy
  - Provides more flexibility than the Factory Method but is more complicated

Comparison of Creational Patterns (cont.)

- Prototype
  - Required to implement a Clone operation that can be complicated in some cases
  - Requires a smaller class hierarchy than `AbstractFactory` and `Factory Method`
  - Requires a new subclass for each new product
  - Clone operation can be used for purposes other then just pure instantiation

Class Review

- Do all Creational Patterns compete with one another?
- Is there always just one Creational Pattern that can be used as a solution to a problem?
- Which Design Pattern is more flexible: `Builder` or `Factory Method`?
- Why do you need to be aware of all Creational Patterns if there are a lot of similarities between them?
- Describe some situations when you would use a `Prototype` Pattern.

Lesson Summary

- Creational patterns are useful for large systems when it is important to have fine control of how objects are created
- Factory Method is a simple pattern that can be used instead of the AbstractFactory pattern
- The Builder pattern gives the system a fine grained control of how complicated objects are build out of simpler components
- The Prototype pattern is useful because it does not force a user to have a new class hierarchy like AbstractFactory or Factory Method
- The Singleton Design Pattern can be used in combination with other creational patterns to enforce that only one concrete Factory exists in a system

Sinclair Community College

Session 3: Exploring Structural Design Patterns

Composite Design Pattern
Adapter Design Pattern
Proxy Design Pattern
Bridge Design Pattern
Facade Design Pattern
Decorator Design Pattern
Survey of Structural Design Patterns

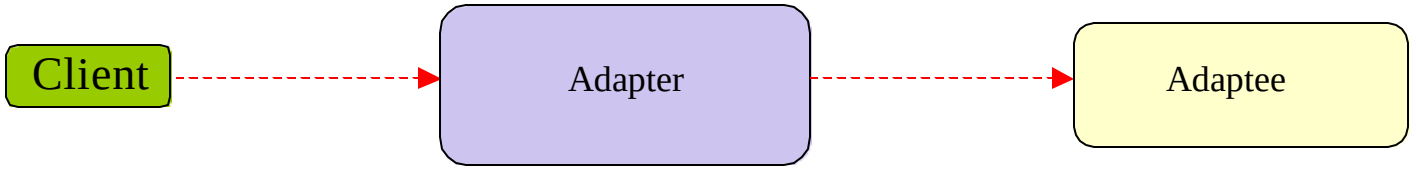
Educate. Collaborate. Accelerate!

Session Objectives

- Compose Interfaces and implementations using structures of objects
- Apply the Structural Design Patterns when developing applications

Structural Patterns Overview

- Structural Patterns concern the composition of classes and objects to form larger structures
- They use inheritance to compose interfaces or implementations
- Simple example: `Adapter`
  - `Adapter` makes one interface conform to another
  - uniform abstraction of different interfaces
- May be implemented as
  - a structure of objects (Object Adapter): Composition
  - a structure of classes (Class Adapter): Inheritance



- Object patterns compose objects to realize new functionality
  - the composition may change at run-time
- Class patterns allow arbitrarily complex structures

Sinclair Community College

Lesson 1: Composite Design Pattern

Overview of Structural Design Patterns

Composite Design Pattern

Adapter Design Pattern

Proxy Design Pattern

Bridge Design Pattern

Facade Design Pattern

Decorator Design Pattern

Survey of Structural Design Patterns

Educate. Collaborate. Accelerate!

Lesson Objectives

- Discover situations where the `Composite` pattern is appropriate
- Write patterns that use the `Composite` pattern
- Describe the trade-offs of using the `Composite` pattern

Composite: Introduction

- Summary
  - Allows the building of complex objects by recursively composing similar objects in a treelike manner
  - Together they form a semantically meaningful whole
  - All objects in the graph have a common superclass or interface
  - They are all manipulated in a consistent manner

Composite: Illustrative Example

- We want to write a simple calculator application to evaluate arithmetic expressions
- An arithmetic expression can be build out of arithmetic operations and digits
- Consider the following grammar to describe such a calculator:

```
expression = digit | digit operator expression
operator = ‘+’ | ‘-‘
digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’       | ‘8’ | ‘9’
```

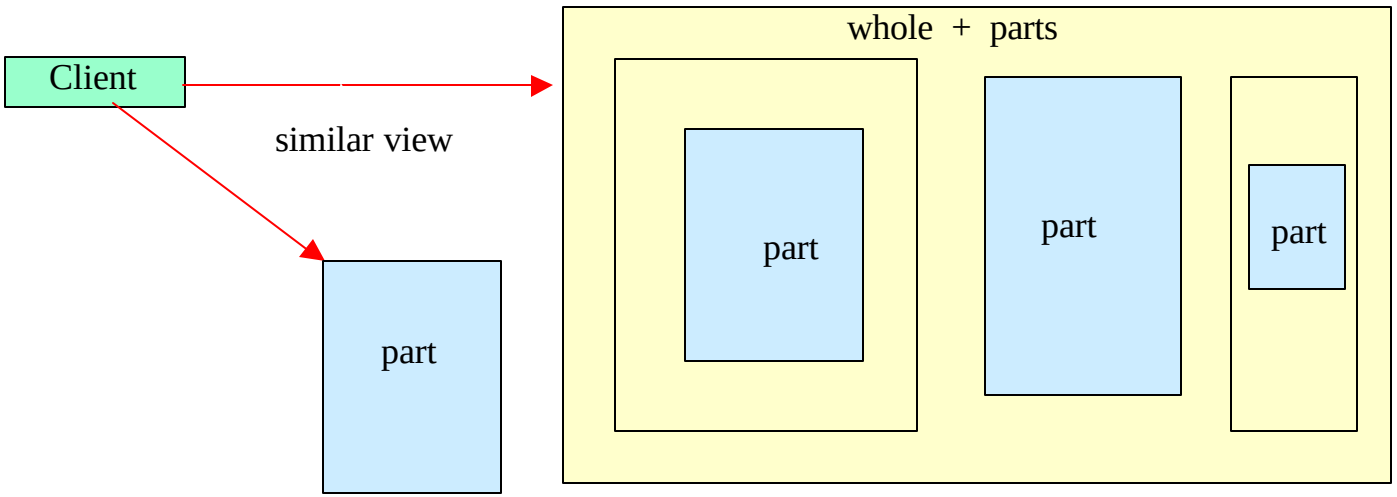
- The above grammar specifies that an expression can be either a digit or a digit followed by an operator followed by another expression

Composite: Illustrative Example

- The grammar defines only two valid operators: plus and minus
- It also defines all of the valid digits
- Each element of the grammar - expression, operator and digit - can be defined as a class
- The expression object is defined recursively in terms of itself

Composite: Context

- There is a set of primitive components that can be grouped together to form larger components
  - whole part relationship
- This process can be repeated recursively to form large composite objects



Composite: The Problem

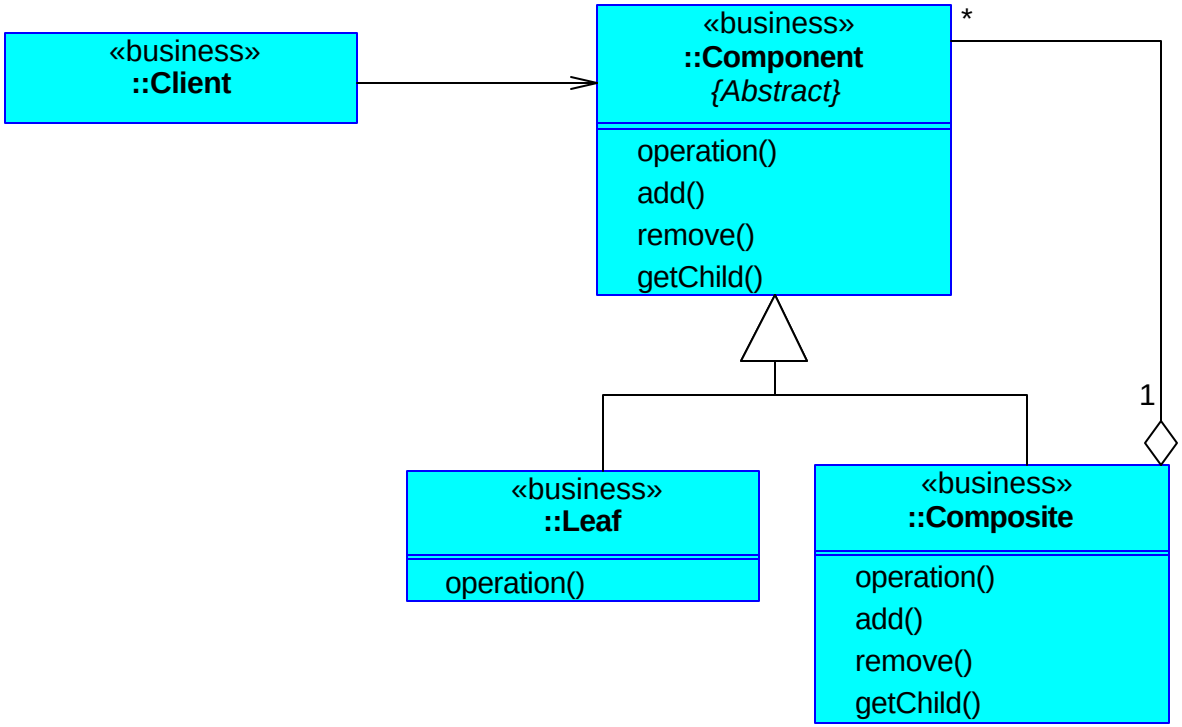
- Clients need to treat a Complex object as a single object
- It is not possible to predict the levels hierarchy
- Complex Objects may require to be decomposed into smaller components, recursively

Composite: Solution Outline

- Application must treat composite and primitive objects in a similar manner
  - the whole and the part appear the same
- Provide functionality for managing Composite objects

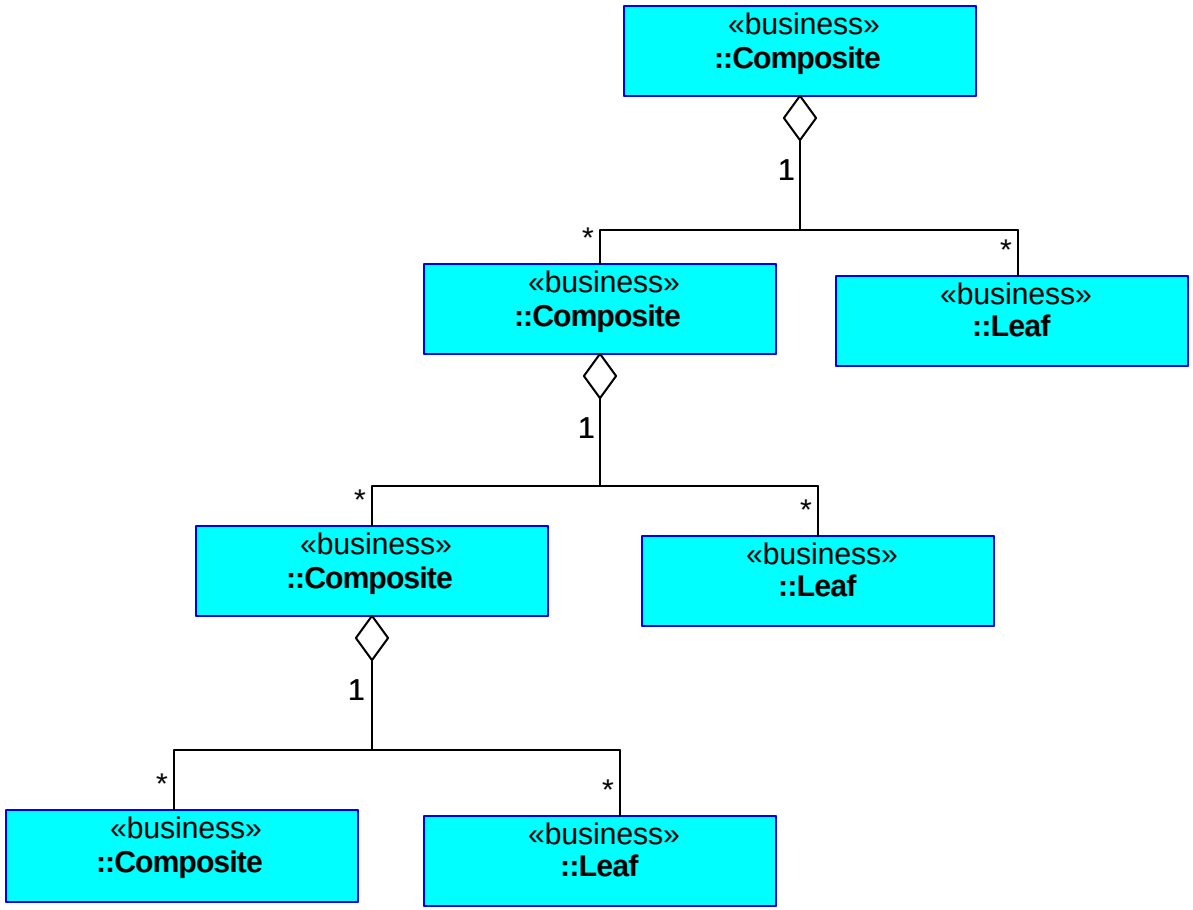
Composite: Solution Structure

- The structure of the Composite Pattern that is used to solve our problem:



Composite: Solution Structure Example

- The following diagram shows what a typical Composite object can look like:



Composite: Solution Components

- Component (expression class)
  - declares an interface for all of the objects in the design pattern
  - enables recursive composition of objects
- Leaf (digit classes)
  - represents basic building blocks in the Composite pattern
  - these are primitive classes that are used to build Composite objects
  - defines behavior for primitive objects
- Composite (operator class)
  - defines behavior for Composite objects
  - stores child components of the Composite object
- Client (application)
  - manipulates all of the objects through the interface of the Component class

Composite: Solution Implementation

- The Component class of the Composite pattern is the key.
  - represents both primitive and composite objects
  - gives the client a consistent view
  - client only sees a Component
- It defines the interface for building the composite objects
  - Methods add() and remove() are used to add/remove primitive objects to/from a Composite object

`Composite`: Implementation (cont.)

- These functions help us to build a composite object for the overall expression we want to evaluate
- Method `getChild()` allows us to walk through the composite object and evaluate the overall expression

Composite: Implementation (cont.)

- Sometimes a reference to the parent object in a child must be maintained
  - This could simplify removing elements of the composite object
  - When you are removing a child in the composite structure you need to update the parent to reflect that. The reference to a parent object simplifies this process.
- It makes sense sometimes to define methods `add()`, `remove()` and `getChild()` for the primitive object
  - This way we improve the flexibility of the design by making interfaces for primitive and composite objects identical
  - However these methods do not make sense for a `Leaf`. You can have the `Component` class define default implementations for these methods and have the Composite class override them. You can have a default "do nothing" implementation.

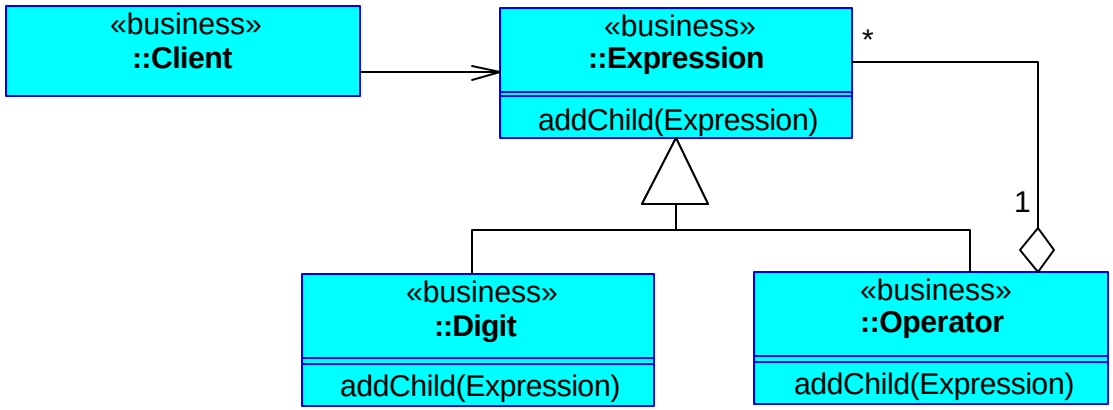
Composite: Code Example

- Code example for the Composite pattern is a calculator
- Assume that each Composite object can have only two Leaf classes
  - It is possible to write a more generic example in which a composite class can contain any number of primitive objects.
  - In this case a careful consideration of what data structure is to be used to keep track of all the child objects is required

`Composite`: Code Example (cont.)

- We will not implement `remove()` or `getChild()` methods
- Define two `add()` methods, one for left child and one for right child
- The following diagram shows the relationship between all of the classes in the design

Composite: Code Example (cont.)



Composite: Code Example (cont.)

```
final class OperatorType {
  private OperatorType() {}
  public static final OperatorType PLUS = new OperatorType();
  public static final OperatorType MINUS = new OperatorType();
}

abstract class Expression {
  abstract public void addRightChild(Expression component);
  abstract public void addLeftChild(Expression component);
}

class Digit extends Expression {
  public Digit(int d) {
    digit = d;
  }
  public void addRightChild(Expression component) {}
  public void addLeftChild(Expression component) {}
  private int digit;
}
```

Composite: Code Example (cont.)

```
class Operator extends Expression {
  public Operator(OperatorType type) {
    operator = type;
  }

  public void addRightChild(Expression component) {
    rightChild = component;
  }

  public void addLeftChild(Expression component) {
    leftChild = component;
  }

  private OperatorType operator;
  private Expression leftChild;
  private Expression rightChild;
}
```

Composite: Code Example (cont.)

```
class Composite_example1 {

  public static void main (String argv[])  {
    // build the following expression 3 + 4 - 5 + 6
    Digit digit3 = new Digit(3);
    Digit digit4 = new Digit(4);
    Digit digit5 = new Digit(5);
    Digit digit6 = new Digit(6);
    Operator firstPlus = new Operator( OperatorType.PLUS );
    Operator secondPlus = new Operator( OperatorType.PLUS );
    Operator minus = new Operator( OperatorType.MINUS );
    firstPlus.addLeftChild( minus );
    firstPlus.addRightChild( digit6 );
    minus.addLeftChild( secondPlus );
    minus.addRightChild( digit5 );
    secondPlus.addLeftChild( digit3 );
    secondPlus.addRightChild( digit4 );
  }
}
```

Composite: Consequences - Benefits

- User code can treat composite and primitive objects the same Way
- The client doesn't know whether he deals with a Composite or a primitive object
- The user code is simple and independent of the Composite object composition
- It is easy to add new types of objects to our design
  - A new type will be a sub class of a Component

`Composite`: Consequences - Liabilities

- It is hard to restrict what can be a valid component of the `Composite` class
- The user must use run-time checks to enforce any restrictions
- Discussion:
  - Why is this a problem?
  - Should the `Composite` care about the nature of its parts?

Class Review

- There can only be one type of the primitive object in the `Composite` Design Pattern. Is this true?
- True or False: There can only be one type of the `Composite` object in the `Composite` Design Pattern.
- True or False: It is easy to add new primitive and `Composite` classes to the class hierarchy.
- How many `Composite` objects exist in our example?
- Give three applications where the `Composite` pattern is useful?

Lesson Summary

- The `Composite` Design Pattern is useful when a collection of simple objects can be put together to form larger `Composite` objects
- The Client is aware only of the abstract `Component` class that provides a common interface to all of the primitive and `Composite` objects

Exercise 4 – Display Arithmetic

- Follow the instructions in the Student Guide

Sinclair Community College

Lesson 2: Adapter Design Pattern

Composite Design Pattern

Adapter Design Pattern

Proxy Design Pattern

Bridge Design Pattern

Facade Design Pattern

Decorator Design Pattern

Survey of Structural Design Patterns

Educate. Collaborate. Accelerate!

Adapter: Introduction

- Summary:
  - changes an interface of a class to be compatible with the interface expected by the clients of that class
  - clients do not know the class that fulfills the implementation
- Also known as Wrapper

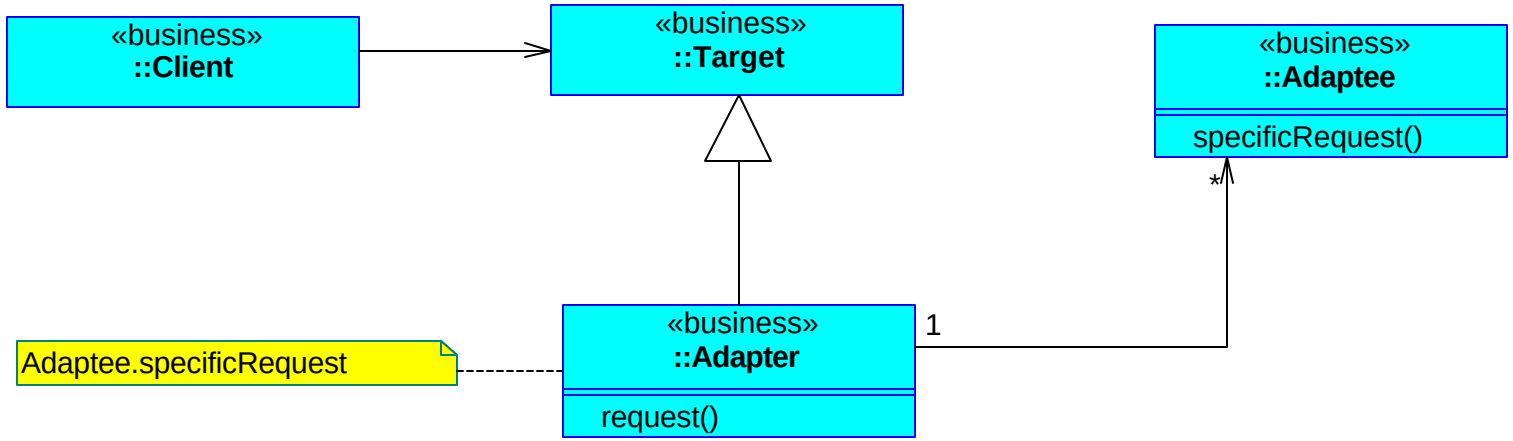
Adapter: Context and Problem

- Context
  - the class you are trying to reuse does not have an interface expected by the client
  - you need to create a reusable class that will work with unforeseen classes
- The Problem
  - Sometimes un-related classes need to work together
  - We can't anticipate all possibilities
  - Classes are not compatible

`Adapter`: Solution Outline

- Inheritance
  - Known as `Class Adapters`
  - Inherit the `Target` interface and extend the required interface
  - The `Adapter` is then implemented in the subclass
- Object Composition
  - Known as Object Adapters
  - Assimilate the target class
  - Add methods to translate calls

Object-Adapter: Solution Structure



Note: this diagram illustrates an object adapter, which uses composition rather than inheritance to adapt to the target

Adapter: Components

- Target
  - Defines the interface used by the client application
- Client
  - Uses objects that confirm to the interface defined by the Target class
- Adaptee
  - Defines an interface that we are trying to adapt in order to be usable by the client
- Adapter
  - Used to adapt an interface defined by the Adaptee class to confirm to the Target interface

Adapter: Implementation

- Identify the Adaptee classes
- Determine the amount of adaptation that the Adapter must perform
  - Interface mapping,
  - Multiple operations involved
- Decide on the form of the Adapter
  - Class vs. Object
  - Single Adaptee vs. multiple
  - Inclusion of sub-classes of Adaptee
- Determine whether the Adapter must be able to adapt to various adaptees
  - The pluggable Adapter
  - Multiple constructors

Adapter: Consequences

- Benefits of a Class Adapter
  - Easy to implement, Inherit, Extend and Override
  - Allows the adapter to adapt some of the behavior, not all
  - Easy to use: the adapter is one object, a subclass of the object we want to adapt.
- Liabilities of Class Adapter
  - The sub-classes will not be included, as we explicitly inherit from the class we want to adapt
  - In single inheritance languages works with only one class
- Benefits of an Object Adapter
  - Allows multiple objects to be adapted
  - Flexible: The adapted object can be passed as parameter
- Liabilities of Object Adapter
  - Explicit adaptation is required, methods must be declared in the Adapter class

Sinclair Community College

Lesson 3: Proxy Design Pattern

Overview of Structural Design Patterns
Composite Design Pattern
Adapter Design Pattern
Proxy Design Pattern
Bridge Design Pattern
Facade Design Pattern
Decorator Design Pattern
Survey of Structural Design Patterns

Educate. Collaborate. Accelerate!

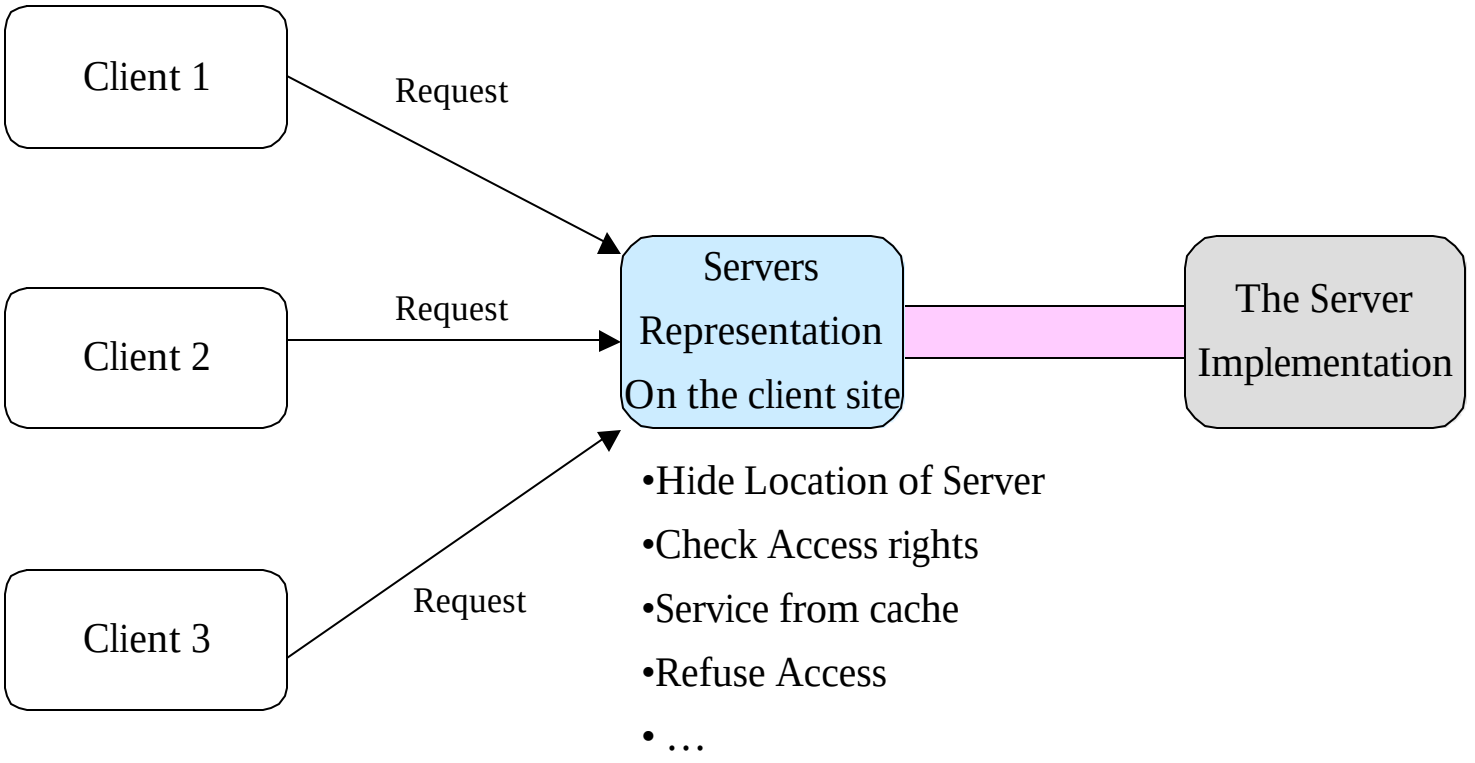
Lesson Objectives

- Discover situations where use of the `Proxy` pattern is appropriate

Proxy: Introduction

- Summary
  - Proxy acts as a surrogate for the object that provides the actual service
  - It effectively hides the actual object that does the work
  - Forces the client for a specific service to call the service indirectly through a local Proxy object
  - Interface for the Proxy and the actual object are the same
- Also Known As:
  - Surrogate

Proxy: Illustrative Example



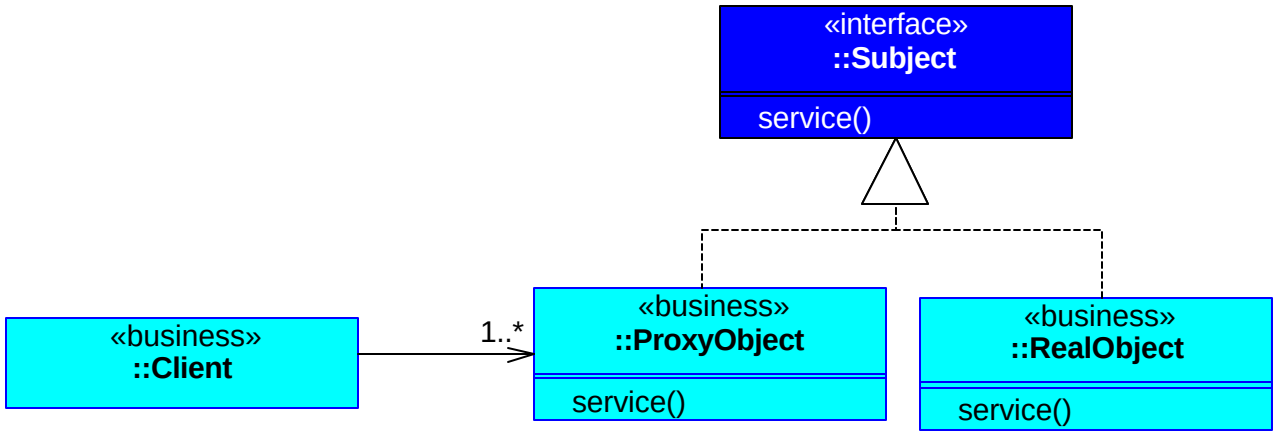
Proxy: Context

- Context:
  - Client needs to invoke the service object but direct access may not be favorable
  - Not all of the server implementation is required
- Problem
  - Access to the server must be location transparent
  - Security control and access checks must be performed before a request arrives at the server
  - Client Server interaction must be kept to a minimum
  - Proxy and Server must remain connected

Proxy: Solution Outline

- Hide the server behind a representative at the client's site. This is the Proxy
- The Proxy's interface must be a sub-set of server's interface
- Place all pre and post conditions for accessing the server in the Proxy

Proxy: The Solution Structure

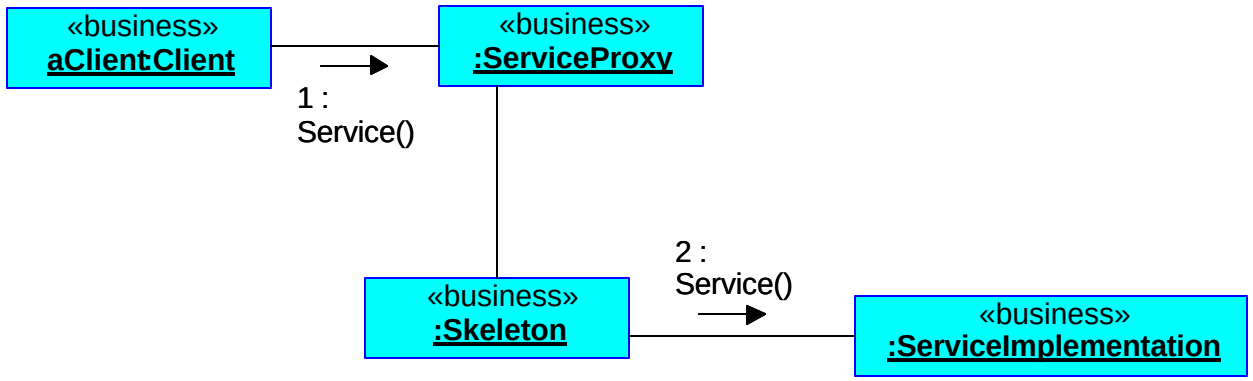


Proxy: Solution Components

- Proxy Subject
  - Provides the interface of the server to its clients
  - Is responsible for all operational issues around accessing the server
- Real Subject
  - Provides the actual implementation of the service
- Interface Subject
  - Serves as the abstract template for both Real and Proxy subjects

Proxy: Solution Dynamics

- For a remote object the process of invoking the service is as follows:



- Client invokes Proxy method
- Proxy marshals the request, which turns it into a wire format
- Wire format request is sent to the remote system

Proxy: Solution Dynamics (cont.)

- Skeleton receives the wire format at the remote end
- Skeleton un-marshals the request and invokes the method on the service provider implementation
- Service provider implementation processes the request and returns the response
- Response is returned in a similar, but reverse, process

Proxy: Implementation

- In the real world, mostly proxies are generated
  - RMI, CORBA,
- A generic approach to Proxy design
- Separate functional and operational concerns
- Regarding accessing the services of a servers
- Categorize the operational concerns
  - Each category leads to a specific type of Proxy with its own set of responsibilities, e.g. Remote, Cache and Virtual proxies
- Connect the Proxy to its server
  - Use/Design communication protocol

Proxy: Implementation, Proxy Types

- Remote Proxy
  - A local representation of an object that is located on the remote system
- Virtual Proxy
  - Creates expensive objects on demand, i.e., when an object is accessed
- Protection Proxy
  - Useful when objects should have different access rights. The access rights are determined by the time of the proxy used.
- Smart Reference
  - Used when a simple reference is not enough
  - Can perform additional functionality like reference counting

Proxy: Implementation (cont.)

- Proxies to a remote object are known as *remote proxies*
- How is the Proxy object kept in sync with the Service object?
  - The service object places "its" proxy in a "known place"
  - The client downloads the proxy from the known place at run time
- How can the ServiceObject be created at run time without the client being aware of it?
  - it becomes the responsibility of the proxy and is transparent to the client

Proxy: Code Example

- This example implements a simple calculator that only adds double integers.

```
public class CalcClient {
  public static void main(String[] args) {
     try {
       Calculator c = (Calculator) LookupService.lookup("calculator");
       System.out.println("3.2 + 4.5 = " + c.add(3.2, 4.5));
     } catch (Exception e) { System.out.println(e);}
   }
}
```

```
C:\>java CalcClient
3.2 + 4.5 = 7.7
```

```
public interface Calculator {
  double add(double a, double b);
}
```

- LookupService object is provided to supply the Proxy object to the client
  - returns an object of type Calculator
  - has one method add()

Proxy: Code Example (cont.)

- Proxy object

```
public class CalcProxy implements Calculator {
  public double add(double a, double b) {
    CalcImpl c = new CalcImpl();
    return c.add(a, b);
  }
}
```

- Creates the object that provides the service
- Invokes the appropriate method: add() in this case
- Calculator implementation

```
public class CalcImpl implements Calculator {
  public double add(double a, double b) {
    return a + b;
  }
}
```

Proxy: Consequences

- The service-providing object provides the services in a manner that is transparent to the object itself and its clients
- Many more potential failure modes exist when this pattern is used
  - e.g. network, marshalling, un-marshalling errors for remote object

Proxy: Examples

- CORBA
- Java RMI

Exercise 5 – Build a Calculator

- Follow the instructions in the Student Guide

Sinclair Community College

Lesson 4: Bridge Design Pattern

Composite Design Pattern
Adapter Design Pattern
Proxy Design Pattern
Bridge Design Pattern
Facade Design Pattern
Decorator Design Pattern
Survey of Structural Design Patterns

Educate. Collaborate. Accelerate!

Lesson Objectives

- Separate an abstraction from its implementation so that the abstraction and its implementation can vary independently

Bridge: Introduction

- Summary
  - Separates the abstraction and the implementation of a typical object in separate independent classes that are combined dynamically
  - Used when there is a hierarchy of abstractions and another hierarchy of implementations
- The Bridge pattern is also known as a "Handle/Body"

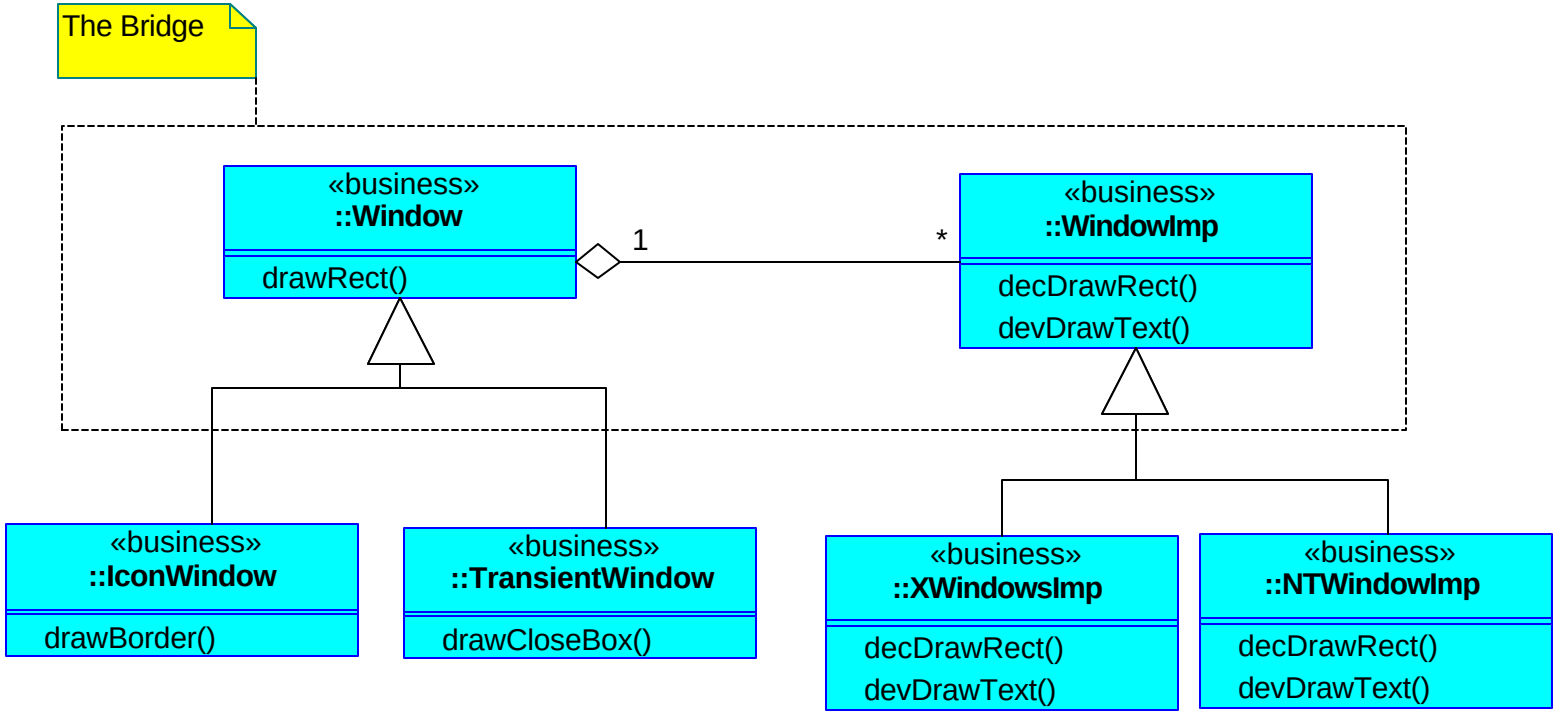
Bridge: Context 1

- Imagine we want to implement a portable user interface toolkit
- We have a `Window` abstraction that we want to support for both Windows NT (NT) and X Window System (X) "look and feel"
- The simple solution is to have a `Window` as a base class and have `NTWindow` and `XWindow` classes subclassed from it

Bridge: Context 2

- Imagine that we want to extend our window abstraction to support various types of windows
- Let's say we want to create a subclass of a Window called `IconWindow`
- We need to create two subclasses of `IconWindow` for each platform that we create
- This makes adding new `Window` subclasses very difficult

Bridge: Context (cont.)



Bridge: Problem

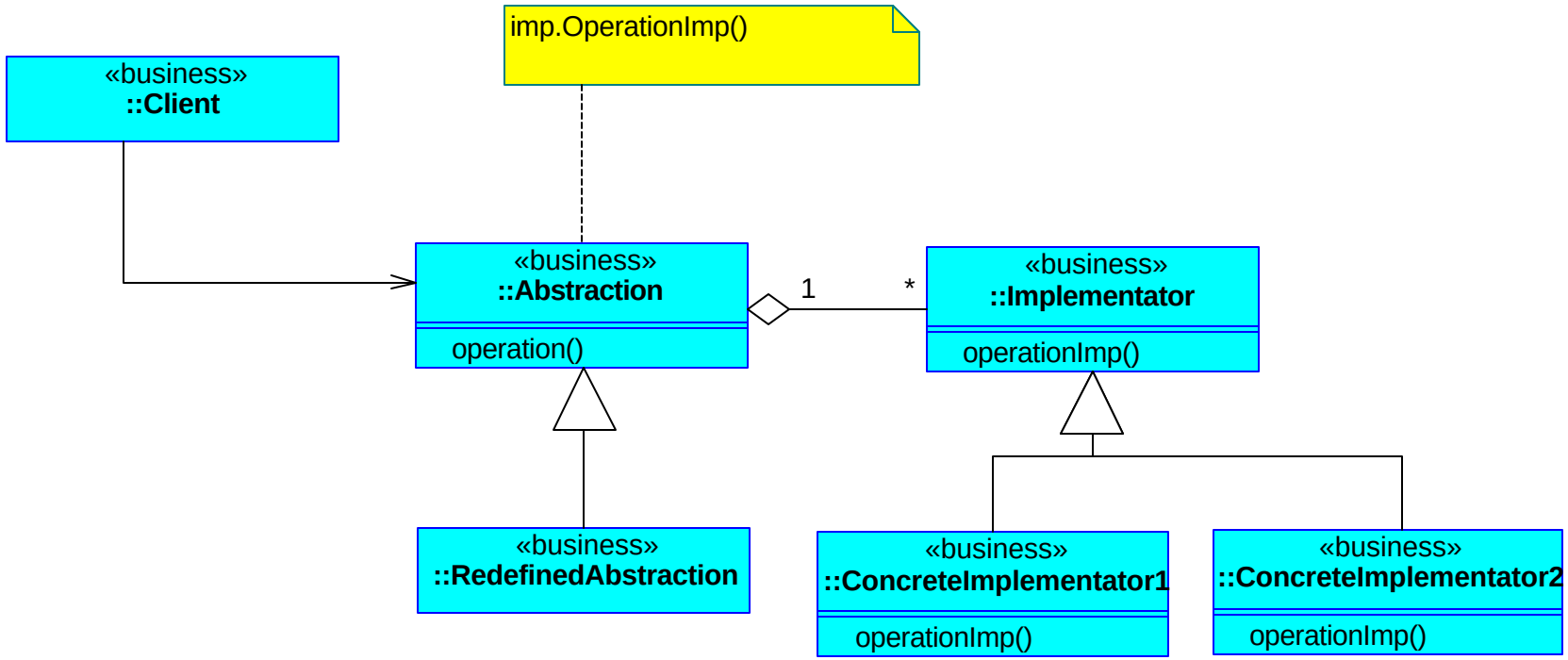
- Adding a new platform is also hard
- We need a new subclass for every kind of window that we support
- Whenever a client creates a window, a concrete class for a specific platform is instantiated.
- This makes client code platform-dependent and hard to port
- Clients should be able to create a window without committing to a concrete implementation
- The `Bridge` class let us solve the above problems by separating an abstract concept from its implementation

Bridge: Solution outline

- The Bridge Design Pattern puts window abstraction and implementation into two separate class hierarchies
- The next slide shows the new class hierarchy
- All operations in the Window subclasses are implemented in terms of abstract operations of the WindowImpl class
- This removes the relationship between window abstraction and the system dependent code
- The relationship between Window and WindowImpl classes is called a Bridge

Bridge: Solution Structure

- UML Pattern of the Bridge design pattern



Bridge: Components

- Abstraction
  - Defines an interface for the abstraction
  - Contains a reference to an implementation object
- RefinedAbstraction
  - Extends the functionality of Abstraction
- Implementor
  - Defines an interface for the implementation
  - This interface does not have to be identical to the Abstraction interface
  - Usually defines low level primitives that are used by higher level operations defined in the Abstraction interface
- ConcreteImplementor
  - Concrete implementation of the Implementor

Bridge: Implementation

- You may want to use the Bridge Design Pattern in the following situations
- You want to decouple an abstraction from its implementation.
  - This is maybe the case when an implementation must be chosen at the run time
- You want to able to easily add new abstractions and to extend implementations
- You want to hide an implementation of the abstraction
  - In C++ the representation of a classis visible in the interface
- You want changes in the implementation have no effect on the client code
  - I.e. the client code does not need to be recompiled
- You want to share an implementation among multiple objects
  - E.g multiple objects sharing a single string representation

Bridge: Code Example, Error Handling

- In this example we have an abstraction for the error message
- The error message can come it two flavors: fatal and system
- There are three different implementations of how to format error messages
- If the user does not specify which format to use then a default one is selected
- The following slides show the source code for this example

Bridge: Code Example, Error Handling

```
abstract class ErrorMessage {
  abstract public void displayErrorMessage(String msg);

    public void SetErrorMessageType (String type) {
      if ( type.equals("type 1")) {
        implementation = new Type1ErrorMessageImp();
      }
      else if ( type.equals("type 2")) {
        implementation = new Type2ErrorMessageImp();
      }
      else {
        implementation = new DefaultErrorMessageImp();

      }
      }

      protected ErrorMessageImp implementation = new DefaultErrorMessageImp();
}
```

Bridge: Code Example, Error Handling

```
class FatalError extends ErrorMessage {

  public void displayErrorMessage(String msg) {
    implementation.display("Fatal Error: " + msg);
  }
}

class SystemError extends ErrorMessage {

  public void displayErrorMessage(String msg) {
    implementation.display("System Error: " + msg);
  }
}
```

Bridge: Code Example, Error Handling

```
abstract class ErrorMessageImp {

  abstract public void display(String msg);
}

class Type1ErrorMessageImp extends ErrorMessageImp {

  public void display(String msg) {
    System.out.print("You've been graced with the following error: ");
    System.out.println(msg);
  }
}

class Type2ErrorMessageImp extends ErrorMessageImp {

  public void display(String msg) {
    System.out.print("Oh no, you have the following error: ");
    System.out.println(msg);
  }
}

class DefaultErrorMessageImp extends ErrorMessageImp {

  public void display(String msg) {
    System.out.println(msg);
  }
}
```

Bridge: Code Example, Error Handling

```
class Bridge_example1 {

  public static void main (String argv[]) {
    SystemError  system = new SystemError();
    system.SetErrorMessageType("type 1");
    system.displayErrorMessage("Drive a is not ready.");
    system.SetErrorMessageType("type 2");
    system.displayErrorMessage("Drive a is not ready.");
  }
}
```

Class Review

- Why do we have two class hierarchies in the `Bridge` Design Pattern?
- How can you use `Singleton` Design Pattern in combination with `Bridge` Design Pattern?
- Is it easy to add a new abstraction?
- Is it easy to add a new implementation?
- Describe situations where you would use a `Bridge` Design Pattern.

Lesson Summary

- `Bridge` Design Pattern is useful for writing code that must behave differently for each operating system
- With a `Bridge` Design Pattern it is easy to add new concepts to the application as well as new operating systems that are supported

Exercise 6 – Error message styles

- Follow the instructions in the Student Guide

Sinclair Community College

Lesson 5: Façade Design Pattern

Composite Design Pattern
Adapter Design Pattern
Proxy Design Pattern
Bridge Design Pattern
Facade Design Pattern
Decorator Design Pattern
Survey of Structural Design Patterns

Educate. Collaborate. Accelerate!

Lesson Objectives

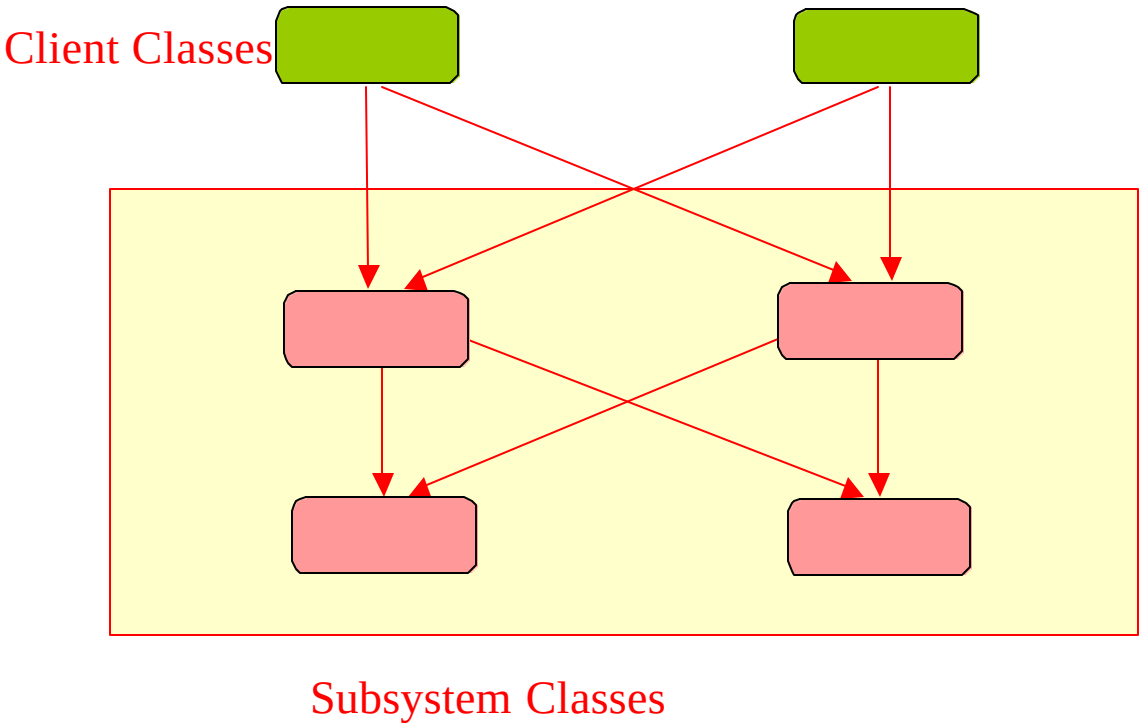
- Write a common interface for a group of related classes
- Learn to recognize the situations where the Facade pattern can be effectively applied

Facade: Introduction

- Summary:
  - provides an object that simplifies all access to a set of related objects
  - serves as a middleware layer between the client objects and the subsystem objects

Facade: Context

- We have a set of classes that are accessible by outside objects
- We want to access these in a unified manner that is easy to use
- This diagram illustrates how we access classes without a Facade pattern

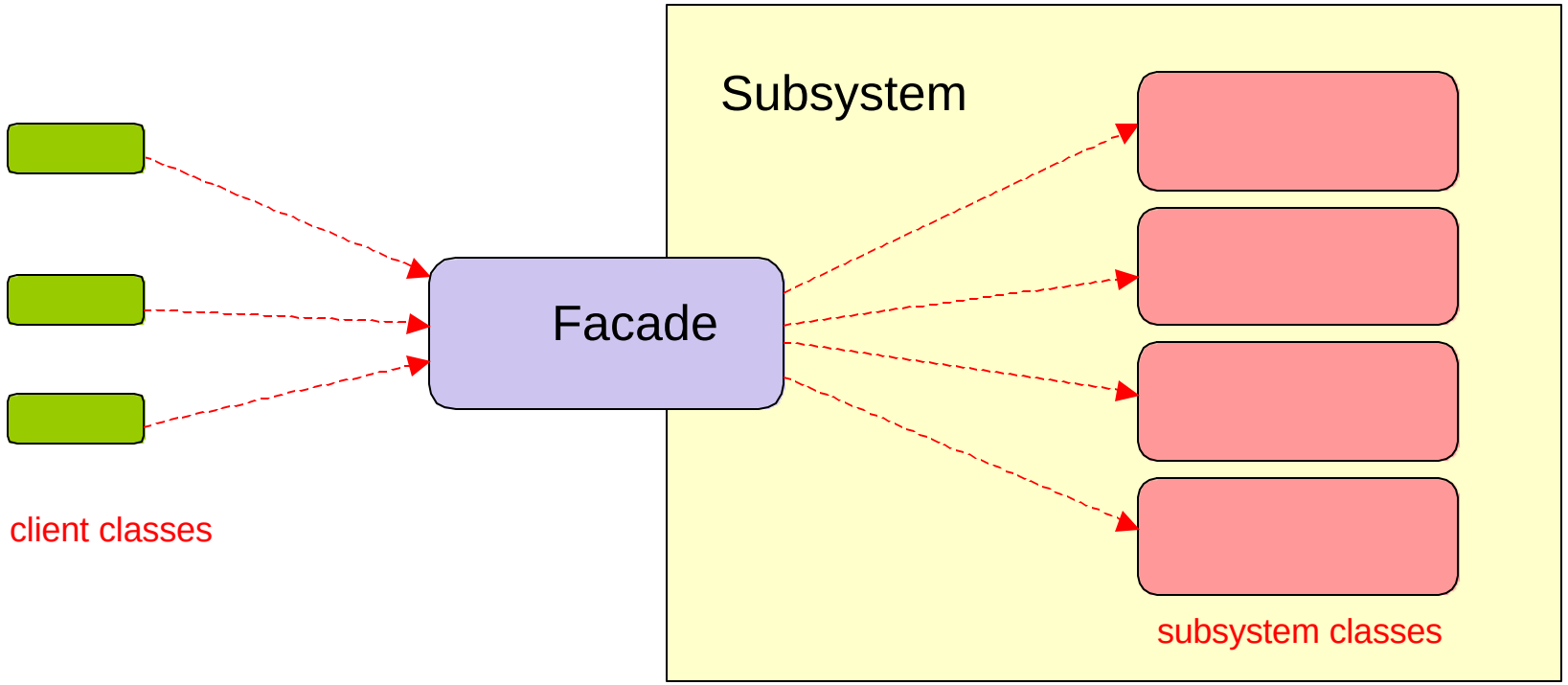


Facade: The Problem

- Clients do not have to know about all of the classes in a subsystem, all of the time
- They require a simple interface to the functionality of the subsystem
  - Compile code service vs. knowing all individual components of a compiler sub-system

Facade: Solution Outline

- Accessing classes through the Facade pattern solves these problems
- Subsystem classes perform the actual work
- They have no knowledge of how the subsystem objects function
- Clients have no direct access to the subsystem classes



Facade: Code Example

- We have an implementation for a simple compiler
- The compiler consists of two components: `Scanner` and `Parser`
- The client has to use both components to compile a program
- We are going to write a `Compiler` class that will hide this complexity from the user
  - The compiler class implements the `Facade` pattern
- The following is the source code for this example

Facade: Example Code (cont.)

- Compiler hides the implementation subsystem classes

```
class Scanner {

  public void scan(){
    System.out.println("Scanning the program");
  }
}

class Parser {

  public void parse(){
    System.out.println("Parsing the program");
  }
}

class Compiler {

  public void compile() {
    Scanner scanner = new Scanner();
    Parser  parser = new Parser();

    scanner.scan();
    parser.parse();
  }
}
```

Facade: Example Code (cont.)

```
class Facade_example1 {

  public static void main (String argv[]) {
    // Create the Facade
    Compiler compiler = new Compiler();
    // Have it operate
    compiler.compile();
  }

}
```

Facade: Consequences, the Benefits

- Provides single interface to the subsystem
- Provides a layer of abstraction thus decoupling the interface from the implementation of your subsystem
- Provides a simpler interface to the complex subsystem
- Hides the subsystem elements for the clients
- Decouples the subsystem from the clients thus making it possible to change it without affecting the clients
- Promotes weak coupling between the clients and the subsystem

Class Review

- Why is a Facade Design Pattern necessary?
- You want to enhance functionality of a Compiler. You are going to add a class Optimizer to optimize source code. Will this have any affect on the software outside of the compiler subsystem?
- Does the Facade pattern forces you to communicate with the subsystem only through the Facade?
- Does the Facade pattern promote weak or strong coupling between subsystems and clients?

Lesson Summary

- It is very important to limit the communication between different modules in the system
- This makes the whole system easier to understand and makes it easier to change each subsystem without affecting the rest of the software
- The Facade Design Pattern accomplishes this by limiting all of the accesses to the subsystem to a single class

Exercise 7 – Scanning and Parsing

- Follow the instructions in the Student Guide

Sinclair Community College

Lesson 6: Decorator Design Pattern

Composite Design Pattern
Adapter Design Pattern
Proxy Design Pattern
Bridge Design Pattern
Facade Design Pattern
Decorator Design Pattern
Survey of Structural Design Patterns

Educate. Collaborate. Accelerate!

Decorator: Introduction

- Summary
  - adds new responsibilities to an object at run time that is transparent to its clients by using an instance of a subclass of the original class that delegates its operation to the original object
- Also known as Wrapper

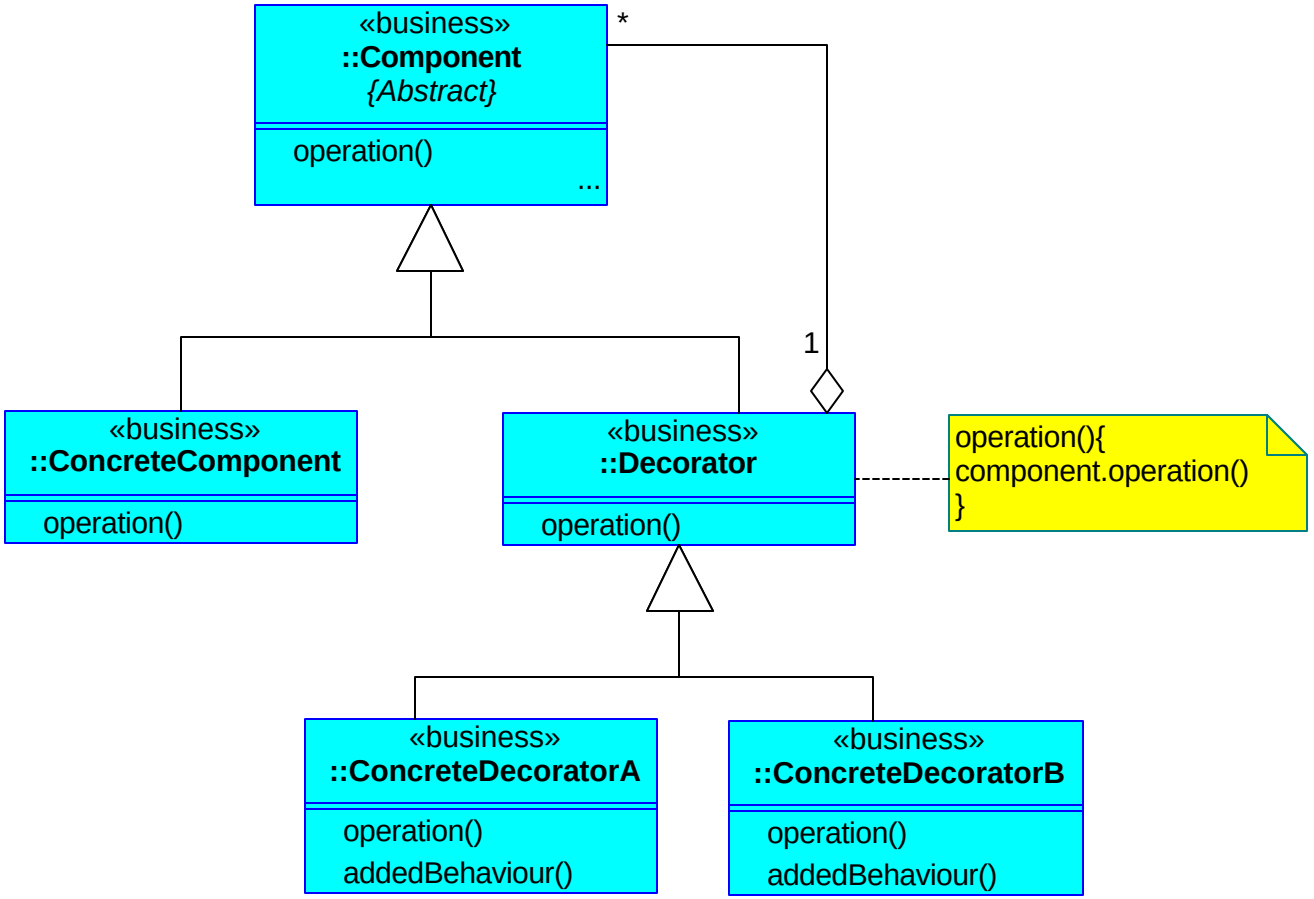
Decorator: Context & Problem

- Context
  - When functionality needs to be added to instances rather than classes
  - The circumstances determine when the functionality needs to be added
  - When the visible object behavior needs to be extended or modified, rather than object's internal behavior
- Problem
  - Pure sub-classing can create a very large number of classes to cover all of the possibilities

Decorator: Solution Outline

- Use composition versus sub-classing to extend functionality
- Functionality is added on a per object basis as opposed to a per class basis
- Provide means for removing functionality at run time
- Design wrapper objects each adding only one new function
- It has the same interface as an interior object
- Exterior is just a wrapper for methods that it does not extend

Decorator: Solution Structure



Decorator: Solution Components

- Component
  - Defines an interface of objects that can have responsibilities added to them at run time
- ConcreteComponent
  - Implements the interface defined by the component class
  - Can have responsibilities added to it at run time
- Decorator
  - Keeps reference to the Component class to enable addition of multiple new responsibilities at run time
  - Defines an interface to confirm to the Component interface
- ConcreteDecorator
  - Implements the interface defined by the Decorator class
  - Used to add new responsibilities to the component at the run time

Decorator: Implementation

- Within the context of an application
- Identify extension requirements
  - The classes vs. cases matrix
- For each extension case of a class, design a `ConcreteDecorator`
- Analyse `ConcreteDecorator`'s
  - Identify commonalities in order to abstract the Decorators
- Make sure the `Decorator` signatures match those of the decorated objects

Decorator: Known Uses

- GUI Development frameworks
  - Adding various sorts of effects to widgets
- Client Server interaction
  - Adding operational functionality, like instrumentation and access control to requests

Decorator: Consequences

- Benefit
  - When you use the Decorator pattern you have a lot of little objects that differ only in how they are interconnected
- Liability
  - This makes a system easy to change but hard to understand
  - Although the interfaces of the Decorator and the decorated objects are compatible, their internal behavior is different. This may make it difficult to localize errors.

Decorator: Related Patterns

- Strategy
  - Strategy is used in order to change the internal behavior of objects. Decorator changes the external appearance of objects
- Adapter
  - The main responsibility of an Adapter is to change the adaptee's interface, not its appearance and visible behavior
- Composite
  - A Composite is meant to aggregate objects into a semantically cohesive composite object, so that it forms a single point of reference for all component objects. Decorator adds functionality to single objects

Sinclair Community College

Lesson 7: Survey of Structural Design Patterns

Composite Design Pattern

Adapter Design Pattern

Proxy Design Pattern

Bridge Design Pattern

Facade Design Pattern

Decorator Design Pattern

Survey of Structural Design Patterns

Lesson Objectives

- Understand the commonalities amongst the Structural Design Patterns
- Decide when to use each Structural Design Pattern

Common Trends Among Structural Patterns

- Show how to compose classes and objects to form larger structures
  - Inheritance is used by Structural class patterns to build implementations or interfaces
  - Composition is used by Structural object patterns to create new functionality at run time
- Adapter pattern is an example of the Structural class pattern.
  - It makes an interface of one class compatible with the interface of another class
- Decorator pattern is an example of the Structural object pattern.
  - It allows you to add new responsibilities to an object at runtime

Comparison of Structural Patterns

- 6 Structural Patterns were introduced:
  - Composite
  - Bridge
  - Facade
  - Adapter
  - Decorator
  - Proxy
- Many structural patterns have similar UML diagrams
- All rely on the same set of mechanisms to accomplish their objectives
- Similar in structure, these design patterns have different goals

Comparison of Structural Patterns (cont.)

- Bridge versus Adapter
  - Both patterns provide a level of indirection
  - Both patterns forward the request to the object
  - These patterns however have different intents:
  - Bridge pattern provides a connection between an abstract concept and its implementations
  - Adapter pattern tries to resolve two different interfaces
- Composite versus Decorator
  - Both Composite and Decorator rely on the recursive composition to handle any number of objects
  - They have completely different goals:
  - Composite lets you treat different objects uniformly
  - Decorator adds new responsibilities to an object

Class Review

- Do you need to have the source code of the `Adaptee` class to make the `Adapter` pattern work?
- Think of several situations where a `Proxy` is useful in an Internet application.

Lesson Summary

- Structural patterns show how to compose smaller objects, classes to form larger structures
- Many of the structural patterns have similar UML diagrams but different intents

Sinclair Community College

Session 4: Exploring Behavioral Design Patterns

Observer Design Pattern
Strategy Design Pattern
Iterator Design Pattern
Visitor Design Pattern
Interpreter Design Pattern
Chain of Responsibility Design Pattern
Command Design Pattern
Mediator Design Pattern
State Design Pattern
Comparison and Summary

Educate. Collaborate. Accelerate!

Sinclair Community College

Lesson 1: Observer Design Pattern

Observer Design Pattern

Strategy Design Pattern

Iterator Design Pattern

Visitor Design Pattern

Interpreter Design Pattern

Chain of Responsibility Design Pattern

Command Design Pattern

Mediator Design Pattern

State Design Pattern

Comparison and Summary

Lesson Objectives

- Understand how to reduce dependency amongst objects
- Understand how the observer pattern enables us to achieve this

Observer: Introduction

- Summary
  - The Observer pattern enables semantically related objects to remain synchronized with one another, while dependency amongst these objects is reduced
- Also known as
  - Publisher-Subscriber, Dependents

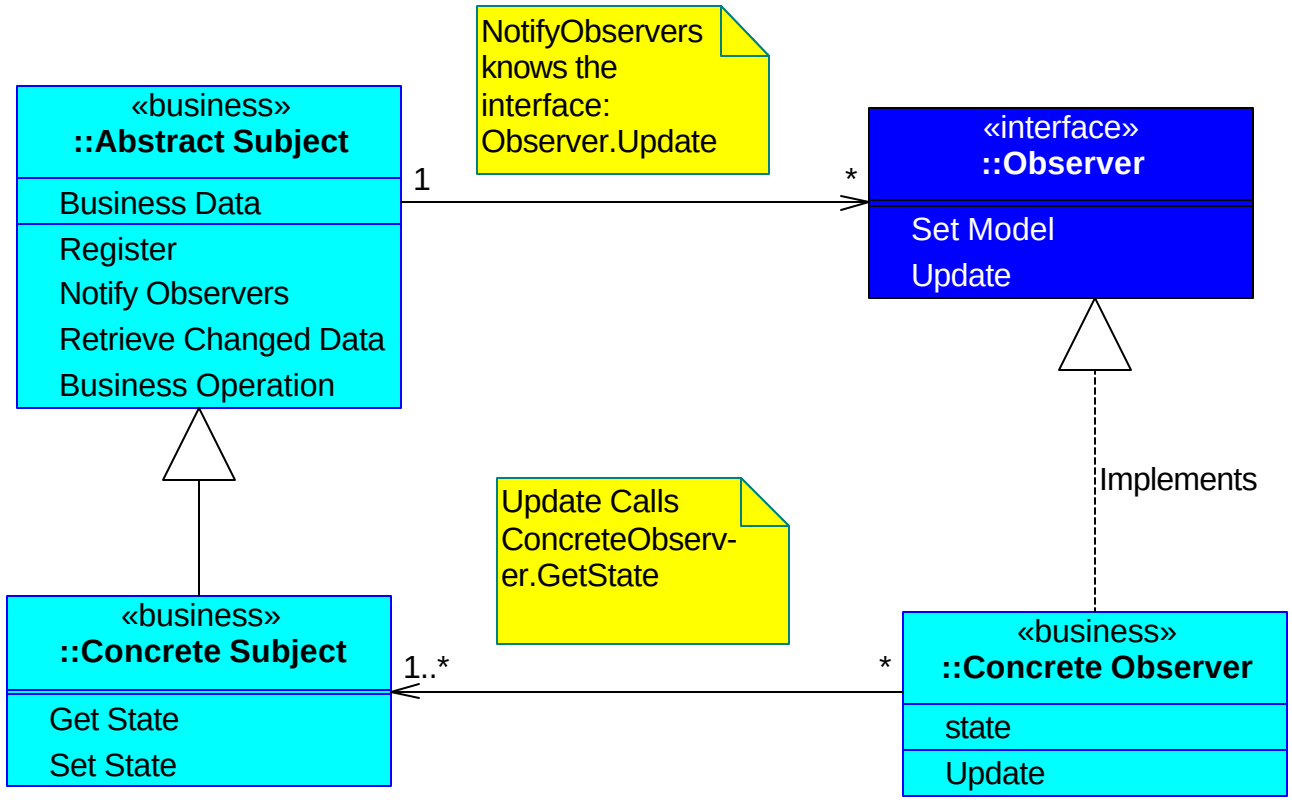
Observer: Context and Problem

- Context
  - When a business object has multiple representations
  - When all of its representations, or dependents must be notified when the object's state changes
- Problem
  - The number of the dependent objects is unknown in advance
  - The exact types of the objects are unknown in advance
  - Making Dependents continuously enquiring about change is impractical, or even impossible

Observer: Solution Outline

- Separate the responsibilities into two distinct entities:
  - The Observable, or Subject: The source of data and state
  - The Observer: All dependant objects on the Subject
- The Subject must maintain a list of all observers
  - But their specific Type must be unknown to Subject
  - Design a common interface for all Observers
- Design Services for registering and un-registering to the Subject
- Choose or design a propagation mechanism
  - Must notify all objects registered with the Subject through a common interface

Observer: Solution Structure



Observer: Solution Components

- AbstractSubject
  - Declares the interface the notifies all Observer objects
- Observer
  - An Abstract Interface that is implemented by all Observer objects
  - Observer consists of one method: Update
- ConcreteSubject
  - Responsible for managing its own state and data
  - When getState is invoked, ConcreteSubject provides its state and data to an Observer
  - May override notifyObservers
  - Maintains a list of its Observer objects
- ConcreteObserver
  - Must respond to a notification, by executing update
  - Implements update
  - Knows its Subject

Observer: Solution Dynamics

- Registering Observers
  - On construction of a new concrete Observer the reference to its ConcreteSubject is set
- Notifying Observers

The state of a subject changes

Subject Decides to notify observers of the change

While observers in the list

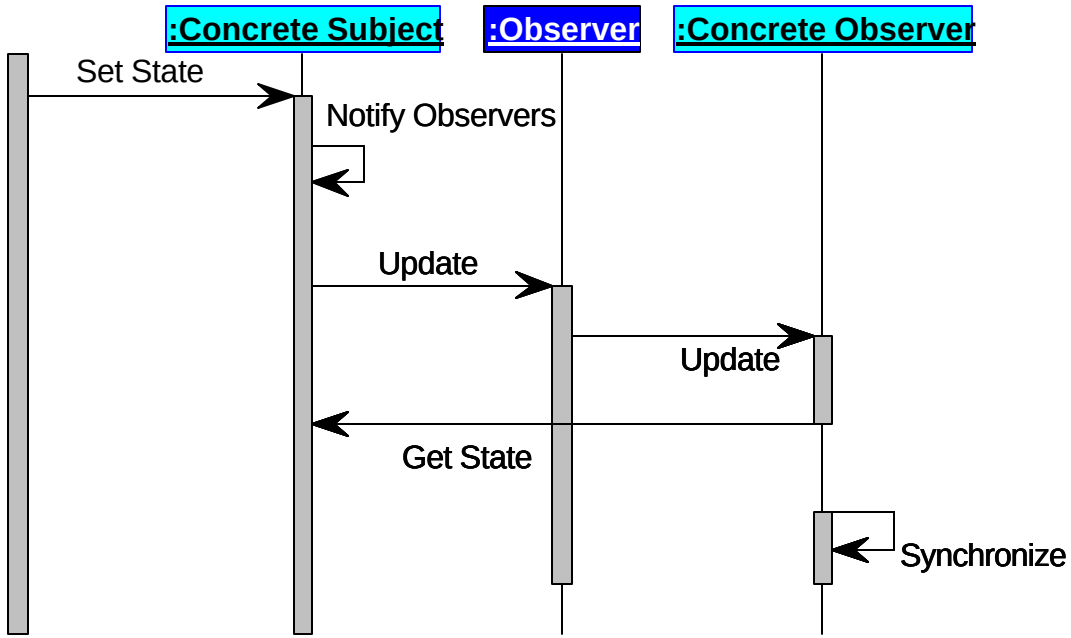
Call the interface

Implementation is executed

Observer queries Subject about the change

Observer synchronizes with Subject

End While observers in the list



Observer: Implementation

- Design your Subject class
  - Inherit from class java.util.Observable
  - Decide when to notify Observers
- Add registry of Observers
  - Class java.util.Vector
- Design the Interface to Observers
  - Interface java.util.Observer
- Design Concrete Observers
  - Implement Interface java.util.Observer
  - Implement update
- The notification mechanism
  - Is implemented by the Java run-time environment
  - Can be implemented using events and listeners

Observer: Example Code, Context

- Hotel with three different views
- When rooms change their state, views are updated automatically

Observer: Example Code (cont.)

- The class Hotel is the Subject Class

```
class Hotel extends Observable{

  public Hotel(int numberOfRooms) //Constructor {
    rooms = new Vector();
    for (int i=0; i<= numberOfRooms; i++){
      Room r = new Room(i);
      rooms.addElement(r);
    }
  }
}
```

- The method reserveRoom decides to notify the Observers

```
public void reserveRoom(boolean state, int i){

  Room r = (Room) rooms.elementAt(i);
  r.reserve();
  setChanged();
  lastChangedRoomNr = i;
    notifyObservers();
  }
```

Observer: Example Code (cont.)

- Method `getChangedData()` provides `Observers` with the details of changes in the `Subject`
  - Must indicate that changed has been handled

```
public  Room getChangedData(){
  Room r = (Room) rooms.elementAt(lastChangedRoomNr);
  clearChanged();
  return r;
}
```

- Other methods for managing rooms
  - Manipulating the Vector

Observer: Example Code (cont.)

- Designing the Room:

```
public Room (int n) // Constructor, sets room status to empty
public void reserve() // Flips room status
public String getRoomNo() // Returns #room in String
public String getRoomStatus() // Returns Room state in a String
private int     number;
private boolean reserved;
```

Observer: Example Code (cont.)

- Designing the ConcreteObservers 1

```
class ViewOne extends Frame implements Observer{

  public ViewOne(Hotel h){
    setTitle("View 1");
    h.addObserver(this); // Adds this observer to hotel
    setSize(200, 200);
    setVisible(true);

    private Observable model;

    private List list

  }
```

Observer: Example Code (cont.)

- Designing the Observers 2
- Initialize the Observer with its Subject

```
public void initialize(Observable o){
   this.model = o; // Reference to hotel, but can be any other
                   // Observable
   if ( o instanceof Hotel ) // Check Type of Observable
   {
     int s = ((Hotel)o).countRooms();
     list = new List (s);
     Room r;
     for (int i=1; i < s; i++) // Draw view
     {
       r = ((Hotel)o).getRoomAt(i);
       list.addItem(r.getRoomNo() + " " + r.getRoomStatus() );
     }
     this.add ( list );
   }
 }
```

Observer: Example Code (cont.)

- Designing the Observers 3
- Implement the update method that holds the synchronization logic

```
public void update(Observable model, Object arg)
  {
    Room r;
    int s = ((Hotel)model).countRooms();
    list.removeAll();

    for (int i=1; i < s; i++){
      r = ((Hotel)model).getRoomAt(i);
      list.addItem(r.getRoomNo() + " " + r.getRoomStatus() );
    }
  }
```

Observer: Example Code (cont.)

- Another Observer can be designed which also handles input or the reservations
  - The same as class ViewOne
  - All rooms are presented as checkboxes
  - User can (un)check to reserve or free a room

```
public void mouseClicked(MouseEvent e)
  {
    ((Hotel)model).reserveRoom(checkBox.getState(),
                   Integer.parseInt(choice.getSelectedItem()));
  }
```

Observer: Example Code (cont.)

- In the **public static main** :
  - Create and init the views

```
Hotel h = new Hotel(6); // Make a hotel with 6 rooms
ViewOne f1 = new ViewOne(h); // Construct a view and pass hotel as
//  the subject. Same for all views. This can be done by any
//  application!!
```

Observer: Advanced Issues

- A protocol for change notification
  - When to notify the observers
- Selective Notification
  - By Event Type
  - By information Type
  - By related objects
  - Combinations
- Many-to-Many Relationships between Observers and Subjects
  - An Observer can subscribe to more than one concrete Subject
  - A Subject can notify more than one Observer

Observer: Consequences

- Total decoupling between the Subject and the Observer
  - The minimum link is update!
  - It is possible to add concrete Subjects AND Observers
- Asynchronous communication from Subjects to dependants
  - Broadcasting, Multicasting

Observer: Consequences (cont.)

- Indiscriminate notification of change can lead to many updates
  - Information Overload for the observers
  - Many `getState()` methods for the `Subject`
- Diminished response times
  - The price of low coupling

Observer: Alternatives

- Alternatives: Push vs. Pull
  - Pull: The Subject notifies, but sends no data. The Observer retrieves all data
  - Push: The Subject notifies and sends all the changed data
  - You may combine both approaches to suit your purpose
- Alternative: Total decoupling between Subjects and Observer
  - The Observer does not directly register to the Subjects, but requests a Mediator to do this
  - The Subject does not notify the Observer, but delegates this to the Mediator
  - The Mediator maintains the registry
  - The Mediator is another place where notification decisions can be placed

Observer: Related Patterns

- Mediator:
  - Use a Mediator to totally decouple Subjects and Observers
- Singleton:
  - To control the number of Subjects
  - To control the number of Mediators
- AbstractFactory:
  - To add Subjects and Observers in run-time

Observer: Known Uses

- MVC Pattern
- Separable Model Architecture in Swing
- CORBA: used at Proxy level to decouple business objects
- Reactor Pattern

Lesson Summary

- Notification of Change
  - Unspecified number of dependant relative
  - Unspecified types of dependants
- Observer Pattern decouples Subject and Dependants
  - An interface of notification of change
- Observers know their subjects
  - They retrieve the changed data upon notification
- Indiscriminate notification can grind a system to halt
- Observer Pattern is extensible
  - Used in GUI and Distributed objects

Exercise 8 – Observe Class room events

- Follow the instructions in the Student Guide

Sinclair Community College

Lesson 2: Strategy Design Pattern

Observer Design Pattern
Strategy Design Pattern
Iterator Design Pattern
Visitor Design Pattern
Interpreter Design Pattern
Chain of Responsibility Design Pattern
Command Design Pattern
Mediator Design Pattern
State Design Pattern
Comparison and Summary

Educate. Collaborate. Accelerate!

Lesson Objectives

- Demonstrate how we can have re-usable algorithms, selected dynamically by clients

Strategy: Introduction

- Summary
  - The Strategy pattern defines a common interface for invoking tasks in general
  - It also describes how specific tasks can be created, independently from when, and how they are needed
  - Tasks conform to the common interface
  - In this way the tasks, or the Strategies can be maintained independently from their clients
- Also known as Policy

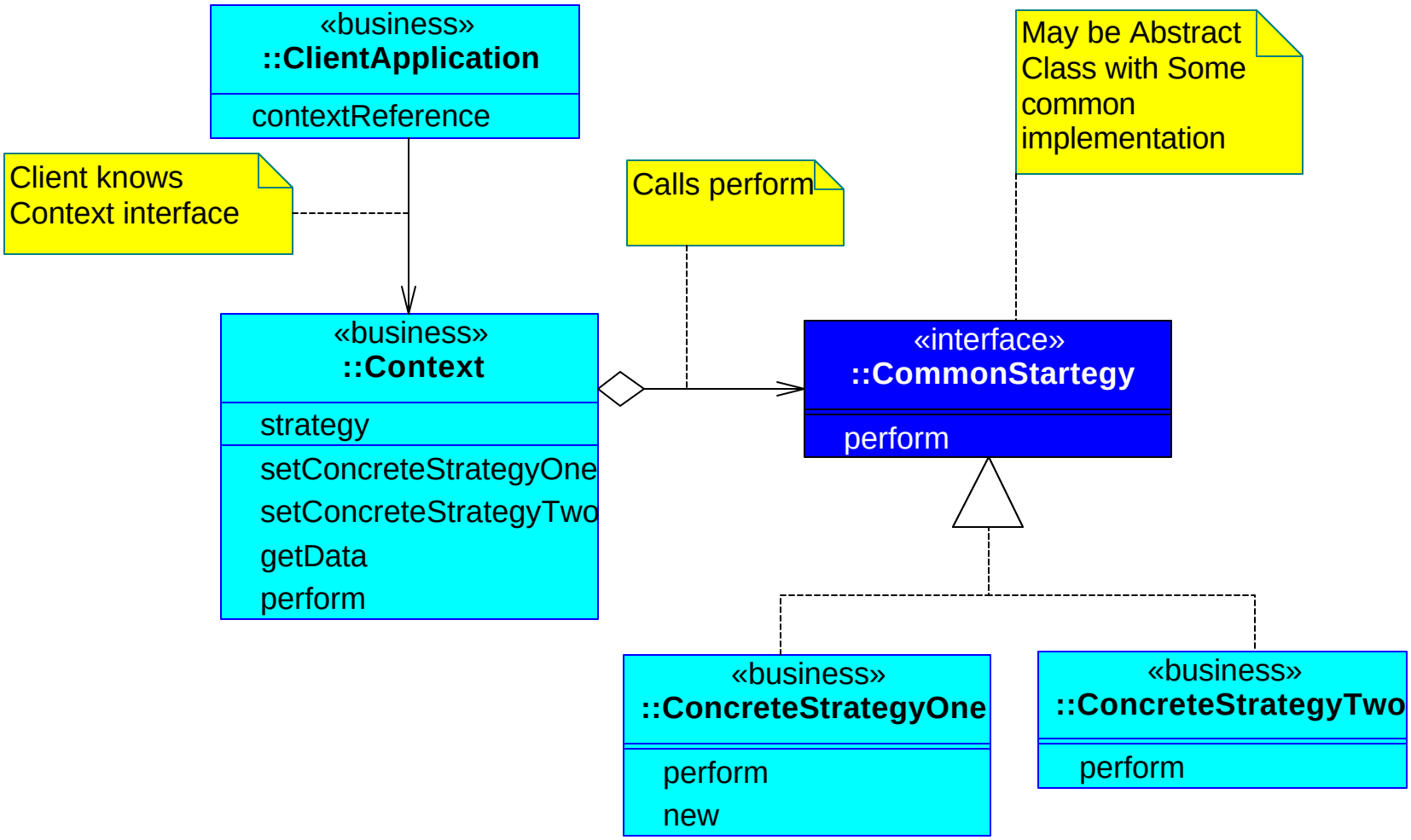
Strategy: Context and Problem

- There are potentially many different ways of doing the same thing i.e. achieving the same end result
  - Based on input for the task
  - Trade-offs between performance and size
  - Each strategy then is one way of doing the same thing, i.e. an Algorithm, or a Policy
- Problem:
  - Where are the Strategies, in the Client?
  - Strategies not interchangeable
  - code redundancy
  - Where is it decided which strategy to use? In the Client?
  - Hard coded Conditional statements
  - Complex code
  - Not possible to change in run-time

Strategy: The Solution Outline

- Separate `Strategies` from their clients
  - Analysis is required
- Design your Concrete Strategies
  - Multiple hierarchies are allowed
  - They must all implement the same interface
- Separate the Strategy Selection logic from the clients
  - Encapsulate this logic in a separate component
  - Configure this component with a set of related concrete strategies
- Separate the data transfer mechanism from Clients
  - The way the Client ultimately passes the data to a concrete Strategy
  - Where would you put this mechanism?

Strategy: Solution Structure



Strategy: Solution Components

- CommonStrategy
  - Defines an abstract interface for the context to invoke methods on concrete strategies
- ConcreteStrategy
  - Implements the Abstract Interfaces
  - Encapsulates all specific complexity
- Context
  - Knows all concrete strategies
  - Passes data to ConcreteStrategy's
- ClientApplication
  - Initiates the required Strategy
  - Retrieves the data

Strategy: Solution Dynamics

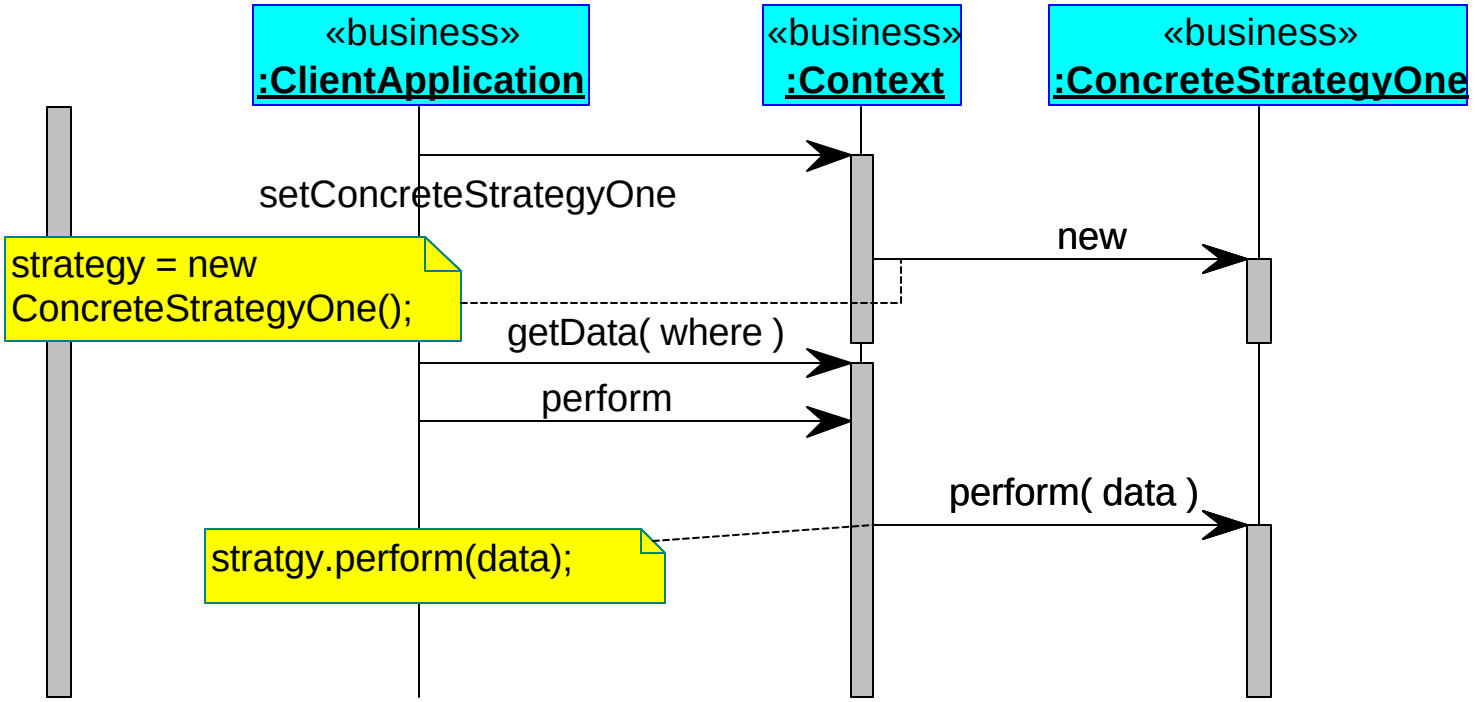
Client Needs to perform a task that requires a strategy

Instantiate the appropriate Strategy

Give the required data

Invoke the opeartion

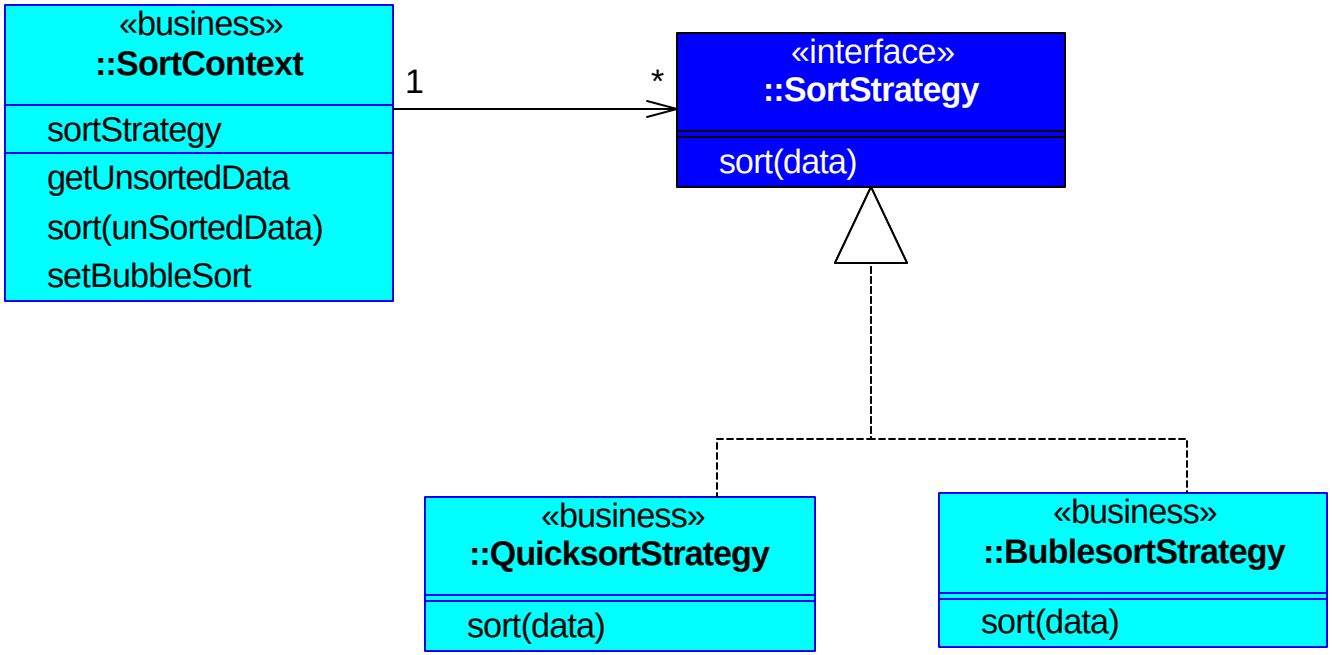
Cal the common interface to execute the strategy



Strategy: Implementation

- Design a CommonStrategy
  - An abstract class with some implementation
  - This can render multiple hierarchies
  - They can be abstracted into one interface
- Simultaneously identify and design ConcreteStrategies
  - Analyze application requirements
  - This can result in many disparate strategies
  - The ConcreteStrategies conform to the interface known by Context
  - Methods must be able to accept data as parameters
- Design Context Class
  - Context Class knows all related Strategies
  - One Context per Strategy interface may be necessary
  - Decide on how to pass the data from the Client to the Context

Strategy: Example



Strategy: Example (cont.)

- The Context class

```
public Class SortContext{

private SortStrategy sortStrategy;
private Vector unsortedData;

// Set the reference to current strategy
public setQuickStrategy(){
  SortStrategy = new QuicksortStrategy()
}

public setBubbleStrategy(){
  SortStrategy = new BubblesortStrategy()
}

public getUnsortedData(location) {
  // find read the data into the vector
}

// Another way to pass data
public getUnsortedData(Vector v) {
  // copy v to unsortedData
}

// Interface to SortStrategy
public sort() {
  SortStrategy.sort(unsortedData);
}
```

Strategy: Example (cont.)

- Client calls the Context to do a bubble sort

```
SortContext ctx
Vector vector = new Vector();
Private loadData() // Read data into vector
public void doSort () {
ctx.setBubbleSort(); //set type of required sort
ctx.getUnsortedData(vector); // pass data to context
ctx.sort(); //sort
```

Strategy: A Business Case

- A Telecom company offers various services
  - Mobile phone, Internet Subscriptions
- When a customers purchases a service:
  - The service must be provisioned
  - The appropriate identifier must be found
- Net result is the same
  - A service may be connected or disconnected
- There are differences in the implementation
  - Each service has its own server node
  - Each server may require a specific protocol
  - The required identifier types are different: Internet address vs. a new telephone number vs. an existing telephone number for Voicemail.

Strategy: Consequences

- Benefits:
  - Strategy provides a dynamic alternative to conditional statements
  - Strategy reduces the number of subclasses for each variant strategy
  - Context decouples the management of a client like adding and modification operations from the concrete strategies
- Liabilities
  - Clients are not totally decoupled from concrete strategies
  - They need to be aware of the existence of strategies
  - This may lead to a substantial amount of strategy objects
  - Excessive data passing between Context and Concrete Strategies objects is possible

Strategy: Supplementary Info

- Related Patterns:
  - Flyweight:
  - There can be too many fine-grained strategies
  - Best to share their intrinsic behavior
  - Template Method:
  - `Flyweight` and `Template` both allow the change of the internal behavior
  - `Template` by sub-classing, `Strategy` by delegation
  - `Strategy` is fine-grained, `Template` more coarse-grained
- Known Uses
  - Layout managers in AWT
  - Validation mechanisms in various GUI frameworks

Lesson Summary

- Sometimes one result can be achieved in many different ways
- It is necessary to declare one class for each way
- The mechanism of obtaining the result can be separated from the clients into a `Strategy` or `Policy`
- The clients can access these `Strategies` through the `Context` class
- In this way internal behavior of `Strategies` and their `Clients` are decoupled
- `Strategies` can be maintained without any effect on the `Clients`

Sinclair Community College

Lesson 3: Iterator Design Pattern

Observer Design Pattern
Strategy Design Pattern
Iterator Design Pattern
Visitor Design Pattern
Interpreter Design Pattern
Chain of Responsibility Design Pattern
Command Design Pattern
Mediator Design Pattern
State Design Pattern
Comparison and Summary

Lesson Objectives

- Understand how clients can traverse a set of any data elements or objects.

`Iterator`: Introduction

- `Iterator` interface
  - Provides methods for sequentially accessing the elements of a collection
  - The internal structure of the collection is unknown

Iterator: Context and Problem

- Context
  - Different types of lists of objects
  - Each list has its own internal structure
- Problem:
  - Clients are just interested in traversing through an aggregate structure
  - It is not possible to predict the structure of the aggregation in advance
  - Clients may want to traverse through a list in different orders
  - Clients may want to have a filtered view of the aggregation

`Iterator`: Solution Outline

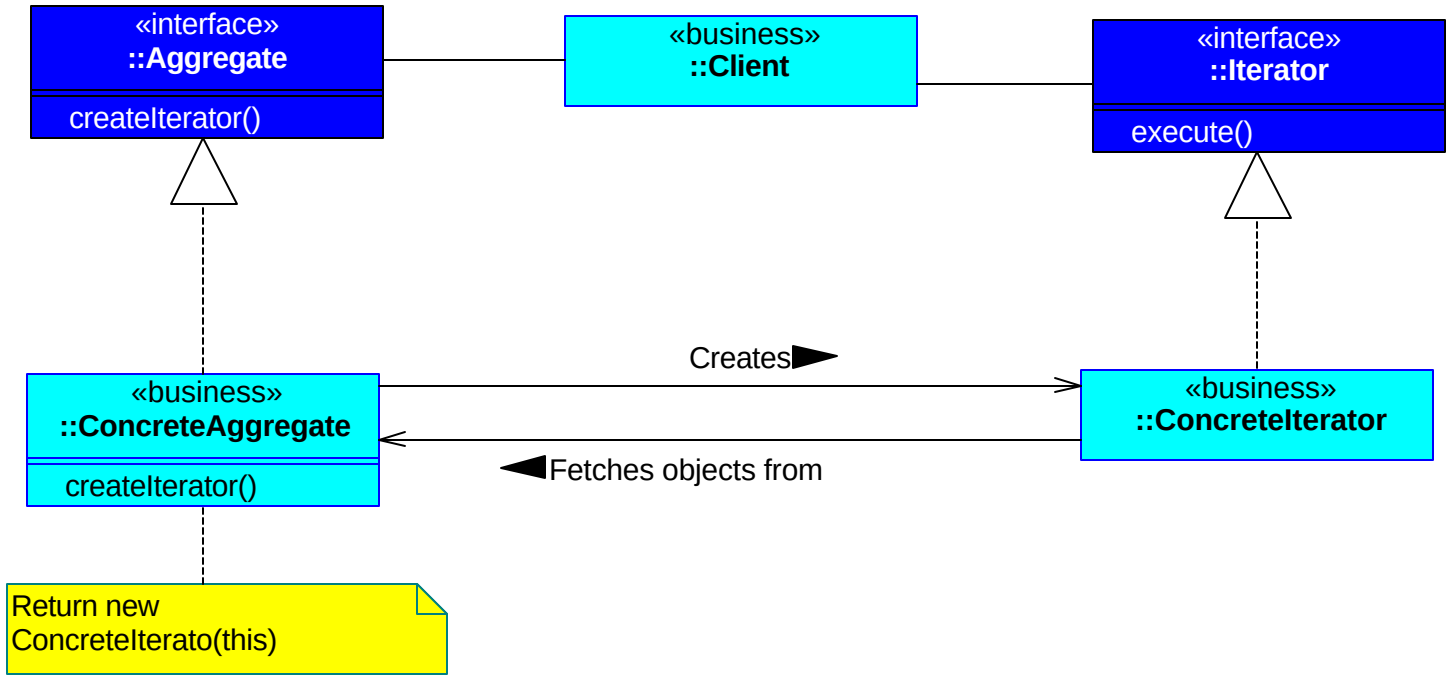
- Classes that implement the interface are separate from - but coupled to - the class that encapsulates the collection
- Clients that access the collection through the interface are independent of the class that implements the interface

`Iterator`: Solution Outline (cont.)

- Separate the responsibility for accessing objects in the list away from it
- Define a uniform interface for accessing functionality for all types of lists
  - Get, Next, current item, …
- Define Concrete `Iterators` for each type of list, while complying with the interface
- Avoid coupling of a concrete `Iterator` to a concrete list
  - A concrete list is responsible for returning the concrete `iterator` to the client

Iterator: Solution Structure

- Structure of the Iterator Pattern



`Iterator`: Solution Components

- Iterator
  - defines an interface for traversing elements of the Aggregate `ConcreteIterator`
  - implements the abstract interface defined by the Iterator class
  - tracks the current element being traversed in the aggregate object
  - Knows which objects have been traversed already
- Aggregate
  - defines an interface for creating an Iterator
  - ConcreteAggregate
  - implements the abstract interface defined by the Aggregate class
  - instantiates and returns a proper concrete iterator object

Iterator: Implementation

- Design of the minimum `Iterator` interface:
  - `first()`: Initializes the `Iterator` to point to the first element
  - `next()`: Moves the `Iterator` to the next element
  - `isDone()`: Tests if the traversing doesn't pass the last element
  - `currentItem()`: Returns the current element
- Design the abstract interface for the lists
- Design the concrete list classes
- Use a `Factory Method` to instantiate a concrete `Iterator` dynamically

Iterator: Implementation (cont.)

- Iterator may be responsible for knowing the algorithm to traverse the Aggregate object
- Aggregate object may know the algorithm and the Iterator just points to the current element
- Such an Iterator is called a cursor
- Iterator must work properly when an Aggregate object is modified while the Iterator is working
- This is usually done by registering the Iterator with the Aggregate object
- The Aggregate object notifies all of the registered Iterators when it is modified
- Multiple Iterators for the same collection allow multiple concurrent traversals

`Iterator`: Supplementary Info

- Benefits
  - Clients end up knowing two interfaces: the `Abstract List` and `Iterator` interface
  - Any change in the concrete lists of `Iterators` will not affect the code in the client
- Related Patterns
  - `Composite`: An excellent way to traverse through the composition
  - `Adapter`: A specialized adapter interface is necessary for the otherwise incompatible interface of a concrete list
  - `Factory Method`: Is used to dynamically instantiate concrete `Iterators`

Lesson Summary

- Clients could only require sequential traversal through a set of objects
  - There are various sequential data structures like lists
- Clients must not be tied to a specific type
- The Iterator pattern defines a common interface for all traversal functions
- For each type of list there is at least one concrete `Iterator`
  - Lists make them dynamically
- Multiple clients may access a list while one `Iterator` is traversing it
- There are two types of Iterators:
  - External: The client determines the current item. The client is in charge of traversal
  - Internal: The client issues a request, and the `Iterator` decides how to Traverse the list

Sinclair Community College

Lesson 4: Visitor Design Pattern

Observer Design Pattern
Strategy Design Pattern
Iterator Design Pattern
Visitor Design Pattern
Interpreter Design Pattern
Chain of Responsibility Design Pattern
Command Design Pattern
Mediator Design Pattern
State Design Pattern
Comparison and Summary

Educate. Collaborate. Accelerate!

Lesson Objectives

- Learn how to design classes independent of operations that apply to them
- Learn to concentrate the logic distributed over multiple classes in a single place

`Visitor`: Introduction

- Summary
  - In a complex structure of objects that all have to perform some action, one way to implement this is to place the logic for each of these objects in the objects themselves
  - You can take the logic of the action out of the objects, and place it in a separate `Visitor` object that visits the objects in turn
  - This way the logic may be varied by using different `Visitor` classes for different objects

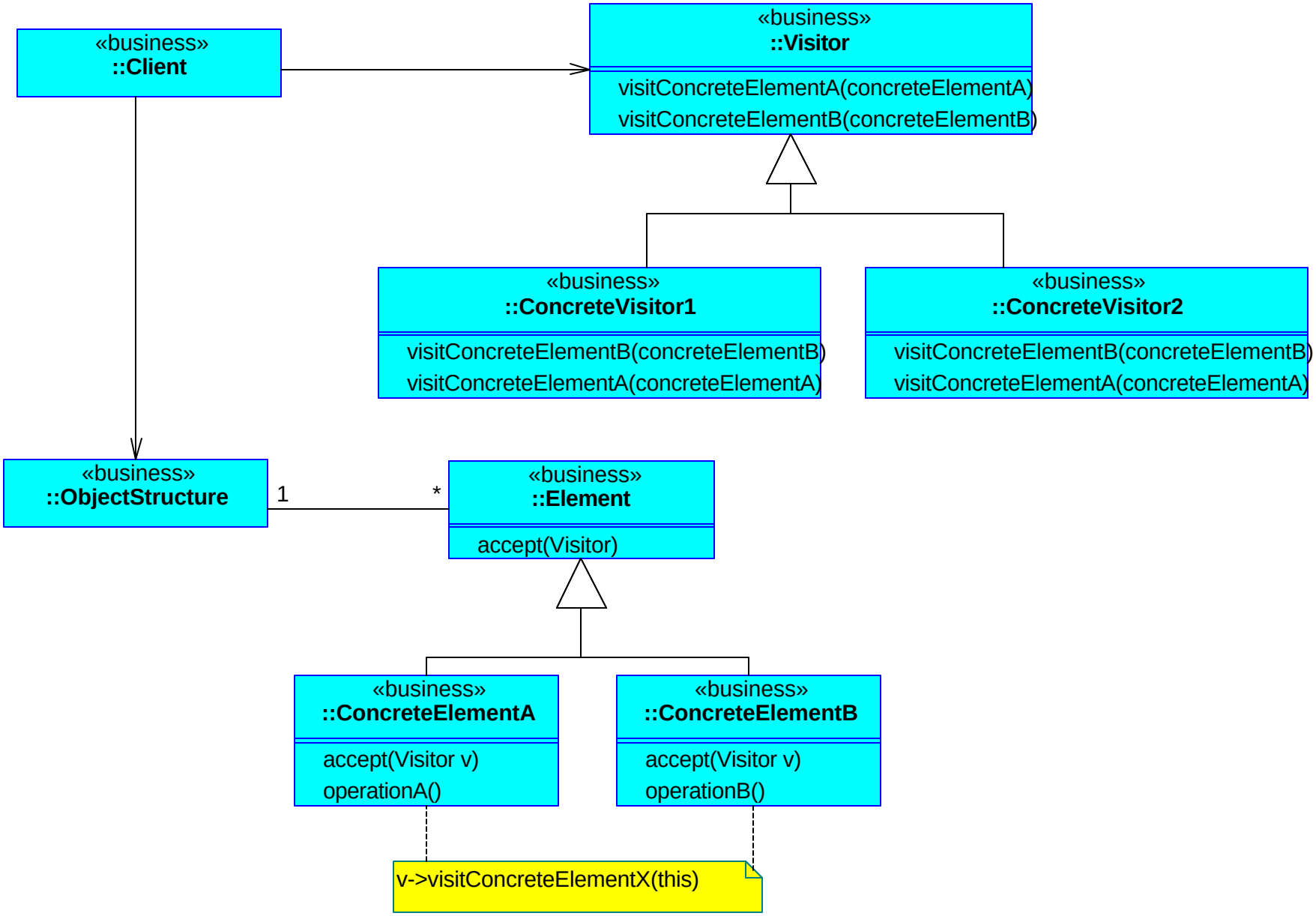
`Visitor`: Context

- Consider the example depicted in the `Composite` Design Pattern
  - We have a set of classes that represent different objects in the grammar
  - Each class has a function `evaluate()` that is responsible for evaluating the expression tree
- The logic of this operation is distributed over multiple classes

`Visitor`: Problem

- It is hard to understand how this operation works because one has to look in multiple places
  - The code can be made a lot easier if all of the logic responsible for the evaluation of the expression tree is concentrated in a single place
- Adding a new operation to all the nodes is difficult.
  - Requires a modification of the super class and each node plus recompilation of a lot of code
  - An example would be type-checking our expression
- The `Visitor` design pattern lets us solve these problems by moving the logic out of the “Expression” classes

Visitor: Solution



`Visitor`: Solution Outline

- The class diagram represents the `Visitor` pattern
- There are two sets of classes: `Visitor` and `Element`
- The `Element` family of classes
  - The `Element` tree hierarchy represents the objects we want to operate on
- The `Visitor` family groups together the various operations that need to be performed on each Element

`Visitor`: Solution Components

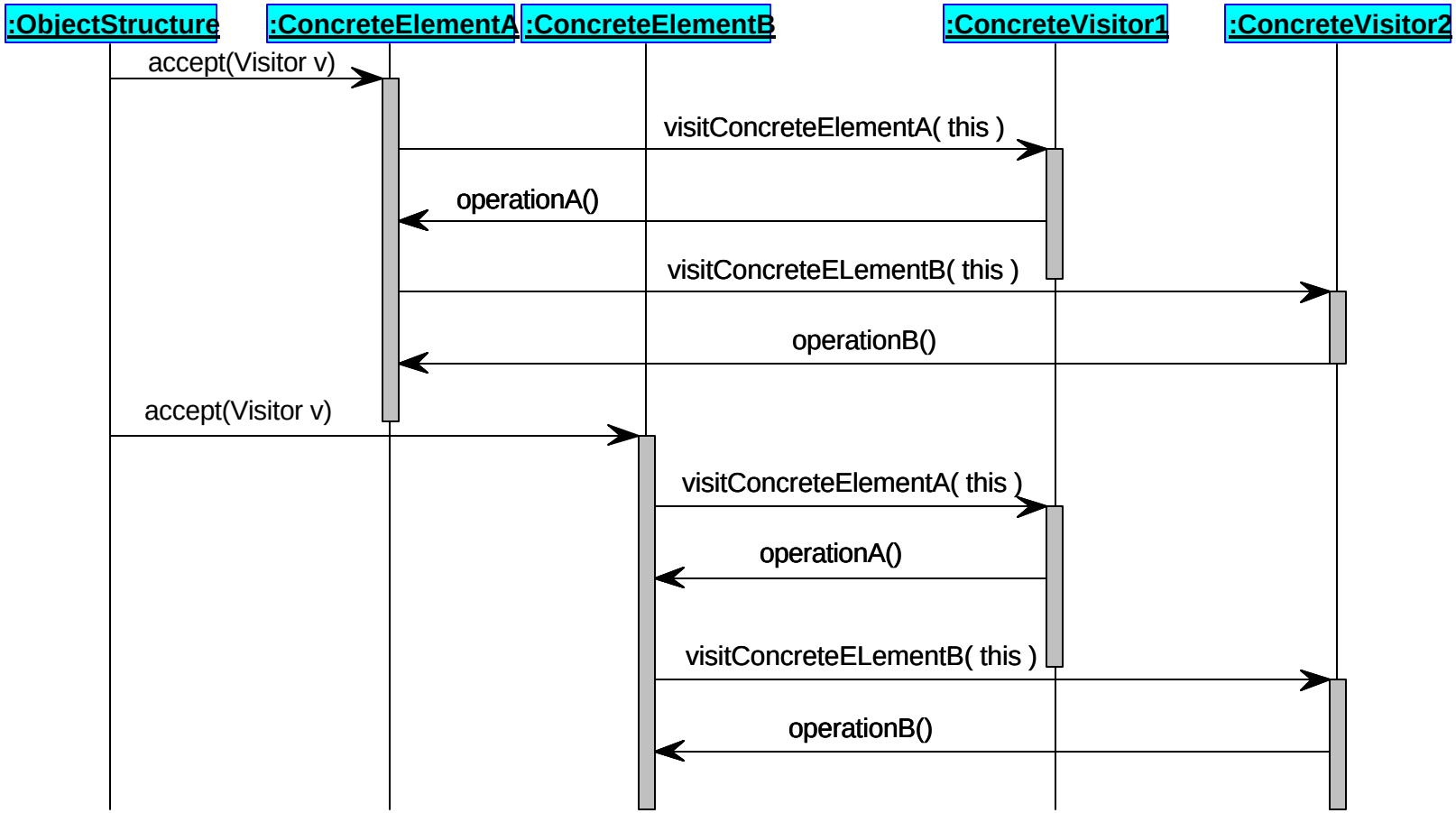
- `Visitor`
  - Declares a `visit()` operation for each class of `ConcreteElement`
  - The operation name and signature identifies the sender of the request
- `ConcreteVisitor`
  - Each `ConcreteVisitor` implements all of the functionality for a particular operation
  - Each operation of the class implements the functionality for a particular `ConcreteElement`
  - Each `ConcreteVisitor` may have a context for the operation it is performing

Visitor: Solution Components (cont.)

- Element
  - Defines an accept() method that takes a Visitor as an argument
- ConcreteElement
  - Defines an accept() method that takes a Visitor as an argument
- ObjectStructure
  - Used to enumerate all ConcreteElements
  - Allows the Client to visit all of the Elements with a specified ConcreteVisitor
  - May be either a composite or a collection of some type

Visitor: Solution Dynamics

- Interaction diagram for two Element's and two Visitor's



`Visitor`: Implementation

- We want to calculate total sales and total income for a set of cars
- There are three different types of cars and three different real cars
- We can perform two functions on each car
  - Calculate sales
  - Calculate income
- Hence we will make use of two `ConcreteVisitors` objects
- This is referred to as "double dispatching"

Visitor: Code Example

```
abstract class Product {
  abstract public void accept( Visitor visitor );

  public int getCount() { return count; }
  protected int count;
}


class EconomyCar extends Product {
  EconomyCar( int count ) {
    this.count = count;
  }

  public void accept ( Visitor visitor ) {
    visitor.visitEconomyCar( this );
  }
}
```

Visitor: Code Example (cont.)

```
class LuxuryCar extends Product {

  LuxuryCar( int count ) {
    this.count = count;
  }

  public void accept ( Visitor visitor ) {
    visitor.visitLuxuryCar( this );
  }
}

class SportsCar extends Product {
  SportsCar( int count ) {
    this.count = count;
  }

  public void accept ( Visitor visitor ) {
    visitor.visitSportsCar( this );
  }
}
```

Visitor: Code Example (cont.)

```
abstract class Visitor {

  abstract public void visitEconomyCar( EconomyCar car );
  abstract public void visitLuxuryCar( LuxuryCar car );
  abstract public void visitSportsCar( SportsCar car );

  protected final int economy_car_cost = 10000;
  protected final int economy_car_sale = 12000;
  protected final int luxury_car_cost = 17000;
  protected final int luxury_car_sale = 20000;
  protected final int sports_car_cost = 20000;
  protected final int sports_car_sale = 25000;

}
```

Visitor: Code Example (cont.)

```
class IncomeVisitor extends Visitor {

  IncomeVisitor() { total_income = 0; }

  public int getTotalIncome() { return total_income; }

  public void visitEconomyCar( EconomyCar car ) {
    total_income += car.getCount() * ( economy_car_sale - economy_car_cost );
  }

  public void visitLuxuryCar( LuxuryCar car ) {
    total_income += car.getCount() * ( luxury_car_sale - luxury_car_cost );
  }

  public void visitSportsCar( SportsCar car ) {
    total_income += car.getCount() * ( sports_car_sale - sports_car_cost );
  }

  private int total_income;

}
```

Visitor: Code Example (cont.)

```
class SaleVisitor extends Visitor {

  SaleVisitor() { total_sale = 0; }

  public int getTotalSale() { return total_sale; }

  public void visitEconomyCar( EconomyCar car ) {
    total_sale += car.getCount() *  economy_car_sale;
  }

  public void visitLuxuryCar( LuxuryCar car ) {
    total_sale += car.getCount() * luxury_car_sale;
  }

  public void visitSportsCar( SportsCar car ) {
    total_sale += car.getCount() * sports_car_sale;
  }

  private int total_sale;

}
```

Visitor: Code Example (cont.)

```
class Visitor_example1 {

  public static void main (String argv[]) {
    EconomyCar   economyCar = new EconomyCar(20);
    LuxuryCar    luxuryCar = new LuxuryCar(7);
    SportsCar    sportsCar = new SportsCar(3);

    IncomeVisitorincomeVisitor = new IncomeVisitor();
    SaleVisitor  saleVisitor = new SaleVisitor();

    economyCar.accept( incomeVisitor );
    economyCar.accept( saleVisitor );
    luxuryCar.accept( incomeVisitor );
    luxuryCar.accept( saleVisitor );
    sportsCar.accept( incomeVisitor );
    sportsCar.accept( saleVisitor );

    int total_sales = saleVisitor.getTotalSale();
    int total_income = incomeVisitor.getTotalIncome();

    System.out.print("Total sales are: ");
    System.out.println( total_sales );
    System.out.print("Total income is: ");
    System.out.println ( total_income );
  }
}
```

`Visitor`: Advanced Implementation Issues

- Many distinct and unrelated operations exist that can be performed on objects in the class hierarchy.
  - Do not mix these operations with the rest of the code
  - Use one `ConcreteVisitor` class to group all of the related operations from all of the classes into one place
- Classes rarely change but new operations are added often
  - This process should be made as painless as possible
  - Adding a new operation to all of the classes requires adding a new `ConcreteVisitor` class
- Keeps business logic and data traversal logic separate

`Visitor`: Consequences, Advantages

- It is easy to add new operations to all of the classes
- Related operations are not spread all over the classes but are gathered in one place
- You can use `Visitor` to iterate through the objects of different classes
- Each `Visitor` object can accumulate the state for each object it visits. Without the `Visitor` pattern, the state would have to be passed as an extra parameter
- For example: flow analysis, statistics gathering

`Visitor`: Consequence, Disadvantages

- It is difficult to add a new concrete element
- Requires modifying the interface of the abstract `Visitor` class and all of the concrete classes
- Each `ConcreteVisitor` needs to know a lot of details about each element
  - E.g. the `getCount()` method in our example
- This requires a powerful interface for the element class

Class Review

- What do you need to do to be able to calculate the cost of all cars in the “Cars” example?
- What do you need to do to add a new type of a car in the “Cars” example?
- Why is Visitor Design Pattern not very applicable when the class hierarchy is constantly changing?
- How can a Visitor pattern force you to break encapsulation in your design?

Lesson Summary

- `Visitor` Design Pattern helps to group all of the related functionality of many objects in a single place
- Adding new functionality to a group of classes does not require modification of their interfaces.
- Each `Visitor` must have knowledge of each of the classes it operates on

Exercise 9 – Display and Evaluate an Arithmetic Expression

- Follow the instructions in the Student Guide

Sinclair Community College

Lesson 5: Interpreter Design Pattern

Observer Design Pattern
Strategy Design Pattern
Iterator Design Pattern
Visitor Design Pattern
Interpreter Design Pattern
Chain of Responsibility Design Pattern
Command Design Pattern
Mediator Design Pattern
State Design Pattern
Comparison and Summary

Educate. Collaborate. Accelerate!

Lesson Objectives

- Write programs where, for a given language, an interpreter will interpret sentences in the language
- Build programs that make use of the interpreter pattern
- Extend our calculator example to evaluate an expression using the interpreter design pattern

Interpreter: Introduction

- Summary
  - A language defines a representation for its grammar, along with an interpreter that uses the representation to interpret sentences in the language

Interpreter: Context

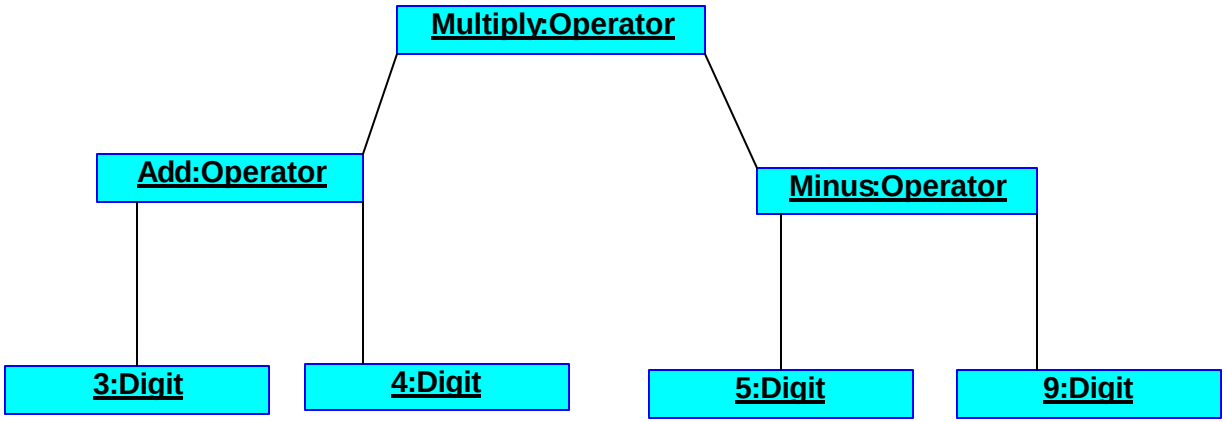
- A literal operator in a class field specifies the behavior
- A little language needs to be interpreted
  - Language statements can be represented as abstract syntax trees
- Grammar of the language is simple
  - Efficiency is not of a great concern
- Practical examples include pattern matching, OO compilers, file formats, evaluating constraints, etc.

Interpreter: Problem

- We are going to expand on the problem presented in the Composite Design Pattern
- In the Composite Design Pattern we learned how to build a composite object for each expression that we want to evaluate

Interpreter: Problem (cont.)

- Structure of the syntax tree for the expression 3 + 4 * 5 - 9

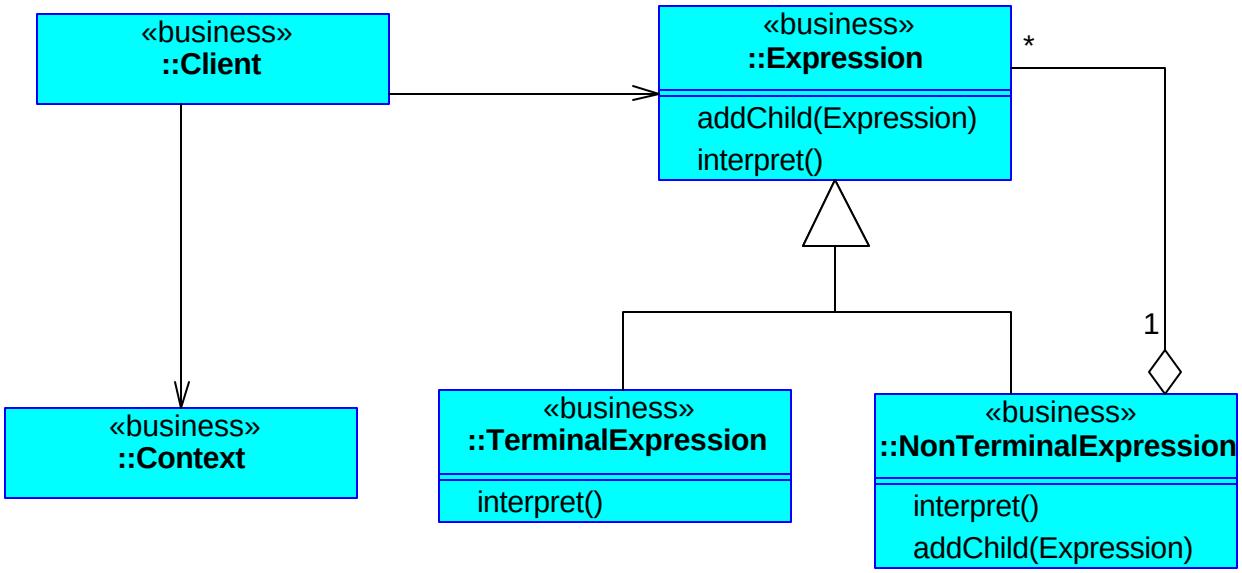


- Each object has an interpret() method which return a result integer
  - The algorithm is object type specific
  - For a Digit it returns the digit
  - For a PlusOperator it adds it left and right children's results etc.

Interpreter: Solution Outline

- The Interpreter Design Pattern uses the Composite Design pattern to solve its objective
- Composite patterns show us how to build composite objects
- Interpreter pattern shows us how to solve a particular problem
- Both patterns have a very similar UML diagram
- Composite pattern is used to build a composite object of the expression we are trying to evaluate.
- We then apply the ideas of the Interpreter pattern to evaluate the composite object

Interpreter: Solution Structure



Interpreter: Solution Components

- AbstractExpression
  - defines operations that are common to all nodes
- TerminalExpression
  - corresponds to the terminal symbol in a grammar (digit).
  - One TerminalExpression object exists for every terminal symbol in the expression
- NonTerminalExpression
  - represents one rule of the grammar
- Context
  - contains information global to the interpreter
- Client
  - Builds a tree composed of Terminal and NonTerminal expression objects

Interpreter: Consequences

- Advantages
  - It is easy to change and extend the grammar of a language
  - It is easy to implement a new grammar from scratch
- Disadvantages
  - It is hard to handle a large grammar because of the large number of classes involved
  - A trade off has been made between convenience and performance

Class Review

- How would you add a “sine” function to the calculator grammar?
- Think of where you can apply an Interpreter pattern?
- Does an `Interpreter` pattern scale well?
- How an implementation of the unary operator would differ from “standard” operators?

Lesson Summary

- The `Interpreter` Design Pattern shows how one can evaluate simple grammars with use of the `Composite` Design Pattern
- The `Interpreter` Design Pattern is very useful for fast implementation of simple grammars

Exercise 10 – `Interpreter` Pattern applied to an Arithmetic Expression

- Follow the instructions in the Student Guide

Sinclair Community College

Lesson 6: Chain of Responsibility Design Pattern

Observer Design Pattern
Strategy Design Pattern
Iterator Design Pattern
Visitor Design Pattern
Interpreter Design Pattern
Chain of Responsibility Design Pattern
Command Design Pattern
Mediator Design Pattern
State Design Pattern
Comparison and Summary

Educate. Collaborate. Accelerate!

Lesson Objectives

- Demonstrate how it would be possible to uncouple senders of request from a number of possible handlers.

Chain of Responsibility: Introduction

- Summary:
  - Allows an object to send a request without knowing which object or objects will process it. The request is sent to a chain of objects.

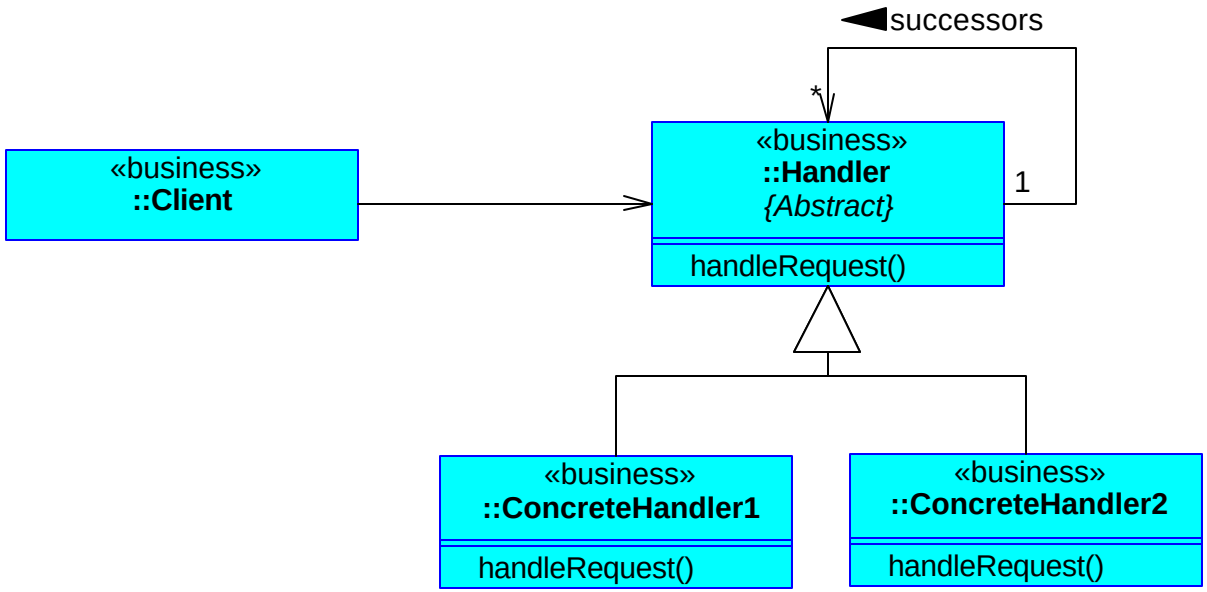
Chain of Responsibility: Context and Problem

- Context
  - A class hierarchy, or a chain of objects
  - They all can handle the same requests to various degrees
  - For example: A help system
- The Problem:
  - The initiator of a request does not know in advance the specific object that handles the request

Chain of Responsibility: Solution Outline

- Identify the related existing objects that need to be chained together
- If they are not linked then implement the links
- Decide on the requirements for the request format:
  - Simple invocations
  - Passing identifiers to handler classes
  - Separate Request objects
- Make sure that there is at least one handler that can handle the request

Chain of Responsibility: Solution Structure



Chain of Responsibility: Solution Components

- Handler
  - defines an interface to handle Client requests
  - implements the link to the Successor handler
- ConcreteHandler
  - implements the interface defined by the Handler class
  - services requests that it is responsible for
  - passes requests it cannot handle to its successor
- Client
  - is responsible for initiating a request to one of the concrete handlers in the chain

Chain Of Responsibility: Solution Dynamics

- Each object in the chain will either process the request or pass it on to the next object in the chain
- An incoming request propagates up in the `Chain of Responsibility` until it can be handled
- Eventually the root, or the last Handler in the chain MUST handle the request

Chain of Responsibility: Consequences

- Benefits
  - Several objects in the chain can handle the same request
  - The recipient of the request is not explicitly specified
  - The `Chain of Responsibility` enables the change of the request handling at run time
- Liabilities
  - It is possible that a request cannot find an appropriate receiver

Chain of Responsibility: Related Patterns

- Composite:
  - Can be used to implement the chain
  - But in composite the traversal is top-down
- Command:
  - Can be used to encapsulate a request

Lesson Summary

- A request may be handled by more than one object
- These objects are linked
- We don’t want to make the sender of a request dependent on a specific Handler
- The `Chain of Responsibility` pattern allows a request to be passed through the chain, until it reaches its proper `Handler`

Sinclair Community College

Lesson 7: Command Design Pattern

Observer Design Pattern
Strategy Design Pattern
Iterator Design Pattern
Visitor Design Pattern
Interpreter Design Pattern
Chain of Responsibility Design Pattern
Command Design Pattern
Mediator Design Pattern
State Design Pattern
Comparison and Summary

Lesson Objectives

- Understand the Command Pattern, and demonstrate how they allow us to treat requests like objects

Command: Introduction

- Summary
  - encapsulates a request as an object
  - parameterization of requests
  - queuing or logging requests
  - support undoable operations
- Also known as `Action`, `Transaction`

Command: Context

- Command dynamically tells objects what to do, and leaves it to the object to decide how to do it
- Command objects can be stored for future use
  - Undo-redo
- Command lifecycle is totally decoupled from the lifecycle of the sender

Command: Problem

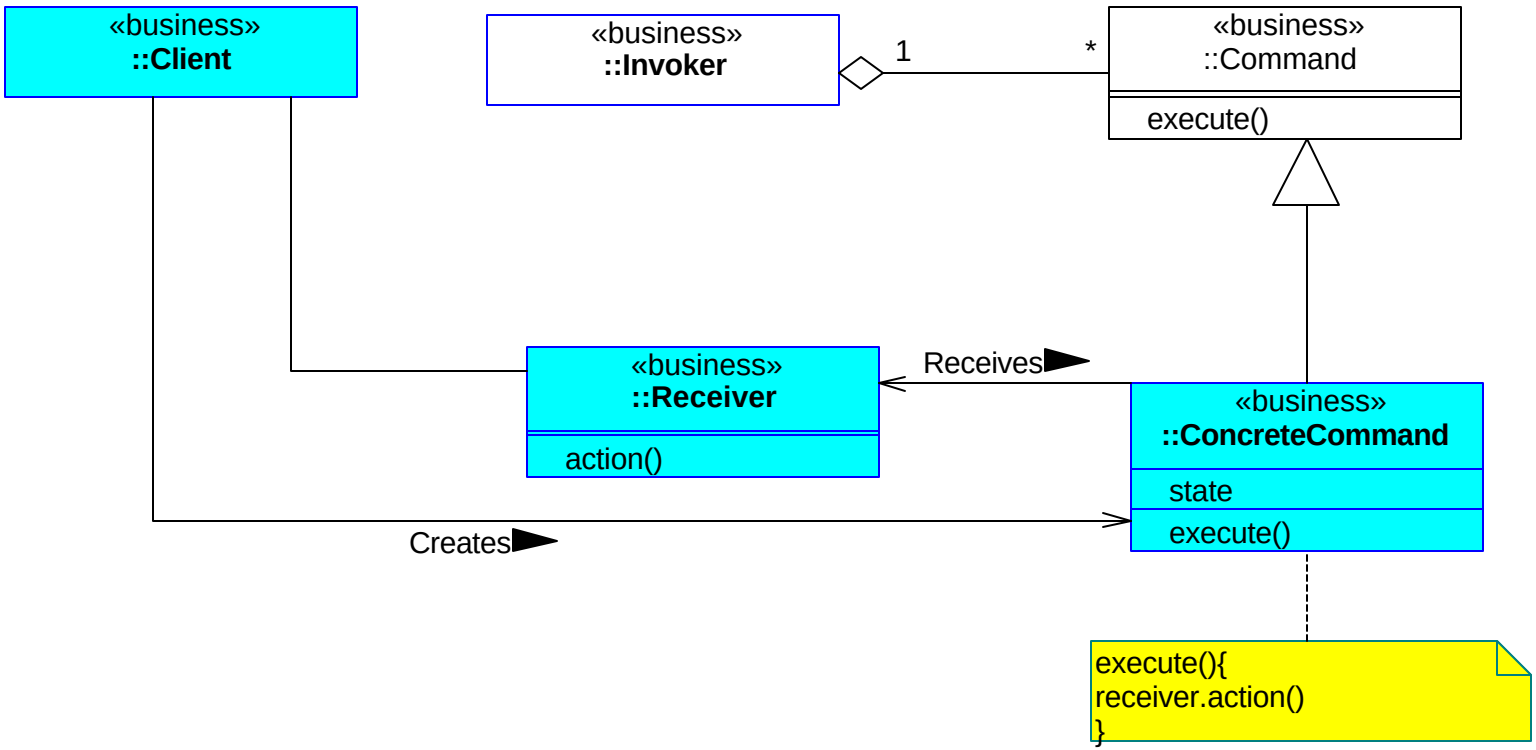
- The sender only needs to know what needs to be done (the command)
  - The sender doesn't need to know how the command will be handled
- The receiver only needs to know how to initiate the execution of the command
  - No knowledge of the internal logic of the command is necessary
- A command may not be executable when it is requested
  - It must be saved, so that it can be performed later
- Different requests can ask for the same command
  - Commands could be re-used amongst objects
- One command may be implemented in many ways
  - Based on the receiver, the sender, …

Command: Solution Outline

- Separate the command and encapsulate this in a Command object
- Define an interface for manipulating the Command
  - The receivers of the Command use this interface
  - Methods: execute, un-execute, suspend, etc.
- Identify and design ConcreteCommands
  - They implement the Command Interface
  - ConcreteCommand classes are responsible for actually performing the Command

Command: Solution Structure

- This UML diagram shows the structure of the Command Pattern

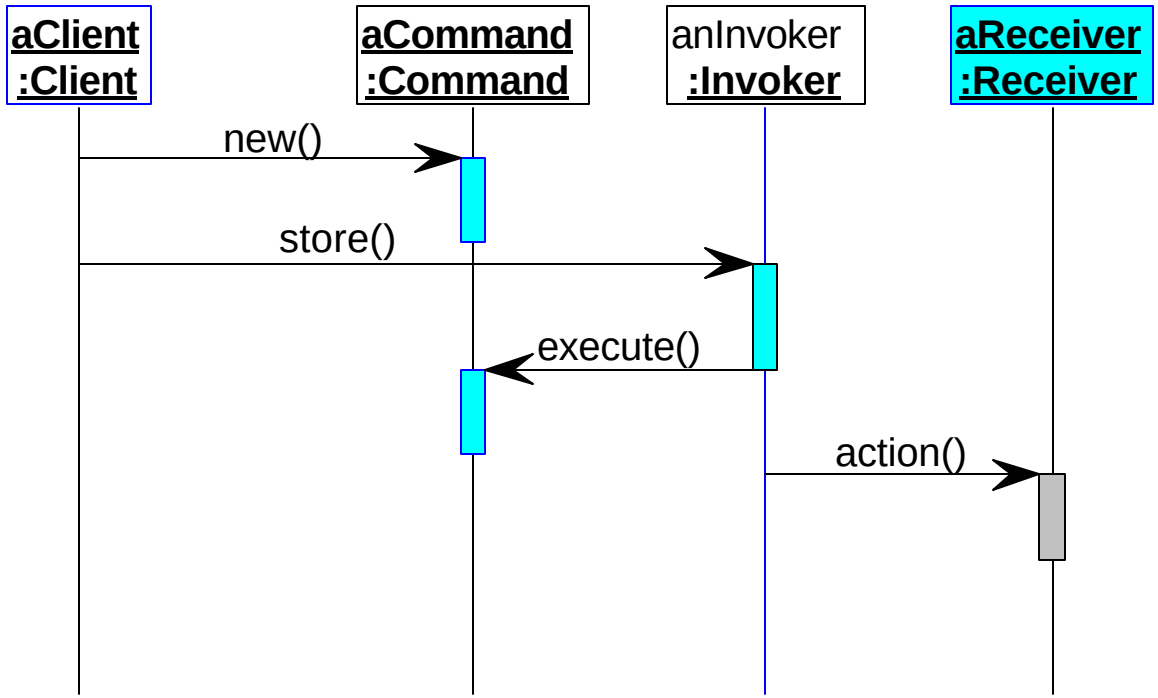


Command: Solution Components

- Command
  - defines an interface to execute an operation
- ConcreteCommand
  - implements the interface defined by the Command class
  - knows how to associate an action with an appropriate Receiver object
  - implements the execute() method by using an interface on the appropriate Receiver object
- Client
  - responsible for creating a ConcreteCommand
  - identifies the receiver of the command
- Invoker
  - responsible for asking a ConcreteCommand to execute the request
- Receiver
  - used by the ConcreteCommand to perform operations necessary to finish the request
  - any class may act as a Receiver

Command: Solution Dynamics

- Sequence of Events



Command: Consequences

- The sender of the request does not need to know the receiver of the request
- The request is an object
  - You can assemble composite commands at run time to create new types of requests
  - You can use the `Composite` pattern to build composite `Command` objects
- By sub-classing your request you can add new types of requests
- Some requests should have a capability to perform undo and redo operations.
  - The `ConcreteCommand` class should provide a way to store the state before and after request execution

Sinclair Community College

Lesson 8: Mediator Design Pattern

Observer Design Pattern
Strategy Design Pattern
Iterator Design Pattern
Visitor Design Pattern
Interpreter Design Pattern
Chain of Responsibility Design Pattern
Command Design Pattern
Mediator Design Pattern
State Design Pattern
Comparison and Summary

Educate. Collaborate. Accelerate!

Lesson Objectives

- Understand how to reduce coupling amongst objects, that may occasionally need to collaborate

Mediator: Introduction

- Summary
  - One object mediates the state changes between two or more objects
  - The logic is localized in the mediator rather than being spread across the objects themselves
  - Decreases the coupling between the other objects
  - Implementation is more cohesive

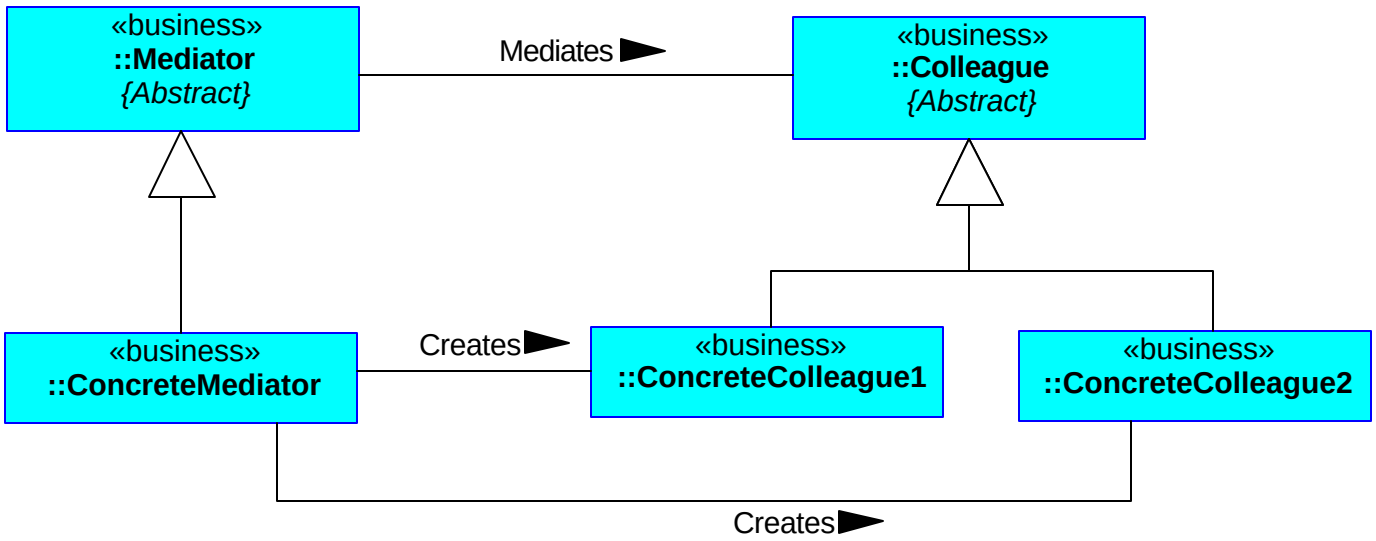
Mediator: Context and Problem

- Context
  - There is an accidental need for collaboration
  - The collaboration amongst objects is complex
- Problem if the collaboration logic is in the objects themselves
  - High coupling: Many objects may require to “know” many other objects
  - Maintenance problem: Modifying the collaboration amongst the object will ripple through many objects

`Mediator`: Solution Outline

- Abstracts the details of the communication protocol
- Gathers all of the details of a communication protocol in a single place
- Introduces one-to-many versus many-to-many communication scheme
- The Objects in a system are aware only of the `Mediator`
- An object starts a communication protocol with a `Mediator`
- The `Mediator` communicates with the initiator and all other objects needed to be involved in this communication

Mediator: Solution Structure



Mediator: Components

- Mediator
  - Defines an interface for the communication between Colleague objects
- ConcreteMediator
  - Implements the interface defined by the Mediator class
  - Implements a concrete communication protocol between Colleague objects
  - Keeps track of all the Colleague objects
- Colleague objects
  - A Colleague object is aware of the ConcreteMediator but it is not aware of other Colleague objects
  - A Colleague object communicates with the ConcreteMediator only

Mediator: Consequences

- The communication protocol can be changed by sub classing the Mediator class
  - If you do not use the Mediator class you need to subclass all of the Colleague objects
- A change in one of the Colleague classes propagates only as far as a Mediator
- All control is centralized in a single class
- A one-to-many communication scheme is easier to understand than a many-to-many scheme
- You do not need an abstract Mediator class if you only have a single version of the communication protocol

Sinclair Community College

Lesson 9: State Design Pattern

Observer Design Pattern
Strategy Design Pattern
Iterator Design Pattern
Visitor Design Pattern
Interpreter Design Pattern
Chain of Responsibility Design Pattern
Command Design Pattern
Mediator Design Pattern
State Design Pattern
Comparison and Summary

Educate. Collaborate. Accelerate!

Lesson Objectives

- Understand how an object may seem to have changed its object in run-time.
- Understand why it is useful to know whether the object changed its state
- Understand the `State` Design Pattern

`State`: Introduction

- Summary
- Objects do change their behavior in run-time
  - The behavior is based on the state they are in
  - Simple states can be handled by an internal “state manager” operation
  - Complicated state behavior favorites the usage of a separate `State` object
- Also known as: Objects for states

`State`: Examples

- Network Protocols
  - Many states and state-transitions
  - Connecting, Connected, Disconnected
- Roles
  - A person traverses an organization in a particular role
  - For each role there is a different response to the subsequent events during the traversal
  - Each role can be implemented as a `State` object

State: Context

- An object changes its “role” during its lifecycle
- The implementation of a final state machine within an object
- An Object that has complex internal state transitions
  - The behavior and appearance change as the state changes

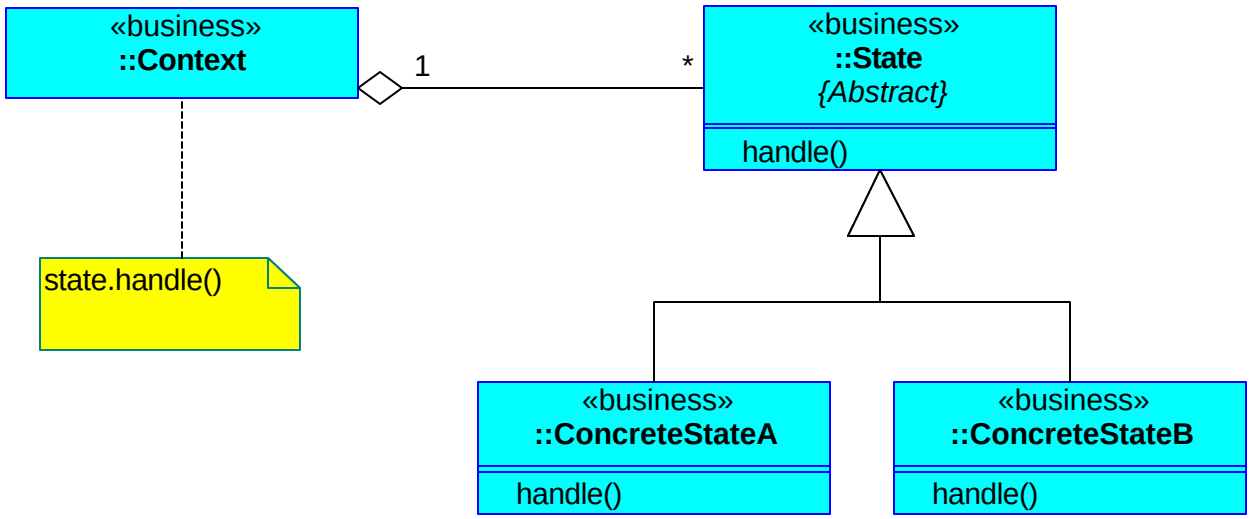
State: Problem

- State management of objects results in a complex set of operations with large conditional statements

`State`: Solution Outline

- Identify which objects have complex behavior
- Separate the Responsibilities (attributes and operations) of objects from their internal behavior
- Analyze and capture the objects' behavior in a state transition diagram
  - A `ConcreteState` Object can encapsulate each state
- Apply polymorphism to the `State` objects
  - Design a `State` object hierarchy
  - A `State` interface is the top of the hierarchy
  - `ConcreteState` objects implement the interface
- Decide on the state transition strategy
  - Encapsulating transitions as objects with references to the states OR
  - Encapsulate all transitions in a separate `Mediator` object

State: Solution Structure



State: Solution Components

- Context
  - Defines an interface to be used by clients
  - Keeps track of the current State by keeping track of the corresponding ConcreteState object
- State
  - Defines an interface to provide access to encapsulated behavior of the State
- ConcreteState subclass
  - Each ConcreteState implements a partial interface provided by the State class
  - Only the interface that is necessary for this particular state of the Context is implemented

State: Implementation

- Design the Abstract State Class for the target object
  - Define all methods to be implemented in the ConcreteState objects
- Enforce Enter state and Exit state controls
  - Conditions and actions for entering and exiting states
  - Each ConcreteState may implement these two
  - An alternative is a “change state” operation
- Identify the initial and final states
  - On instantiation the object is set to this state
- Each ConcreteState is responsible for creating itself
  - Declare a protected constructor in the abstract class

`State`: Implementation (cont.)

- Declare all variables referring to states as `static`
- Where is the transition defined?
  - A look-up table
  - A `Mediator` class interacting with the `Context`
  - In the `State` objects

State: Related Patterns

- Mediator
  - It can be used to decouple various state objects from one another
- Flyweight
  - This pattern can be used when there are too many states, or when states can be shared amongst different objects
- Singleton
  - Use a Singleton when you need precisely one instance of a state object

Sinclair Community College

Lesson 10: Comparison and Summary

Observer Design Pattern
Strategy Design Pattern
Iterator Design Pattern
Visitor Design Pattern
Interpreter Design Pattern
Chain of Responsibility Design Pattern
Command Design Pattern
Mediator Design Pattern
State Design Pattern
Comparison and Summary

Educate. Collaborate. Accelerate!

Lesson Objectives

- Describe when to use each Behavioral Pattern

Common Trend Among Behavioral Patterns

- Behavioral Patterns
  - Deal with the assignment of responsibilities between objects
  - Hide the details of how objects communicate in the system
  - Focuses your attention on how objects are interconnected
- Some patterns (Command, State) encapsulate the behavior of an object and describe how to delegate a request
- `Mediator` pattern hides all peers from each other and introduces an intermediate object for communication
  - (many-to-many versus many-to-one)
- `Chain of Responsibility` pattern hides the receiver of the request from the sender

Encapsulation

- Used by many Behavioral Patterns to represent a part of the program that changes frequently
- These patterns usually have an abstract class to describe an encapsulating object
- `State` pattern encapsulates the state-dependent behavior of the system
- `Iterator` pattern encapsulates how you access and traverse an aggregate object
- `Mediator` pattern encapsulates the communication protocol between objects in a system

Objects as Arguments

- The `Visitor` pattern has a visitor object that is passed as an argument
- The `Visitor` object holds the functionality that was previously distributed across multiple objects
- The `Command` pattern has a command object passed around
- This object contains information needed to execute an operation

Decoupling Senders from Receivers

- If the Sender has an explicit knowledge of who receives and handles its requests it becomes dependent on it
- Behavioral patterns help us to remove this dependency
- The `Command` pattern uses a `Command` object to define the relationship between the sender and the receiver
- The `Mediator` pattern forces the sender to refer to the receiver indirectly through the `Mediator` object
- The `Chain of Responsibility` pattern provides a chain of potential receivers of the request to the sender

Class Review

- The implementation of a `Mediator` pattern in a distributed system increases the efficiency. Is this true?
- Discuss which behavioral patterns decrease the performance of the system.
- Discuss situations where you can apply several behavioral patterns simultaneously
- Discuss situations where you can apply behavioral and creational patterns simultaneously

Lesson Summary

- Most of the behavioral patterns complement each other and can be combined during the design of the system
- The `Chain of Responsibility` pattern can be used with a `Command` pattern to represent request objects
- The `Interpreter` needs to keep track of the parsing context. The `State` pattern can be used for that
- Behavioral patterns also can be combined with non-behavioral patterns
- `Composite` and `Visitor` patterns can be combined together as we have seen in the calculator example
- You can use the `Builder` or `Prototype` pattern to build composite objects
- All behavioral patterns replace complex conditional code with simple polymorph method calls to other objects

Sinclair Community College

Addendum: Exploring Frameworks

Introduction to Frameworks

Frameworks Illustration: JDBC

Frameworks Illustration: Swing

Lesson Objectives

- Understand why we use frameworks
- Describe the basic characteristics of frameworks
- Understand how frameworks relate to patterns and other software artifacts

Patterns and Productivity

- Patterns enhance productivity
  - Analysis Patterns: Schema's containing specific Domain Knowledge
  - Architectural Patterns: Schema's for fundamental structural organization and behavior of Software Systems
  - Design Patterns: Schema's for refinement of the internal structure of subsystems and components, and their internal dynamics
- Patterns focus on generic design:
  - Schema = a model or an abstraction of something
  - Reusing abstract (micro) architectures
- Patterns do not guarantee reuse of code

Other Approaches to Re-use

- Re-use of Concepts
  - Ideas
  - Architectural and design Styles
  - Heuristics
  - Best practices
  - Patterns
- Re-use of (binary) code
  - Program statements
  - Functions and Procedures
  - Classes and Objects
  - Class libraries and Packages
  - Components
  - Sub-systems
  - Frameworks

Other Approaches to Productivity

- Class Libraries
  - Mostly provide stateless services
  - Always involve extension and/or instantiation
  - A white box approach
  - Fine grained
  - Examples: Algorithms, Collections
- Component Libraries
  - Provide both state-full and stateless services
  - A Black-box approach
  - Fine-grained
  - Examples: JavaBeans, Delphi and Active-X components
- Frameworks

Frameworks: An Introduction

- A partly developed (sub)system
  - Frameworks are not meant to contain 100% required functionality
  - They are used as the basis for developing or generating complete applications
  - Abstraction of a specific domain
- Targets a specific area:
  - Vertical Domains
  - Telecom OS, DOT, GUI, Finance
  - Horizontal Domains
  - Three major target areas: Infrastructure, Middleware, Domain Specific

Frameworks: An Introduction (cont.)

- Provide the fundamental architecture
  - Provide the fundamental ingredients i.e. (abstract) classes, interfaces, components
- Encapsulate significant behavior
  - Invariant (static) behavior: Uniform behavior in every instantiation of the framework
  - Variant behavior: Specialized behavior in different situations, implemented by design patterns
- Frameworks are active
  - They express run-time behavior that is not initiated by the client application

Frameworks: How We Use Them

- As an SDK, or class library:
  - Instantiation of Classes
  - Extension of Classes and Interfaces
  - Implementation of Interfaces
- Configuration or parameterization
  - Install Time
  - Run-time
- Calling Services on Components
- Adding Functionality in the run-time

Frameworks: Requirements

- Simple
  - A framework should not have a long learning curve
- Flexible
  - Frameworks must offer enough flexibility for application specific behavior
- General
  - A framework must be applicable to slightly different cases
  - Frameworks should not determine the requirements for these cases
- Specific
  - Justification of the investment requires the availability of certain features in a specific framework

Frameworks: Benefits

- Higher Productivity
  - Re-use of architecture, design and functionality
- Reduce the possibility of errors
  - A Framework is already tested
- Reduce the time for application customisation
  - Through hot-spots and plug-in

Frameworks: The Liabilities

- Frameworks may require high Investment
  - In order to find the right level of abstraction, developing a framework requires a lot of research
  - Its usage will call for a long learning curve, courses and mentoring
- Maintaining Frameworks is a daunting task
  - There may be numerous applications that depend on the framework

Frameworks: The Liabilities (cont.)

- Difficult to debug
  - The logic is encapsulated in the framework
  - It is difficult to validate generic behavior in compile time
- May result in poor performance
  - This is the price for adaptability and changeability
- Difficult to integrate with other frameworks
  - Various levels of encapsulation
  - Incompatible technologies

Frameworks: Working with Other Approaches

- Patterns can make a Framework
  - OO Techniques may not be enough
  - Design Patterns are used the achieve the desired qualities
- Class Libraries
  - Are used within frameworks

Frameworks: Working with Other Approaches

- Components are the building blocks of Frameworks
- Behavior of Frameworks is encapsulated in components
- Frameworks can be used to develop components
- A whole framework can be encapsulated in a Component!

Frameworks: Compared to Other Approaches

- Compared with Patterns
  - Patterns: Generic, Abstract, Loosely Coupled, Fine Grained, leverage bottom-up approach
  - Frame Works: Specific, Concrete, Tightly Coupled, Coarse Grained, encourage top-down approach
- Compared with Class Libraries
  - Class libraries: Generic, Passive
  - Frameworks are: Specific, Active
- Compared with Components
  - Components: black-box, loosely coupled
  - Framework: black box and white-box, tightly coupled

Frameworks: Compared with Applications

- Frameworks are used to develop Applications
- Applications
  - Are more concrete
  - Have more features
  - Are easier to develop
  - Are developed for end-users
  - Adaptable only through configuration
  - Can only have (parameterized) invariant behavior
- Frameworks
  - Are meant to be flexible and extensible
  - Are designed to outlast many applications
  - Are developed for re-use
  - Can be manipulated through a variety of techniques

Lesson Summary

- There are different Approaches for enhancing Productivity
- Frameworks are the most mature approach
- Design Patterns can be used to implement Frameworks
- Frameworks realize the adaptability and changeability requirements
- Frameworks are not the same as applications

Sinclair Community College

Frameworks Illustration: JDBC

Introduction to Frameworks

Frameworks Illustration: JDBC

Frameworks Illustration: Swing

Educate. Collaborate. Accelerate!

Lesson Objectives

- Illustrate a few Framework concepts, using JDBC as an Example
- Develop a basic understanding of JDBC
- Discuss JDBC as a Framework

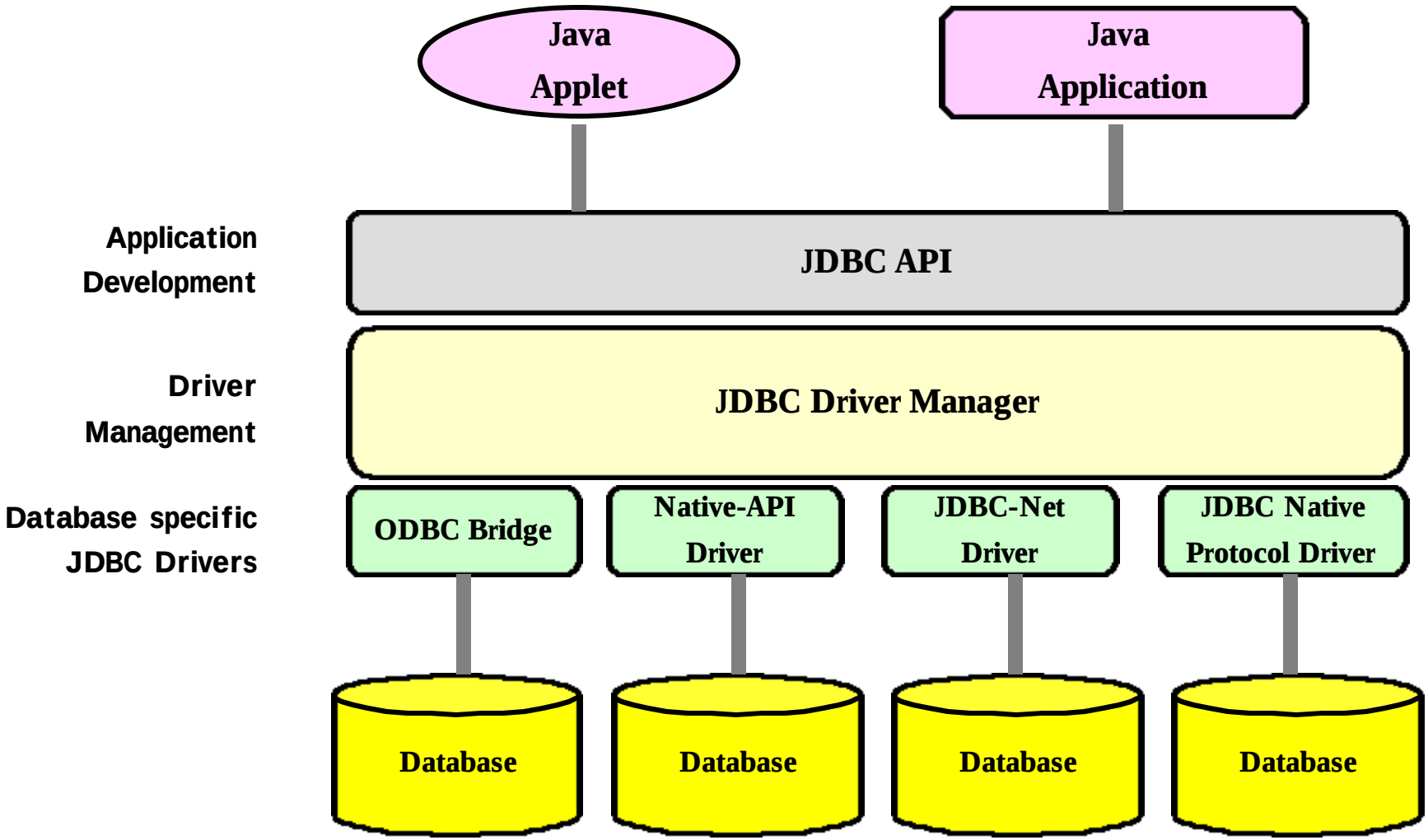
JDBC: Introduction

- JDBC provides uniform database access to java applications
- DBMS independent
- Run-time DBMS driver Management
- Run-time database connection management

JDBC: Introduction (cont.)

- Transaction Management
- Submit DDL and DML statements
- Retrieval of results of SQL statements
- Type mapping between Java and SQL types
- Obtaining relational schema information (metadata)

JDBC: Architecture



JDBC Drivers

- Database vendors supply the JDBC drivers
  - Sun defines the interfaces they have to comply with
- Includes java class (jar) files
  - Drivers may include native and Java files
- Each Driver must implement java.sql.Driver
- JDBC compliancy
  - SQL 92 standard

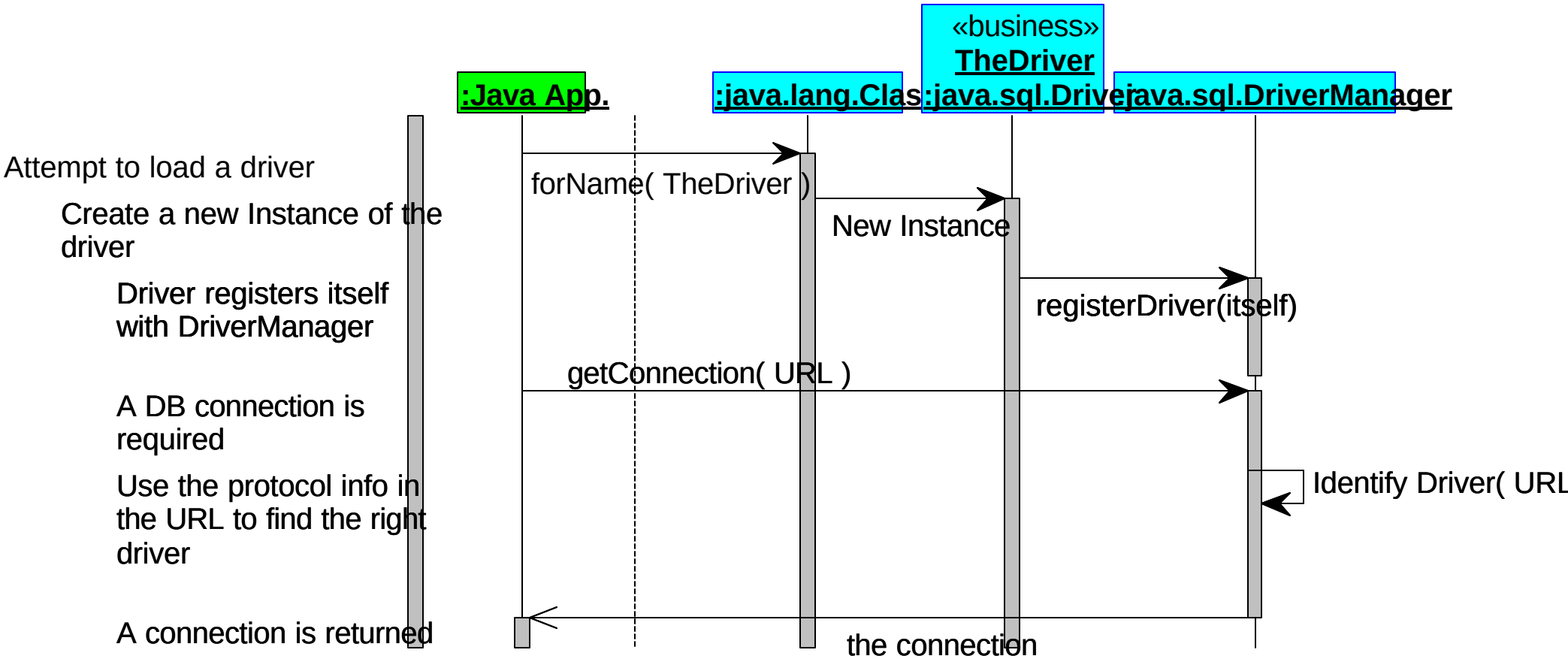
JDBC: Driver Types

- JDBC-ODBC Bridge
- Native API
- JDBC-Net pure Java
- Native Protocol

JDBC: JDBC Driver Loading

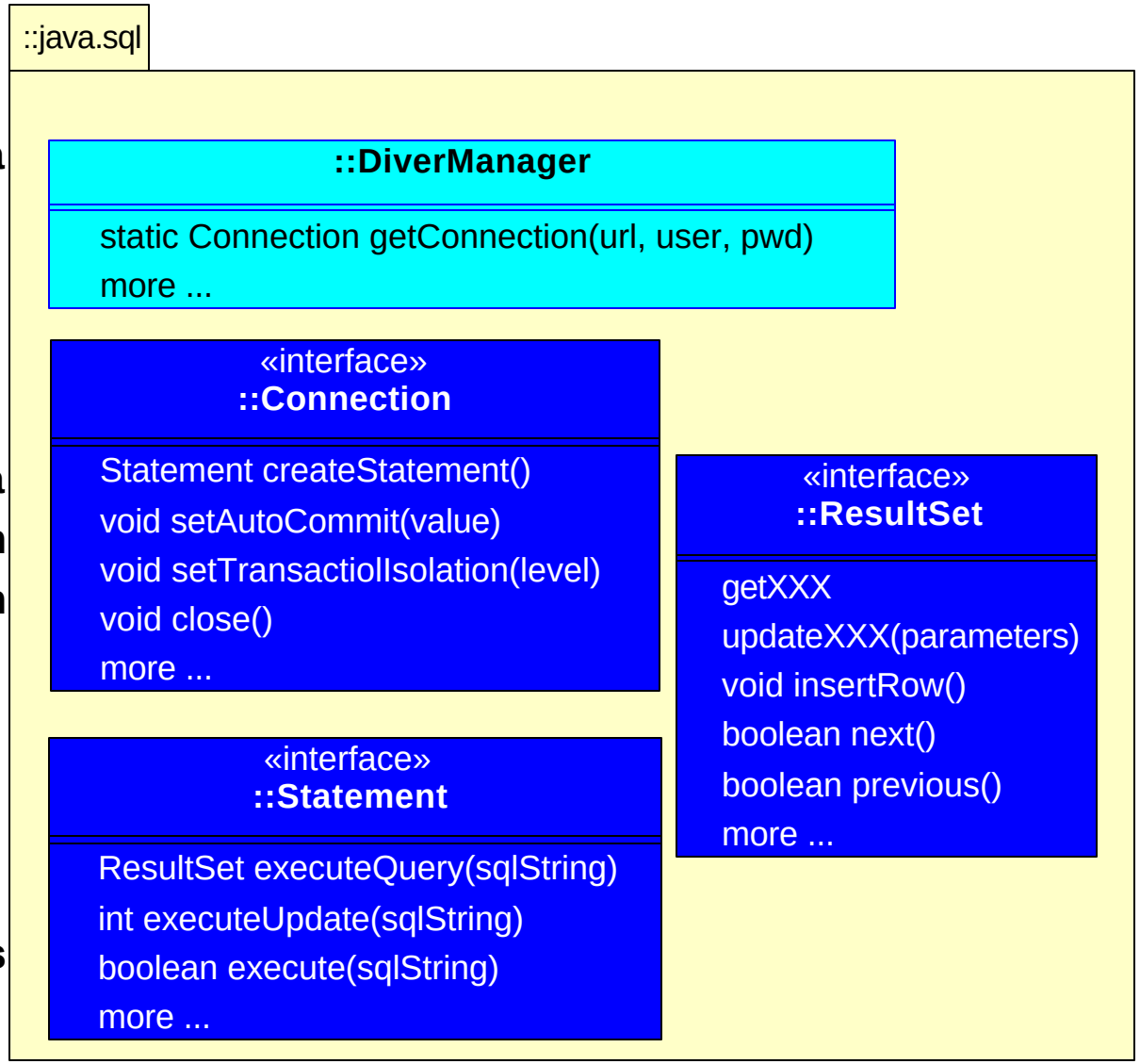
Run-time Driver Management

- Dynamic Loading
- Automatic Driver Identification



JDBC: The Application Developers API

- A set of Classes and Interfaces



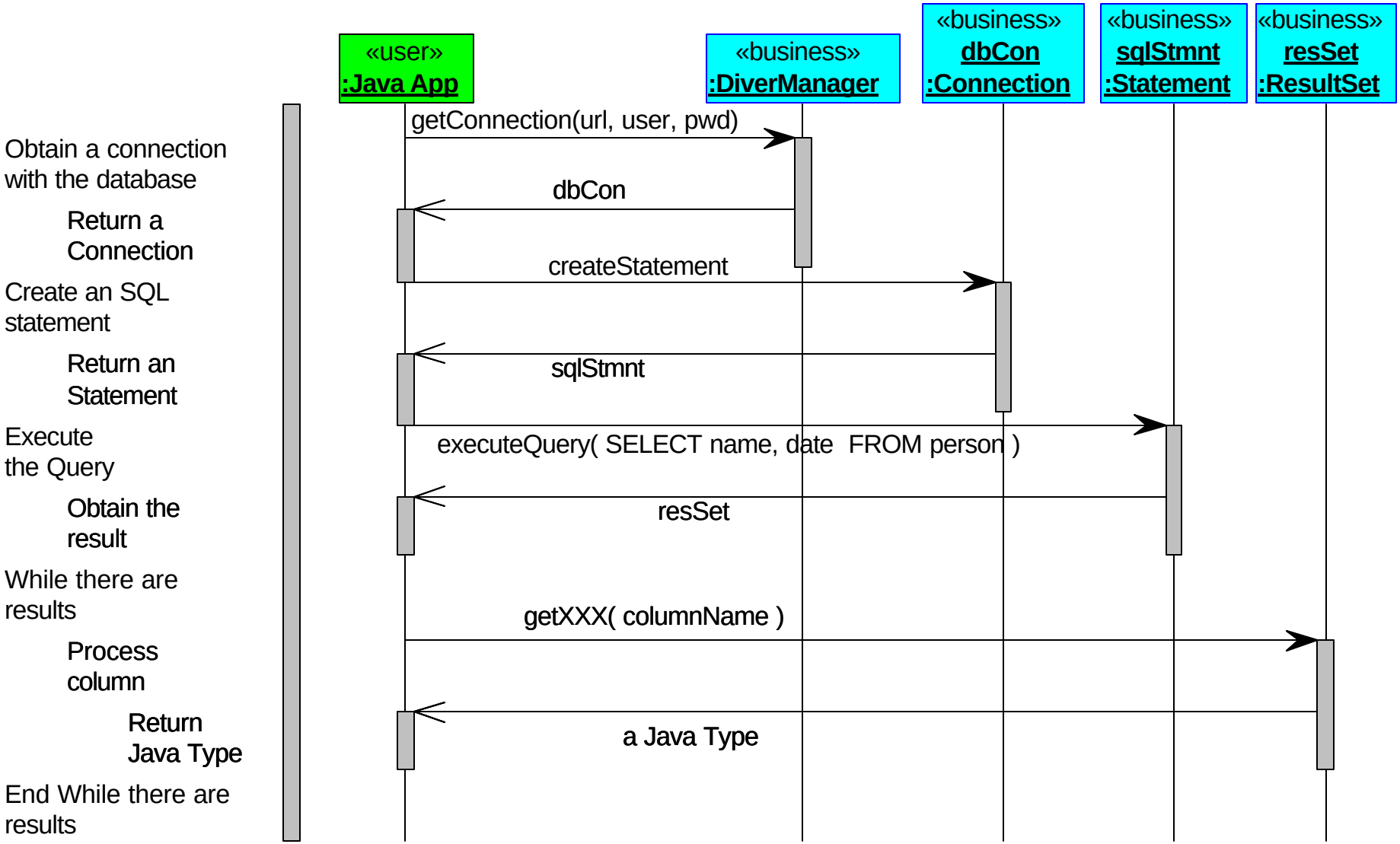
Management of a Collection of JDBC Drivers

Set of Services for Management of a Single Database session Or Connection

Set of Services for executing SQL statements

A set of services for manipulation And navigation Of Results of An Executed SQL statement

JDBC: Typical Scenario



Lesson Summary

- JDBC Target area is Database Access from java applications
  - Infrastructure Level
- Provides a basic Architecture
  - Classes, interfaces in the `java.sql` package
  - Data Type Mapping, Transaction Management
  - Driver Management
  - Plug-in functionality to add new drivers
- JDBC Qualities:
  - Quick to learn
  - Extensible
  - Provides useful services
  - Applicable in many Application architecture scenario's

Lesson Summary (cont.)

- As an SDK
  - Various methods on the interfaces in `java.sql` can be invoked
  - Provides for the implementation of interface drivers
  - Instantiation of various objects
- No Configuration required
- Driver Management Component
- Adding new drivers in run-time

Sinclair Community College

Frameworks Illustration: Swing

Introduction to Frameworks

Frameworks Illustration: JDBC

Frameworks Illustration: Swing

Educate. Collaborate. Accelerate!

Lesson Objectives

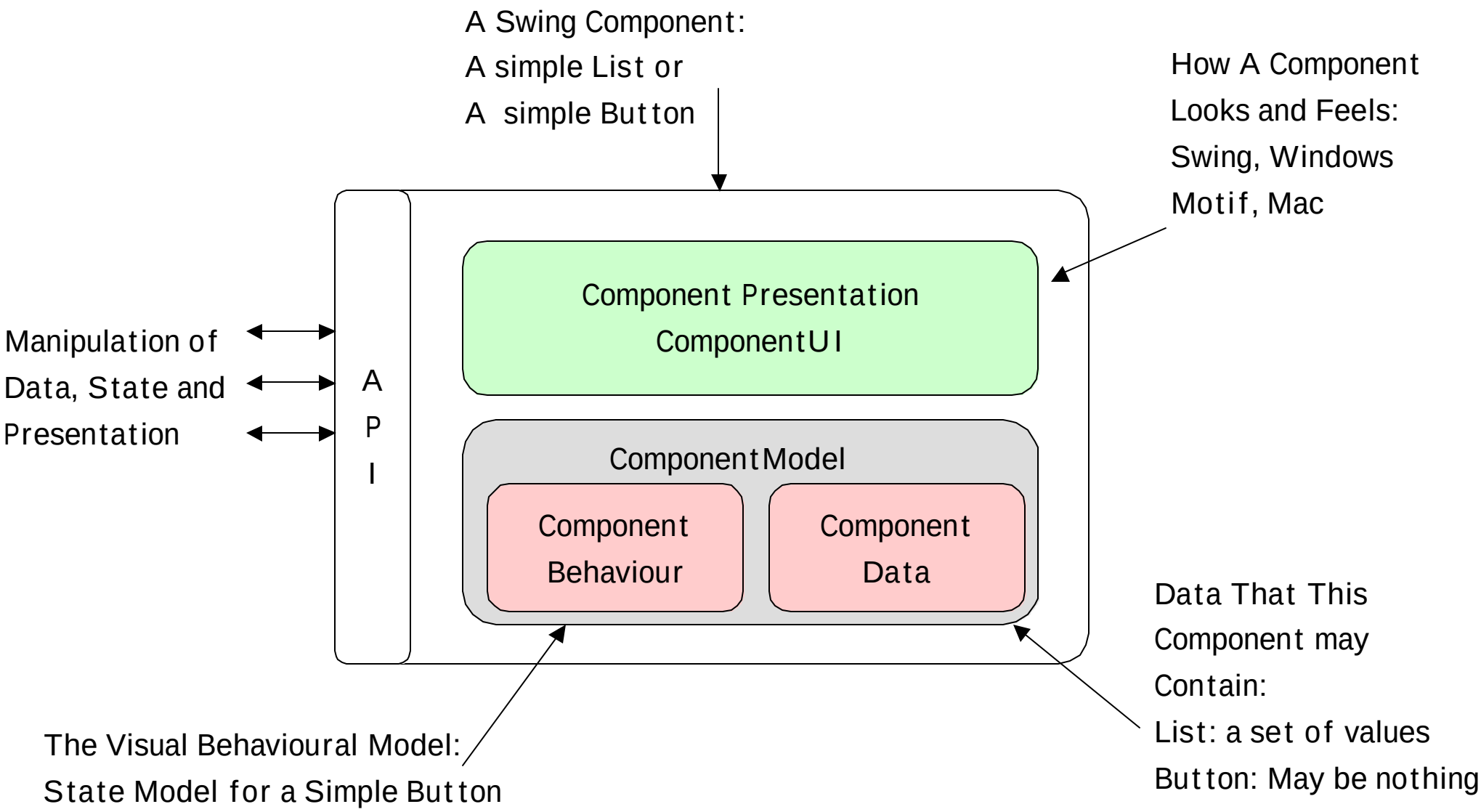
- Develop a basic understanding of Swing
- Appreciate the objectives of Swing
- Demonstrate how Design Patterns help to achieve Swing's objectives

Swing: Overview

- A Framework for GUI development
- Major Objectives
  - Eliminate the need for native code
  - Separate Look&Feel from GUI functionality; pluggable Look & Feels
  - Lightweight Components -> Portability
- A huge library of GUI components or widgets
  - The base class: `JComponent`
  - Enforces common component architecture
  - Buttons, Lists, Trees, Frames, Tables
- Common micro-architectures
  - Essential design and architectural patterns
  - Model-View-Controller at the core
- Pluggable Look and Feel Architecture
  - Package `javax.swing.plaf`
  - Package `javax.swing.plaf.xxx`

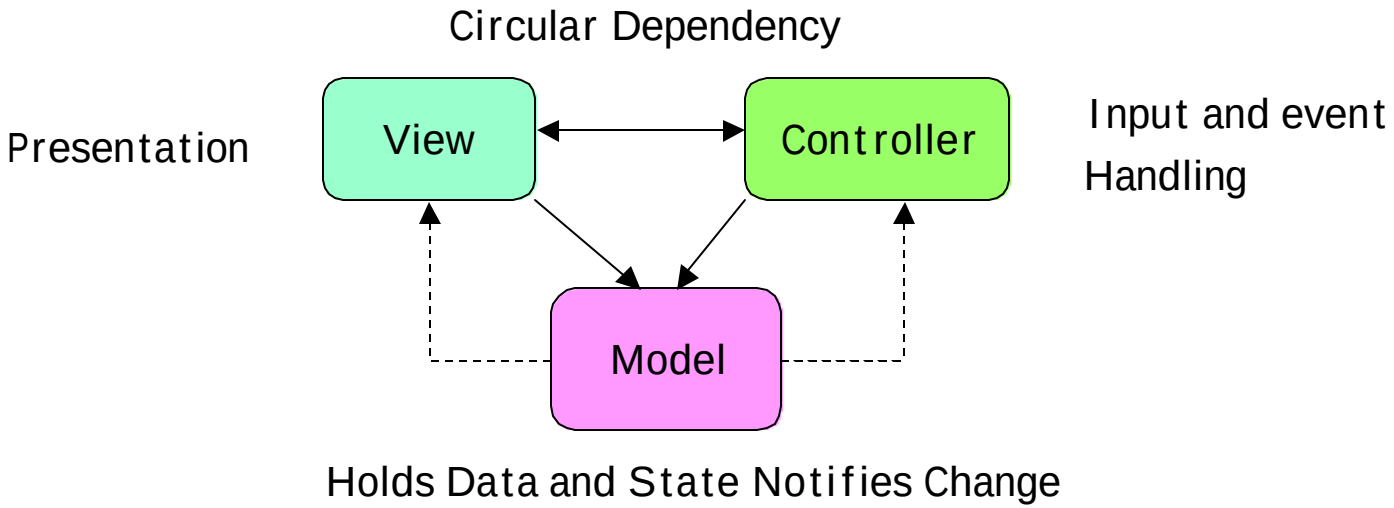
Swing: Common Component Architecture

- All Swing Components have the same fundamental Architecture

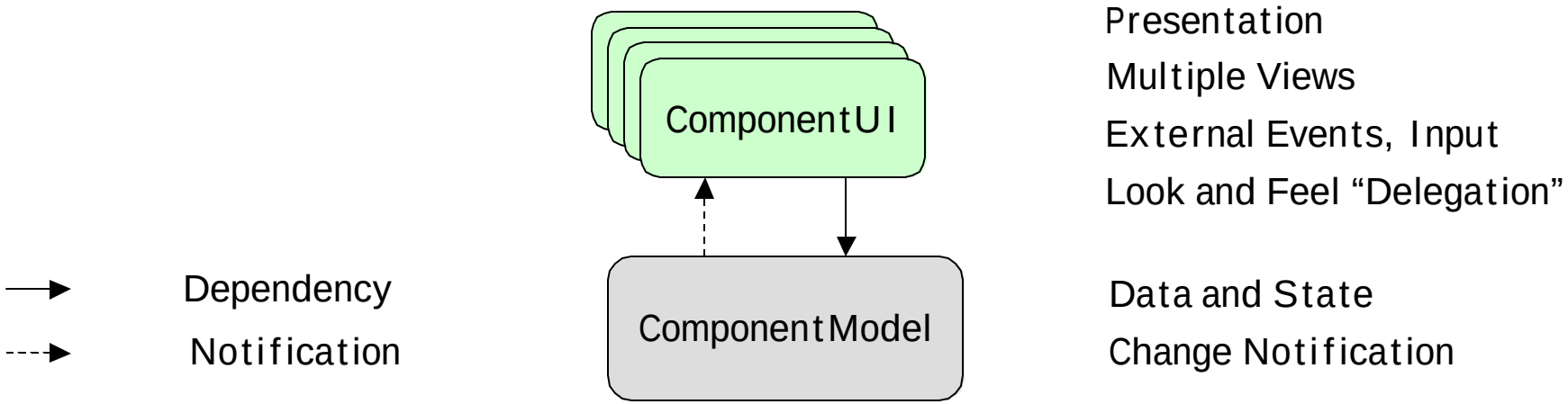


Swing: The Analogy with the MVC Architecture

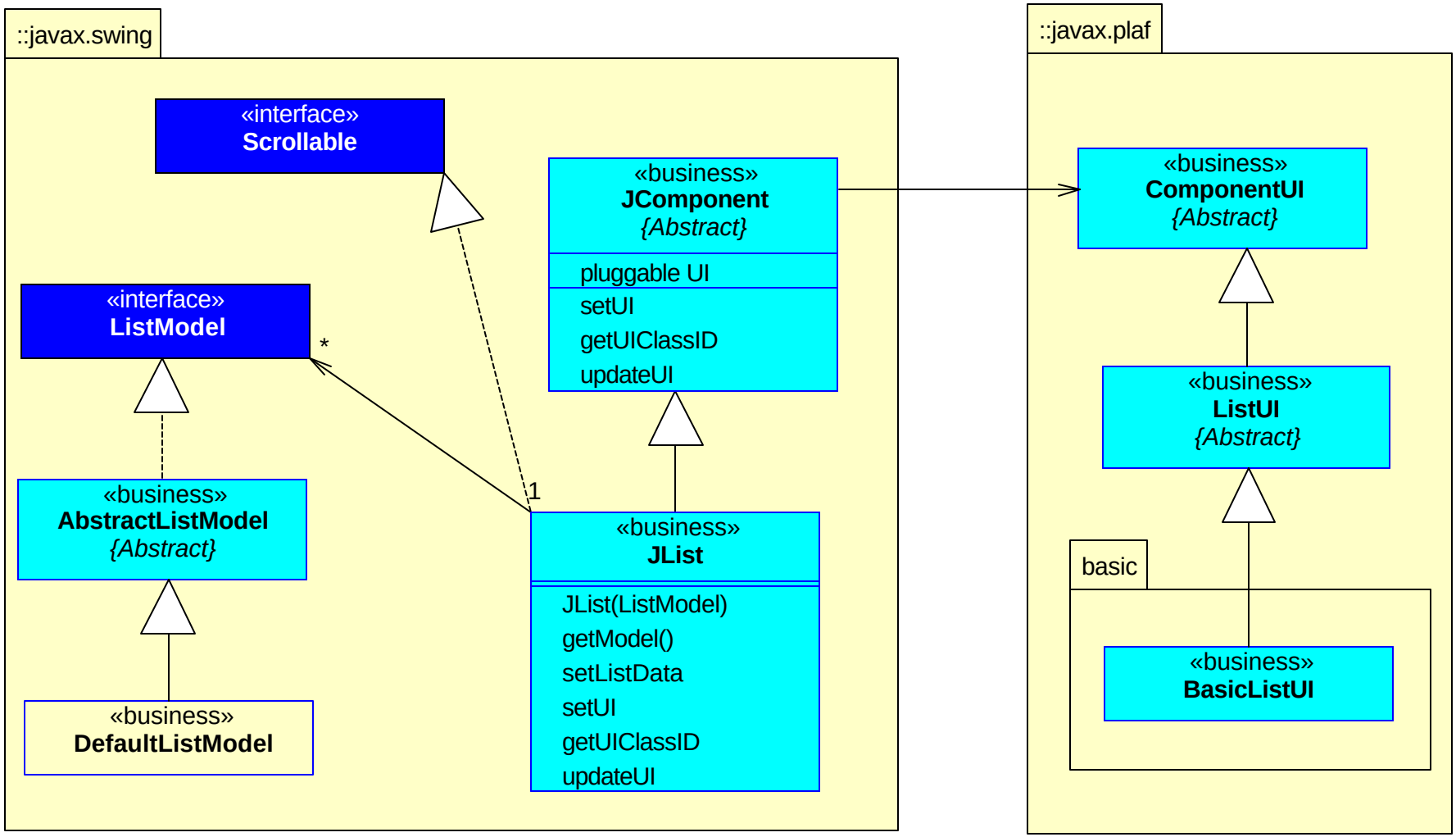
- Traditional MVC Architecture



- Swing's "Separable Model" Architecture



Swing Architecture: `JList` Class Diagram



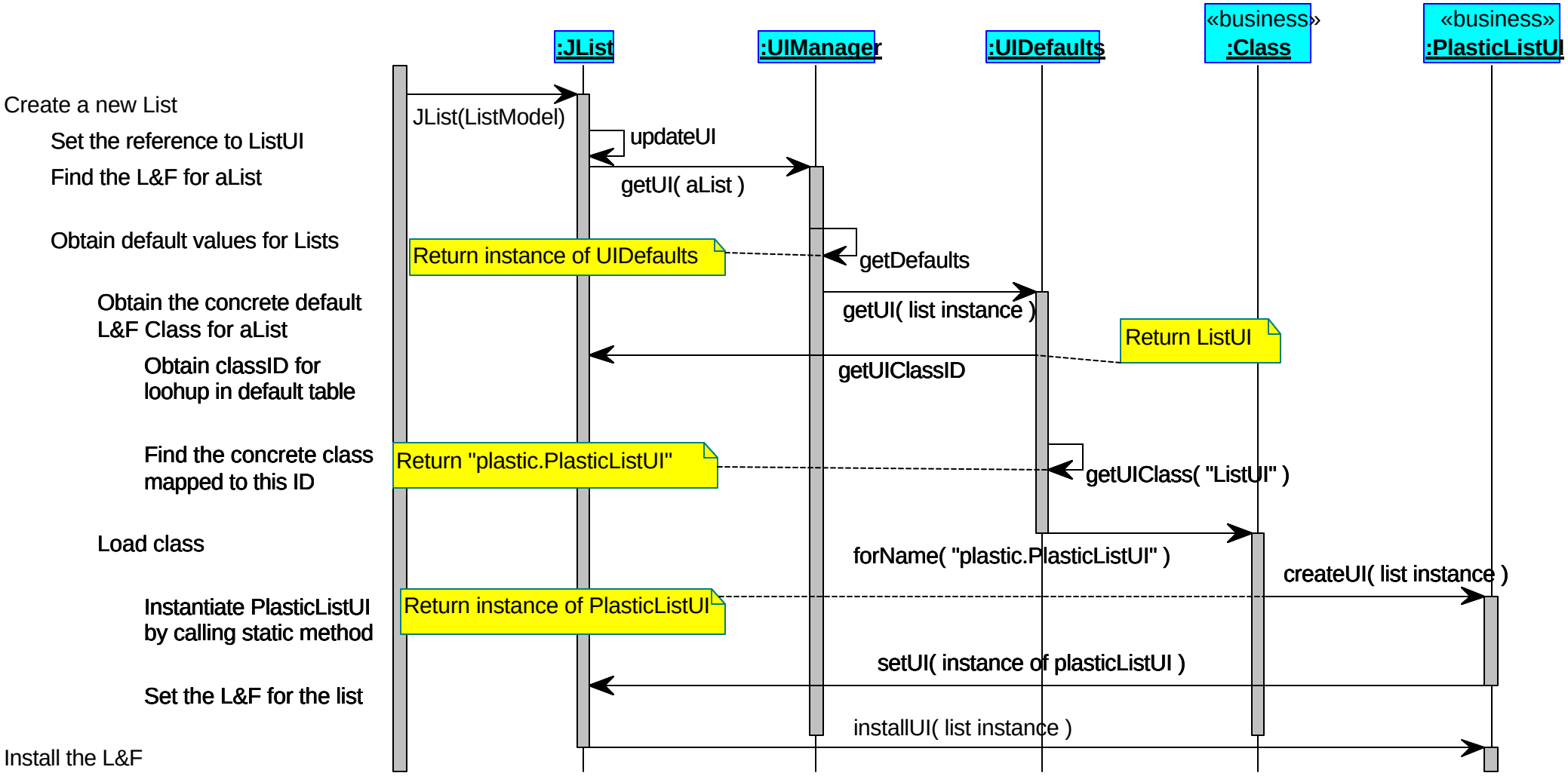
Swing: Pluggable Look&Feel

- ComponentUI abstract class
  - The single interface for delegation for all JComponent subclasses
  - Provides the fundamental functionality for pluggable Look&Feel
  - Installation, Un-installation of Look&Feel
  - Delegation of positioning and painting
- Abstract UI classes
  - Inherit from ComponentUI
  - Abstract API for all L&F subclasses
  - ListUI, ButtonUI, …
- Look&Feel implementations
  - Subclasses of the abstract UI classes in javax.swing.plaf
  - In separate packages
  - L&F implementation is not the responsibility of the application developer
  - The basic Look&Feel: javax.swing.plaf.basic
  - The Swing Look&Feel: javax.swing.plaf.metal

Swing: Managing Look&Feel

- Look&Feel installation
  - Loading and referring to a concrete UI class for a concrete subclass of JComponent
- Look&Feel Management
  - Default Look&Feel
  - Changing Look&Feel in run-time
  - Adapting to the defaults of an OS
- Class UIManager
  - Obtains and remembers the Look&Feel settings
  - Management of the defaults table for the current Look&Feel
- Class UIDefaults
  - Contains a table of defaults for Swing
- Class LookAndFeel
  - Abstract class containing all vital information for Swing
  - Concrete subclasses must provide this information

Swing: Default Instantiation



Source: JavaSoft

Swing Patterns: AbstractFactory

- Java Class: `javax.Swing.UIManager`
  - `public static void setLookAndFeel(String classID)`
  - The `classID` is the name of the class that implements the `LookAndFeel`
- Implements the `Factory` mechanism
- Implements a `classID` to a Java Class mapping
- Enables adding new classes at run-time
  - The classes must comply with the `ComponentUI` class Interface
  - They must comply with the naming standards

Swing Patterns: Mediator

- Class `javax.swing.ButtonGroup`
  - An implementation of Mediator
- A collection of `Button` objects
  - **`JRadioButton, JButton, JRadioButtonMenuItem`**
- Provides exclusivity scope
  - Makes sure that only ONE button in the group is selected
- Decouples all `Button` objects from one another

Swing Patterns: Command

- Interface **javax.swing.Action**
  - Is the common interface for all request-like objects
  - The `actionPerformed(ActionEvent e)` is inherited
- An `ActionEvent` occurs when a user
  - Clicks a button
  - Types return
  - Selects a Menu Item
- This causes the invocation of an execute method
  - Invocation of `actionPerformed(ActionEvent e)`
- `Action` centralizes the details of the request
  - `NAME`, `SHORT_DESCRIPTION`,etc.
  - State of the `Action`
- `Action` is the common interface amongst `Swing` components:
  - `ToolBar`
  - `Menu`

Swing Patterns: Command (cont.)

- The add method on each Component handles Action differently
  - Toolbar adds a JButton
  - Menu adds a JMenuItem
- Registers an ActionListener for the Action
  - Monitor changes to its state
- Handle any change to the Action locally
- The application only knows how to issue a request
  - It is unaware of the way in which widgets handle this request

Swing Patterns: Command Sample Code

```
public class MyButton extends AbstractAction implements Action{

  // some container for the properties
   public MyButton(String caption, Icon img, JFrame jframe){

       // Code for setting properties
       JFrame frame;
       frame = jframe;
   }

    public void putValue(String key, Object value) {
      properties.put(key, value);
    }

    public Object getValue(String key) {
     return properties.get(key);
   }

    public void actionPerformed(ActionEvent e) {
     System.exit(0);
   }

}
```

Swing Patterns: Command Sample Code (Cont.)

```
// Create an Action Object
Action doClose = new MyButton("Close", new ImageIcon("close.bmp"),
    this);
//Construct Menu (aMenu) and ToolBar (aToolBar)
// Add The Action Object to Menu and ToolBar
aMenu.add(doClose);
aToolBar.add(doClose)
```

Swing Patterns: Observer

- The mechanism behind the Separable Model Architecture
  - Swing version of MVC
- The **`XXXModel`** classes and interfaces are the “Observable”
  - Examples: `DefaultListModel`, `DefaultComboBoxModel`
- The subclasses of the `JComponent` are the “Observers”
- Many instances of Observers (**`JComponent`**) may share one instance of a Model Class
- Changing the State and the data in a model causes the notification of all its observers

Lesson Summary

- The Swing framework focuses at interactive java applications
  - Infrastructure Level
- Swing provides a fundamental architecture
  - Many UIComponents in javax.swing
  - MVC Architecture: Separation of Model from Presentation
  - Default Look&Feel Installation
  - Framework for managing Look&Feel at run-time
  - Adding new Look&Feel implementations at run-time
- Swing Pros:
  - Complete
  - Adaptable
  - Extensible
- Swing Cons:
  - It is not very easy to learn
  - It can be quite slow sometimes

Sinclair Community College

Thank you for attending this workshop.

Questions? Please contact us!

Jackson Reed, Inc.
Training@jacksonreed.com
www.jacksonreed.com

Educate. Collaborate. Accelerate!