

These are the slides for the The Java Design Patterns Course on Udemy. They are provided free of charge to all students.

More information about the course: <https://lpa.dev/u1javadp>

If you have any questions or queries, please add your feedback in the Q&A section of the course on Udemy.

Best regards,

Tim Buchalka
Learn Programming Academy

Welcome to Class!

Jason Fedin

A little bit about myself

- Masters of Science (Computer Science) Binghamton University
- Software Developer (16 Years)
- Online Instructor (11 years)
 - Instructed over 20 different classes, everything from Object-Oriented Programming to Compiler Theory
- Working with the Java programming language for over 20 years
 - utilizing design patterns throughout my career
- Created beginner Java and design pattern courses at multiple universities
- Software Developer at Xerox Corporation
- Mainly concentrate on Object Oriented Languages and Mobile Programming
- Focus on writing High Quality Code



Topics

- overview of Design Patterns (definition, gang of four)
- advantages of Design Patterns (why use design patterns?)
- types of Design Patterns (creational, behavioral, structural)
- important Design principles and strategies related to design patterns (Java)
 - Design Smells
 - Programming to an interface
 - Composition over Inheritance
 - Delegation Principles
 - Single Responsibility (Cohesion)
 - Open-Closed Principle
 - Liskov Substitution Principle
 - Interface Segregation Principle
 - Dependency Inversion and Dependency Injection

Topics (cont'd)

- UML
 - class diagrams
 - object-Oriented concepts in UML (inheritance, interfaces, composition, annotation)
- Creational Design Patterns
 - Factory
 - Abstract Factory
 - Singleton
 - Builder
 - Prototype

Topics (cont'd)

- Structural Design Patterns

- Adapter

- Bridge

- Composite

- Decorator

- Façade

- Flyweight

- Proxy

Topics (cont'd)

- Behavioral Design Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Topics (cont'd)

- many challenges, solutions, and examples (all Java based)
- organized around theory and many demonstrations
- hands on coding

Course Outcomes

- understand the fundamentals of design patterns
- create Java Applications that use design patterns
- master the art of problem solving in programming using efficient, proven methods
- learn how to write high quality Java code
 - modular, high cohesion and low coupling
- become proficient in basic design principles
- you will have fun!

Class Organization

Jason Fedin

Class Organization

- Lectures are designed around explaining the why and providing the how
 - A complete learning experience
 - Understand the programming language as a whole
 - Lacking in most Udemy courses
- Powerpoint slides
 - a way to present abstract concepts and definitions that I am not able to demonstrate in code (as easily)
 - Not just reading from slides!
 - Providing insight via experience
 - Providing thorough explanations

Demonstrations of code

- Demonstrations in the IDE (Integrated Development Environment)

- Code examples (concrete, real-world)
- More practical/hand-on learning

- Challenges (coding assignments/projects)

- A way for you to assess your own learning
 - Code alongs – walking through the code of a solution for a particular challenge
 - All source code provided in class

Overview

Jason Fedin

Overview

- we all talk about the way that we do things in our everyday work, hobbies, and home life and recognize repeating patterns all the time
- these “patterns” can also happen in programming
- software engineers face common problems during the development of various software programs
 - when we tell a colleague how we accomplish a tricky bit of programming so that the colleague doesn't have to recreate it from scratch
- in the past, there were no standards to instruct them how to design and proceed
 - becomes a significant problem when a new member (experienced or unexperienced) joins a team and is assigned to do a task
 - takes a lot of effort to become familiar with the existing product
- design patterns are aimed towards Object-Oriented programming languages
- designing object-oriented software is hard, and designing reusable object-oriented software is even harder
 - design patterns addresses the above issues and makes a common platform for all developers
 - they help provide a standard and give direction on how to solve common software problems

Definitions

- some useful definitions of design patterns have emerged as the literature in this field has expanded
- "Design patterns are recurring solutions to design problems you see over and over." [Smalltalk Companion]
- "Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development." [Pree, 1994]
- "Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design... and implementation." [Coplien and Schmidt, 1995]
- "A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it." [Buschmann and Meunier, et al., 1996]
- "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components." [Gamma, Helm, Johnson, and Vlissides, 1993]

Definitions (summary)

- design patterns represent the best practices used by experienced object-oriented software developers
 - solutions to general problems that software developers faced during software development
 - solutions are obtained by trial and error by numerous software developers over quite a substantial period of time
- design patterns are not just about the design of objects
 - also about interaction between objects
 - include strategies for object inheritance and containment
- design patterns can exist at many levels, from very low-level specific solutions to broadly generalized system issues

History

- the concept of design patterns originated from Christopher Wolfgang Alexander (Austria) who was an architect
 - initially applied to architecture for buildings and towns, but, not computer programming for writing software
- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”
(Alexander)
- in 1994–95, the “Gang of four” applied Alexander’s concept to software development
 - Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
 - solutions are expressed in terms of objects and interfaces instead of walls and doors

History (cont'd)

- Gang of four published “Design Patterns—Elements of Reusable Object-Oriented Software” (Addison-Wesley, 1995)
 - applied the idea of patterns to software design—calling them design patterns
 - described a structure within which to catalog and describe design patterns
 - cataloged 23 such patterns
 - postulated object-oriented strategies and approaches based on these design patterns
- Gang of Four did not create the patterns described in the book
 - they identified patterns that already existed within the software community
 - patterns that reflected what had been learned about high-quality designs for specific problems

Four Essential Elements

- a design pattern has four essential elements
- pattern name
 - a handle we can use to describe a design problem, its solutions, and consequences in a word or two
 - lets us design at a higher level of abstraction
 - makes it easier to think about designs and to communicate them and their trade-offs to others
- problem
 - explains the problem and its context
 - might describe specific design problems such as how to represent algorithms as objects
 - might describe a class or object structures that are symptomatic of an inflexible design
 - sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern

Four Essential Elements (cont'd)

- solution

- describes the elements that make up the design, their relationships, responsibilities, and collaborations
- does NOT describe a particular concrete design or implementation
 - a pattern is like a template that can be applied in many different situations
 - the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects) solves it

- consequences

- the results and trade-offs of applying the pattern
- critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern
- often concern space and time trade-offs
- may address language and implementation issues as well
- includes its impact on a system's flexibility, extensibility, or portability regarding software reuse

Advantages of Design Patterns

Jason Fedin

Why use design patterns?

- the most commonly stated reasons for studying and utilizing design patterns are because they enable us to do the following:
 - use the best solutions for certain problems faced during software development
 - make it easier to reuse successful designs and architectures (software reuse)
 - solutions have been evolved over a long period of time
 - learning these patterns helps unexperienced developers to learn software design in an easy and faster way
 - get the benefit of learning from the experience of others
 - can get a head start on my problems and avoid gotchas
 - help you to avoid reinventing the wheel
 - understand basic object-oriented principles that achieve high quality design
 - keep classes separated and prevent them from having to know too much about one another
 - encapsulation, inheritance, and polymorphism

Why use design patterns?

- improve team communications and individual learning by establishing a common platform
 - design patterns provide a common point of reference during the analysis and design phase of a project
 - allow you to describe your programming approach succinctly in terms that other programmers can easily understand
 - for example, programmers can refer to a program that uses a single object as the singleton pattern
 - more junior team members see that the senior developers who know design patterns have something of value and useful to the junior members
 - provides motivation for them to learn some of these powerful concepts
- design patterns give a higher perspective on the problem and on the process of design and object orientation analysis and design
 - frees you from dealing with the details too early
 - help a designer get a design “right” faster
- improve modifiability and maintainability of code
 - time-tested solutions have evolved into structures that can handle change more readily than what often first comes to mind as a solution
 - easier to understand code—making it easier to maintain
 - improved documentation resulting from common terminology

Problems addressed by design patterns

- design patterns solve many of the day-to-day problems object-oriented designers face, and in many different ways
- how to identify the appropriate objects to utilize/create
 - object-oriented programs are made up of objects
 - object packages both data and the procedures that operate on that data
- how to specify object interfaces
 - every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's return value
 - an object's interface characterizes the complete set of requests that can be sent to the object
- how to specify object Implementations
 - an object's implementation is defined by its class
 - the class specifies the object's internal data and representation and defines the operations the object can perform

Problems addressed by design patterns (cont'd)

- determining object granularity
 - objects can vary tremendously in size and number
 - can represent everything down to the hardware or all the way up to entire applications
- design patterns address the above issue
 - the facade pattern describes how to represent complete subsystems as objects
 - the flyweight pattern describes how to support huge numbers of objects at the finest granularities
 - abstract factory and builder yield objects whose only responsibilities are creating other objects
 - visitor and command yield objects whose only responsibilities are to implement a request on another object or group of objects

Summary

- reuse existing, high-quality solutions to commonly recurring problems
- establish common terminology to improve communications within teams
- shift the level of thinking to a higher perspective
- decide whether I have the right design, not just one that works
- improve individual learning and team learning
- improve the modifiability/quality of code
- facilitate adoption of improved design alternatives, even when patterns are not used explicitly
- discover alternatives to large inheritance hierarchies

Types of Design Patterns

Jason Fedin

Overview

- design patterns vary in their granularity, level of abstraction and how they relate to one another
- some patterns are often used together
 - composite is often used with iterator or visitor
- some patterns are alternatives
 - prototype is often an alternative to Abstract Factory
- some patterns result in similar designs even though the patterns have different intents
 - the structure diagrams of composite and decorator are similar
- there are many design patterns and so, we need a way to organize them
 - design patterns are classified into families of related patterns
 - helps you learn the patterns faster, and it can direct efforts to find new patterns as well

Overview (cont'd)

- the 23 design patterns that we will study all have several known applications and are on a middle level of generality
- they are divided into three types (gang of four)
 - organized by purpose (reflects what a pattern does)
- creational - concern the process of object creation.
- structural - deal with the composition of classes or objects
- behavioral - characterize the ways in which classes or objects interact and distribute responsibility
- design patterns can also be organized by scope (whether the pattern applies primarily to classes or to objects)
 - class patterns deal with relationships between classes and their subclasses
 - these relationships are established through inheritance
 - object patterns deal with object relationships, which can be changed at run-time and are more dynamic
 - describes how objects can be composed into larger structures using object composition or by including objects within other objects

Creational Patterns

- a program should not depend on how objects are created and arranged
- creational design patterns provide a way to create objects
- in Java, the simplest way to create an instance of an object is by using the new operator
 - fred = new Fred(); //instance of Fred class
- creational design patterns abstract the instantiation process
 - the creation logic is hidden
 - encapsulates knowledge about which concrete classes the system uses
 - programmer may call a method or use another object, rather than instantiating objects directly using the new operator
- all the system at large knows about the objects is their interfaces as defined by abstract classes
 - gives the programmer a lot of flexibility in what gets created, who creates it, how it gets created, and when
 - lets you configure a system with “product” objects that vary widely in structure and functionality
 - configuration can be static (compile-time) or dynamic (at run-time)

Creational Patterns (cont'd)

- sometimes creational patterns are competitors
 - there are cases when either Prototype or Abstract Factory could be used profitably
- sometimes creational patterns are complementary
 - builder can use one of the other patterns to implement which components get built
 - prototype can use Singleton in its implementation
- creational class patterns defer some part of object creation to subclasses
- creational object patterns defer it to another object
- there are five creational patterns that we will study
 - will highlight their similarities and differences

Structural patterns

- describes how classes and objects can be combined to form larger structures
 - utilizes inheritance to compose interfaces or implementations
 - structural object patterns describe ways to assemble objects
 - e.g. complex user interfaces and accounting data
- these design patterns concern class and object composition
- the composite design pattern
 - describes how to build a class hierarchy made up of classes for two kinds of objects
- the proxy design pattern acts as a convenient surrogate or placeholder for another object.
 - provide a level of indirection to specific properties of objects
- there are seven structural patterns that we will study
 - will highlight their similarities and differences

Behavioral Patterns

- these design patterns are specifically concerned with communication between objects
 - characterize complex control flow that is difficult to follow at run-time
 - shift the focus away from flow of control to let you concentrate just on the way objects are interconnected
- these patterns increase flexibility in carrying out this communication
- provide solutions on how to segregate objects to be both dependent and independent
- concerned with algorithms and the assignment of responsibilities between objects

Behavioral Patterns (cont'd)

- behavioral class patterns use inheritance to describe algorithms and flow of control
 - the template method is an abstract definition of an algorithm
 - defines an algorithm step by step
 - a subclass fleshes out the algorithm by defining the abstract operations
- behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone
 - uses object composition rather than inheritance
 - the mediator pattern uses a mediator object for peer object communication
 - mediator provides the indirection needed for loose coupling
- there are eleven behavioral patterns that we will study

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Design Patterns: Elements of Reusable Object-Oriented Software

By: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Selecting and Using Design Patterns

Jason Fedin

Overview (Software Design Approaches)

- designing software is normally thought of as a process of putting things together
 - a common approach is to look immediately for objects and classes and components and then think about how they should fit together
- a better approach is to use one that is based on design patterns
- start out with a conceptual understanding of the whole in order to understand what needs to be accomplished
- identify the patterns that are present in the whole
 - think about your problem in terms of the patterns that are present
 - the purpose of the pattern is to define relationships among entities
- start with those patterns that create the context for the others
 - identify these patterns

Overview (Software Design Approaches)

- work inward from the context
 - look at the remaining patterns and at any other patterns that you might have uncovered
 - pick the patterns that define the context for the patterns that would remain
 - Repeat
- finally, refine the design and implement within the context created by applying these patterns one at a time
 - as you refine, always consider the context implied by the patterns
 - the implementation incorporates the details dictated by the patterns
- designing by adding concepts within the context of previously presented concepts is attainable
 - many patterns create robust software because they define contexts within which the classes that implement them can work

Choosing a Pattern

- with more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem
- here are several different approaches to finding the design pattern that is right for your problem
- consider how design patterns solve design problems
 - determine object granularity
 - specify object interfaces
 - understanding these concepts can help guide your search for the right pattern
- understand the design patterns intent
 - provided with the pattern
 - identify if intent is relevant to your problem
 - you can use the classification scheme (creational, structural, behavioral) to narrow your search

Choosing a Pattern (cont'd)

- study how patterns interrelate
 - understanding relationships between design patterns can help direct you to the right pattern or group of patterns
- study patterns of like purpose
 - understand the similarities and differences between patterns of like purpose
- examine a cause of redesign
 - find any issues in your problem that may cause a redesign
 - look at the patterns that help you avoid the causes of redesign
- consider what should be variable in your design
 - the opposite of focusing on the causes of redesign
 - consider what you want to be able to change without redesign
 - the focus here is on encapsulating the concept that varies, a theme of many design patterns

Using and applying a pattern

- once you have picked a design pattern, how do you use it?
- read the pattern once through for an overview
 - pay particular attention to the applicability and consequences of a pattern to ensure the pattern is right for your problem
- go back and study the structure, participants, and collaborations of the pattern
 - make sure you understand the classes and objects in the pattern and how they relate to one another
- look at sample code to see a concrete example of the pattern in code
 - studying the code helps you learn how to implement the pattern
- define the classes
 - declare their interfaces
 - establish their inheritance relationships
 - define the instance variables that represent data and object references
 - identify existing classes in your application that the pattern will affect, and modify them accordingly

Using and applying a pattern (cont'd)

- choose names for pattern participants that are meaningful in the application context
 - useful to incorporate the participant name into the name that appears in the application
 - helps make the pattern more explicit in the implementation
 - e.g. if you use the Strategy pattern for a text compositing algorithm, then you might have classes SimpleLayoutStrategy or TextLayoutStrategy
- define application specific names for operations in the pattern
 - use the responsibilities and collaborations associated with each operation as a guide
 - be consistent in your naming conventions
 - e.g. you might use the “Create” prefix consistently to denote a factory method
- implement the operations to carry out the responsibilities and collaborations in the pattern
 - again, look at coding examples

How not to use a design pattern

- design patterns should not be applied indiscriminately
- often they achieve flexibility and variability by introducing additional levels of indirection
 - can complicate a design and/or cost you some performance
 - a design pattern should only be applied when the flexibility it affords is actually needed
 - the consequences of the design pattern are most helpful when evaluating a pattern's benefits and liabilities

Software Design Principles

Jason Fedin

Overview

- software design is the most important phase of the software development life cycle
 - thinking about how to structure code before you start writing it is critical
- changes and updates always happen when creating software
 - requirements always change
 - software will become legacy if no features are added or maintained on regular basis
- the cost of these changes are determined based on the structure and architecture of the system
 - the more time you put into designing a resilient and flexible architecture, the more time will save in the future when changes arise
- good software design, plans and makes it easier to add new features and handle changing requirements
- designing software is an exercise in problem solving
 - requires you to break a task down into its component parts
 - need to decide how you will address each part and how all the components will assemble together to produce the desired functionality

Design Principles

- designing software can be made easier by studying key design principles
 - principles help in creating easily maintainable and extendable software
- design patterns utilize a certain set of these key design principles (OO)
- by learning these design principles, it will help you better understand each design pattern that you study
- by learning these design principles, you will know what to do even in situations where a design pattern has not yet been discovered
 - you will already have the underlying concepts needed to solve the problem in the way that a pattern would
 - you are learning the mindset that the developers had when they solved the problems that were later identified as patterns
- the principles that we will cover in this course are based on Object-Oriented programming

Common characteristics

- the design principles that we will cover all have some common characteristics
- they all contain some form of modularization
- there is a separation of concerns for each module
 - each module knows what another module does, but it does not know how it does it
 - strive for high cohesion/low coupling
- they identify the aspects of your application that vary and separate them from what stays the same
- they favor abstraction over implementation
 - the design depends on an abstraction layer
 - makes your design open for future extensions
 - the abstraction is applied on the dynamic parts of the application (which are most likely to be changed regularly)
 - hide low-level implementation through an abstract layer
 - low-level modules have a very high possibility to be changed regularly, so they are separate from high-level modules

Common characteristics (cont'd)

- they do not contain duplication
 - instead use abstraction to abstract common things in one place
 - if you have a block of code in more than two places consider making it a separate method
 - when using a hard-coded value more than one time, make it a public final constant
- the design for classes/methods/modules have a single focus and single responsibility
 - minimizes regressions
- encapsulation is a big deal
 - encapsulate the code you expect or suspect to be changed in future
 - it is easier to test and maintain proper encapsulated code
- these principles will suggest making variable and methods private by default
 - increasing access step by step (from private to protected and not public)
- remember, because design patterns utilize these principles, design patterns also have these common characteristics
 - factory design pattern is one example of utilizing encapsulation
 - encapsulates object creation code and provides flexibility to introduce a new product later with no impact on existing code

Common strategies that design patterns use

- patterns are designed to use interfaces
- patterns favor aggregation over inheritance
 - aggregation means a collection of things that are not part of it
 - Independent
 - e.g. airplanes at an airport
 - composition means something is a part of another thing
 - cannot exist independent of the parent
 - e.g. wheels on an airplane
- the distinction loses much of its importance in languages that have garbage collection, because you do not have to concern yourself with the life of the object
 - when the object goes away, should its parts also go away?
- patterns use encapsulation for variation
 - find what varies and encapsulate it
- patterns favor alternatives to large inheritance hierarchies
 - design patterns enable you or your team to create designs for complex problems that do not require large inheritance hierarchies
 - avoiding large inheritance hierarchies will result in improved designs

Design Principles that we will cover in detail

- Programming to an interface
- Favor Composition over Inheritance
- Delegation principles
- Single Responsibility
- Open Closed Design
- Liskov Substitution
- Interface Segregation
- Dependency Injection (Inversion)

Design Smells

Jason Fedin

Overview

- structures in the design that indicate violation of fundamental design principles and negatively impact design quality
- General Design Smells
 - Rigidity
 - the tendency of software to be difficult to change
 - a single change causes a cascade of subsequent changes in dependent modules (tight coupling)
 - the more modules that must be changed, the more rigid the design
 - Fragility
 - the tendency of software to break in many places when a single change is made
 - the problems often occur in places that have no obvious relation to the area that was changed
 - as the fragility of a module increases, the likelihood that a change will introduce unexpected problems rise
 - Immobility
 - contains parts that could be useful in other systems, but the effort and risk of separating them from the original system are too great

General design Smells (cont'd)

- Viscosity
 - a system has a high viscosity if a design-preserving change is more difficult to use than a hack
 - If running unit tests and compilation takes a lot of time, it is likely for a developer to bypass procedures and implement a hack without running all automated tests
- Needless repetition
 - copy and pasting code throughout the system
 - happens when necessary abstractions have not been made
- Opacity
 - the tendency of a module to be difficult to understand
 - when code is not written in a clear and expressive manner, it is said to be opaque
 - code that evolves over time tends to become more difficult to understand over time
- Needless complexity
 - one of the most important smells of bad design
 - in a passionate attempt to avoid the other smells, developers introduce all sorts of abstractions and preparations for potential changes in the future
 - good software design is lightweight, flexible, easy to read and understand and above all easy to change
 - there is no need to keep into account all potential changes in the future
 - do not “over-design”

smells in java code

- several classes that duplicate 90% of each other
 - use inheritance to remove duplication
- too many public classes, members, and methods
 - violates encapsulation
- classes that are too large
 - combining every single concept in your application into one massive class
- design smells often end up creating code that is
 - hard to read
 - breaks easily
 - hard to maintain (adding new features or doing basic bug fixing)

Summary

- bottom line is to follow good design principles as guidelines and be aware of design smells
- always strive for highly cohesive and loosely coupled solutions, code and design
- don't try to create the perfect design or architecture
 - it does not exist and trying to achieve it will most likely lead to needless complexity
- So....., this all leads us to the conclusion that
 - DESIGN PATTERNS ARE AWESOME!
 - since they solve the above problems and ensure a high quality design in your application

Programming to an Interface

Jason Fedin

Overview

- the word interface is overloaded
 - there is the concept of interface, but there is also the Java construct interface
 - you can program to an interface, without having to actually use a Java interface
- “Program to an interface” really means “Program to a supertype”
 - the declared type of the variables should be a supertype, usually an abstract class or interface
 - the objects assigned to those variables can be of any concrete implementation of the supertype
 - the class declaring them doesn’t have to know about the actual object types
- do not declare variables to be instances of a particular concrete class
 - instead, commit only to an interface defined by an abstract class (interface or abstract)
- always program for the interface and not for the implementation
 - will lead to flexible code which can work with any new implementation of the interface
- programming to an interface is a common theme of the design patterns

Overview (cont'd)

- manipulating objects solely in terms of the interface is beneficial to clients
 - clients do not need to know the specific types of objects they use
 - as long as the objects adhere to the interface that clients expect
 - clients do not need to know the classes that implement these objects
 - they only know about the abstract class(es) defining the interface
- “programming to an interface” greatly reduces implementation dependencies between subsystems
- we can use interface types on variables, return types of methods or parameter types in a method
- the point is to exploit polymorphism by programming to a supertype so that the actual runtime object is not locked into the code

Polymorphism

- I know this is probably redundant, but just to make sure we are on the same page, imagine an abstract class Animal, with two concrete implementations, Dog and Cat
- programming to an implementation would be:

```
Dog d = new Dog();
d.bark();
```

- declaring the variable “d” as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation

Polymorphism (cont'd)

- programming to an interface/supertype would be:

```
Animal animal = new Dog();
animal.makeSound();
```

- we know it is a Dog, but we can now use the animal reference polymorphically
- even better, rather than hardcoding the instantiation of the subtype (like new Dog()) into the code, assign the concrete implementation object at runtime:

```
a = getAnimal();
a.makeSound();
```

- we do not know WHAT the actual animal subtype is
 - all we care about is that it knows how to respond to makeSound()

Abstract Classes vs. Interfaces

- with support of default methods in interfaces since the launch of Java 8, the gap between when to use an interface and when to use an abstract classes has been reduced
- variables in interfaces are public static final
 - abstract classes can have other access modifiers for variables (private, protected, etc.)
- methods in interfaces are public or public static
 - methods in abstract classes can be private and protected too
- utilize abstract classes to establish a relationship between interrelated objects
 - when you want to share code among several closely related classes then this common state or behavior can be put in the abstract class

Abstract Classes vs. Interfaces (cont'd)

- utilize interfaces to establish a relationship between unrelated classes
 - the interfaces Comparable and Cloneable are implemented by many unrelated classes
- utilize interfaces if you want to specify the behavior of a particular data type, but are not concerned about who implements its behavior
- utilize interfaces if you want to take advantage of multiple inheritance
- **one is not better than the other**
- you might create an interface and then have an abstract class implement that interface

Demonstrate example in intellij

- a computer monitor is designed for display purposes
 - the computer is a product
 - the computer monitor is a part or module of the computer which is responsible for the display operation
- later on, the client needs change and now they want to display on a projector
- the display module of a computer should be flexible
 - we need to change it easily, or we can change it dynamically (at runtime)

Demonstrate example in intellij

```
interface displayModule
{
    public void display();
}

public class Monitor implements displayModule
{
    public void display()
    {
        System.out.println("Display through Monitor");
    }
}
```

```
public class Projector implements displayModule
{
    public void display()
    {
        System.out.println("Display through projector");
    }
}
```

Demonstrate example in intellij

```
public class Computer
{
    // programming through interface as variable part
    displayModule dm;

    public void setDisplayModule(displayModule dm)
    {
        this.dm=dm;
    }

    public void display()
    {
        dm.display();
    }
}
```

```
public static void main(String args[])
{
    Computer cm =new Computer();
    displayModule dm = new Monitor();
    displayModule dm1 = new Projector();
    cm. setDisplayModule(dm);
    cm. display();
    cm. setDisplayModule(dm1);
    cm. display();
}
```

Output:
Display through Monitor
Display through projector.

Demonstrate example in intellij

- the client may change the display operation style
- we create an interface called displayModule
 - all display equipment must implement that interface and provide its own implementation of the display operation
- remember to always code through an interface so you can change your strategy at runtime with actual implementation

Composition over Inheritance

Jason Fedin

Composition

- composition is referred to as a HAS-A relationship between classes in Object Oriented design
 - an object contains (owns) another object as a member variable of its class
- composition implies a relationship where the child cannot exist independent of the parent
 - something is a part of another thing (wheels on an airplane)
 - rooms in a house - each house has a room or many rooms, rooms do not exist separate to a house
 - cells in a body - when the body object is destroyed, the cells get destroyed with it
- when you put two classes together like this you are using composition
 - instead of inheriting their behavior, the houses get their behavior by being composed with the right behavior object
- composition is used in many design patterns

Aggregation

- aggregation is a HAS-A relationship between objects and is closely related to composition
- aggregation implies a relationship where the child can exist independently of the parent
 - a collection of things that are not part of it
 - airplanes at an airport
 - students in a class - get rid of the class and the students still exist
 - tires on a car - the tires can be taken off of the car and installed on a different one
- aggregation and composition are almost completely identical except that composition is used when the life of the child is completely controlled by the parent
 - the distinction loses much of its importance in languages that have garbage collection
 - you do not have to concern yourself with the life of the object

Composition over Inheritance

- favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task
 - your classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters
- inheritance breaks encapsulation because sub classes are dependent upon the base class behavior
 - inheritance is tightly coupled whereas composition is loosely coupled
 - when behavior of super class changes, functionality in sub class may get broken, without any change on its part
- java does not support multiple inheritance
 - composition can be a “work-around” to this
- most design patterns favor Composition over Inheritance
 - Strategy
 - Decorator
 - If design patterns use composition then that means it has been tried and tested
- composition offers better test-ability of a class than when using Inheritance
 - you can easily provide a mock implementation of the classes that you are using
 - when designing using Inheritance it is harder to test because you need to mock both the base and subclasses
 - unit testing is one of the most important things to consider during software development
 - test driven development

What about Software Reuse?

- both composition and inheritance promote code reuse through different approaches
- designers tend to over use inheritance as a reuse technique
- designs are often made more reusable (and simpler) by depending more on object composition
- do not use Inheritance just for the sake of code reuse
 - composition is a more flexible and extensible mechanism when reusing existing code
- you should be able to get all the functionality you need just by assembling existing components through object composition
- composition allows for code reuse from final classes
 - impossible with Inheritance because you cannot extend a final class in Java

Software Reuse

- here is an excerpt from “Head First Design Patterns, Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra” concerning inheritance and reuse

Master: Grasshopper, tell me what you have learned of the Object-Oriented ways.

Student: Master, I have learned that the promise of the object-oriented way is reuse.

Master: Grasshopper, continue...

Student: Master, through inheritance all good things may be reused and so we come to drastically cut development time like we swiftly cut bamboo in the woods.

Master: Grasshopper, is more time spent on code before or after development is complete?

Student: The answer is after, Master. We always spend more time maintaining and changing software than on initial development.

Master: So Grasshopper, should effort go into reuse above maintainability and extensibility?

Student: Master, I believe that there is truth in this.

Master: I can see that you still have much to learn. I would like for you to go and meditate on inheritance further. As you've seen, inheritance has its problems, and there are other ways of achieving reuse.

Summary

- creating systems using composition gives you a lot of flexibility
 - lets you encapsulate a family of algorithms into their own set of classes
 - lets you change behavior at runtime as long as the object you are composing with implements the correct behavior interface
- a design based on object composition will have more objects (if fewer classes)
 - the system's behavior will depend on their interrelationships instead of being defined in one class
- there are many more reasons to favor Composition over inheritance, which you will start discovering once you start using design patterns

Delegation Principles

Jason Fedin

Overview

- delegation is the concept of one class “delegating” its behavior to another class
 - don't do all stuff by yourself, delegate it to a respective class
 - when you delegate, you are simply calling up some class which knows what must be done
 - you do not really care how it does it, all you care about is that the class you are calling knows what it needs to do
- delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate
- delegation is an extreme example of object composition
 - shows that you can always replace inheritance with object composition as a mechanism for code reuse
 - delegation means that you use an object of another class as an instance variable, and forward messages to the instance
- it is better than inheritance for many cases
 - it makes you to think about each message you forward
 - the instance is of a known class, rather than a new class
 - it does not force you to accept all the methods of the super class
 - you can provide only the methods that really make sense

Advantages

- the primary advantage of delegation is run-time flexibility
 - makes it easy to compose behaviors at run-time and to change the way they are composed
- delegation is a good design choice only when it simplifies more than it complicates
 - how effective it will be depends on the context and on how much experience you have with it
 - delegation works best when it is used in design patterns
- several design patterns use delegation
 - **State** - an object delegates requests to a State object that represents its current state
 - **Strategy** - an object delegates a specific request to an object that represents a strategy for carrying out the request
 - **Visitor** - the operation that gets performed on each element of an object structure is always delegated to the Visitor object

Examples

- assume your class is called B and the delegated class is called A
 - use delegation if you want to enhance A, but A is final
- the equals() and hashCode() method in Java is a classic example of delegation
 - in order to compare two objects for equality, we ask the class itself to do comparison instead of client class doing that check
- event delegation is another example of delegation
 - an event is delegated to handlers for handling

Demonstrate this example in intelliJ

```
class RealPrinter {  
    // the "delegate"  
    void print()  
    {  
        System.out.println("The Delegate");  
    }  
}  
  
class Printer {  
    // the "delegator"  
    RealPrinter p = new RealPrinter();  
  
    // create the delegate  
    void print()  
    {  
        p.print(); // delegation  
    }  
}
```

```
public class Tester {  
    // To the outside world it looks like Printer actually prints.  
    public static void main(String[] args)  
    {  
        Printer printer = new Printer();  
        printer.print();  
    }  
}
```

Single Responsibility

Jason Fedin

Overview

- states that every class should have responsibility over a single part of the functionality provided by the software
 - the responsibility should be entirely encapsulated by the class
 - all its methods should be narrowly aligned with that responsibility
 - a class should have only one job
- “a class should have a single responsibility, where a responsibility is nothing but a reason to change”
- should make sure that one class at the most is responsible for doing one task or functionality among the whole set of responsibilities that it has
 - only when there is a change needed in that specific task or functionality should this class be changed
- the Single Responsibility Principle is closely related to the concepts of coupling and cohesion
- coupling is the degree of interdependence between software classes or methods
 - a measure of how closely connected two classes or two methods are
 - the strength of the relationships between classes

Overview (cont'd)

- low coupling means small dependencies between classes/methods
 - easier to change code without introducing bugs in other classes or other methods
- tight coupling means two classes/methods are closely connected
 - a change in one module may affect another module
- cohesion refers to what the class (or method) can do
 - low cohesion would mean that the class does a great variety of actions
 - it is broad, unfocused on what it should do
 - high cohesion means that the class is focused on what it should be doing
 - contains only methods relating to the intention of the class
- the single responsibility principle is about limiting the impact of change by designing loosely (low) coupled classes that are highly cohesive

Examples of responsibilities

- some examples of responsibilities to consider that may need to be separated include:
 - Persistence
 - Validation
 - Notification
 - Error Handling
 - Logging
 - Class Selection / Instantiation
 - Formatting
 - Parsing
 - Mapping

Example

```
public class Employee{  
    private String employeeId;  
    private String name;  
    private String address;  
    private Date dateOfJoining;  
  
    public boolean isPromotionDueThisYear(){  
        //promotion logic implementation  
    }  
  
    public Double calcIncomeTaxForCurrentYear(){  
        //income tax logic implementation  
    }  
  
    //Getters & Setters for all the private attributes  
}
```

- **the Employee class looks logically correct**
 - **has all the employee attributes like employeeId, name, age, address & dateOfJoining**
 - **tells you if the employee is eligible for promotion this year**
 - **calculates the income tax he has to pay for the year**
- **however, the Employee class breaks the Single Responsibility Principle**

Example (cont'd)

- the logic of determining whether the employee is due a promotion this year is not a responsibility which the employee owns
- the company's HR department owns this responsibility based on the company's HR policies which may change every few years
- on any such change in HR policies, the Employee class will need to be updated as it is currently has the responsibility of promotion determination
- similarly, income tax calculation is not a responsibility of the Employee
 - it is the finance department's responsibility which takes care of the current tax structure which may get updated every year
- if Employee class owns the income tax calculation responsibility then whenever tax structure/calculations change Employee class will need to be changed
- the Employee class should have the single responsibility of maintaining core attributes of an employee

Example

- so, lets refactor the Employee class so that it adheres to the Single Responsibility Principle
- first, we need to move the promotion determination logic from Employee class to the HRPromotions

```
public class HRPromotions{  
    public boolean isPromotionDueThisYear(Employee emp){  
        //promotion logic implementation using the employee information passed  
    }  
}
```

Example (cont'd)

- second, lets move the income tax calculation logic from Employee class to FinITCalculations class

```
public class FinITCalculations{  
    public Double calcIncomeTaxForCurrentYear(Employee emp){  
        //income tax logic implementation using the employee information passed  
    }  
}
```

Example (cont'd)

- now the Employee class adheres to a single responsibility of maintaining core employee attributes

```
public class Employee{  
    private String employeeId;  
    private String name;  
    private String address;  
    private Date dateOfJoining;
```

//Getters & Setters for all the private attributes

}

Open Closed Design Principle

Jason Fedin

Overview

- Bertrand Meyer proposed the open-closed principle (OCP)
 - classes and methods should be Open for extension (new functionality) and Closed for modification
 - a class should be easily extendable without modifying the class itself
- a module is said to be open if it is still available for extension
 - it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs
- a module is said to be closed if it is available for use by other modules
 - assumes that the module has been given a well-defined, stable description
 - the interface in the sense of information hiding (not a java interface)
- general idea of this principle is that it tells you to write your code so that you will be able to add new functionality without changing the existing code
 - prevents situations in which a change to one of your classes also requires you to adapt all depending classes
 - reduces tight coupling
- Robert C. Martin considered this principle as the “the most important principle of object-oriented design”

Overview (cont'd)

- unfortunately, Bertrand Meyer proposed the use of inheritance to achieve the open/closed principle
- however, inheritance introduces tight coupling if the subclasses depend on implementation details of their parent class
- others redefined the Open/Closed Principle to the Polymorphic Open/Closed Principle
 - uses interfaces instead of super classes to allow different implementations
 - interfaces can be reused through inheritance but implementation need not be
 - can easily substitute without changing the code that uses them
 - multiple implementations can be created and polymorphically substituted for each other
- interfaces are closed for modifications
 - you can provide new implementations to extend the functionality of your software
 - new implementations must implement the interface
- interfaces introduce an additional level of abstraction which enables loose coupling
 - interfaces are independent of each other and don't need to share any code (usually)

Example (demonstration in intellij)

- we need to calculate areas of various shapes
- start with creating a class for our first shape
 - rectangle which has 2 attributes length & width

```
public class Rectangle{  
    public double length;  
    public double width;  
}
```

Example (cont'd)

- next we create a class to calculate area of this Rectangle
 - has a method calculateRectangleArea()
 - takes the Rectangle as an input parameter and calculates its area

```
public class AreaCalculator{  
    public double calculateRectangleArea(Rectangle rectangle){  
        return rectangle.length *rectangle.width;  
    }  
}
```

Example (cont'd)

- now lets say we want to create a circle shape
 - we create a new class Circle with a single attribute radius

```
public class Circle{  
    public double radius;  
}
```

Example (cont'd)

- we then modify the AreaCalculator class to add circle calculations
 - calculateCircleArea()

```
public class AreaCalculator{  
    public double calculateRectangleArea(Rectangle rectangle){  
        return rectangle.length *rectangle.width;  
    }  
  
    public double calculateCircleArea(Circle circle){  
        return (22/7)*circle.radius*circle.radius;  
    }  
}
```

Example (cont'd)

- you should notice that there are some issues with the way we designed our software
- what if we want to add another shape?
 - we will end up modifying the AreaCalculator class
 - as we keep adding new shapes, the AreaCalculator class keeps on changing
 - all consumers of this class will have to keep on updating their code to contain AreaCalculator
- the AreaCalculator class will not be finalized
 - every time a new shape comes it will be modified
 - design is not closed for modification
- this design is also not extensible
 - AreaCalculator will need to keep on adding new logic (another method) for each new shape
 - we are not expanding the scope of shapes
 - we are adding a new solution for every shape that is added
- we need to modify the current design so that it complies with the Open/Closed Principle

Example (cont'd)

- we need to define a base type (interface) Shape and have Circle & Rectangle implement a Shape

```
public interface Shape{  
    public double calculateArea();  
}
```

```
public class Rectangle implements Shape{  
    double length;  
    double width;  
    public double calculateArea(){  
        return length * width;  
    }  
}
```

```
public class Circle implements Shape{  
    public double radius;  
    public double calculateArea(){  
        return (22/7)*radius*radius;  
    }  
}
```

- the above design allows the code to be much more extensible

Example (cont'd)

- the reasons this design is more extensible:
 - there is a base interface Shape
 - all shapes now implement the base interface Shape
 - shape interface has an abstract method calculateArea()
 - both circle & rectangle provide their own overridden implementations of calculateArea() method using their own attributes
 - shapes are now an instance of Shape interfaces
 - allows us to use Shape instead of individual classes wherever these classes are used by any consumer

Example (cont'd)

```
public class AreaCalculator{  
    public double calculateShapeArea(Shape shape){  
        return shape.calculateArea();  
    }  
}
```

- this AreaCalculator class now fully removes our previous design flaws
 - offers a clean solution which adheres to the Open-Closed Principle

Example (cont'd)

- the resulting design is open for extension
 - more shapes can be added without modifying the existing code
 - just need to create a new class for the new shape and implement the calculateArea() method with a formula specific to that new shape
- the resulting design is closed for modification
 - AreaCalculator class is complete (for calculating areas of shapes)
 - will work for all shapes (any classes that implement that interface)

Liskov Substitution

Jason Fedin

Overview

- the Liskov Substitution principle was introduced by Barbara Liskov
- the principle defines that objects of a superclass can be replaceable with objects of its subclasses without breaking the application
 - requires the objects of your subclasses to behave in the same way as the objects of your superclass
 - methods which use a superclass type must be able to work with the subclass without any issues
- an overridden method of a subclass needs to accept the same input parameter values as the method of the superclass
 - do not implement any stricter validation rules on input parameters than implemented by the parent class
 - any code that calls this method on an object of the superclass might cause an exception, if it gets called with an object of the subclass
- the return value of a method of the subclass needs to comply with the same rules as the return value of the method of the superclass
 - you can only decide to apply stricter rules by returning a specific subclass of the defined return value or by returning a subset of the valid return values of the superclass
- in order to follow LSP the subclass must enhance functionality, but not reduce functionality

Overview (cont'd)

- this principle is in line with what Java also allows
 - a superclass reference can hold a subclass object
 - superclass can be replaced by subclass in a superclass reference at any time
 - the Java inheritance mechanism follows the Liskov Substitution Principle
- LSP is closely related to the Single responsibility principle and Interface Segregation Principle
 - If a base class has more functionality than a subclass might not support some of the functionality and does violate LSP
- this principle extends the Open/Closed principle
 - the Open/closed principle says that a class should be open for extension and closed for modification
 - we override the original class and implement the functionality to be changed in the overriding class
 - when the subclass object is used in place of the super-class the overridden functionality is executed
 - this is exactly in line with the Liskov Substitution Principle

Example

- Suppose we have a class Vehicle and it has two sub classes: Car and Bus

```
public class Vehicle {  
    int getSpeed();  
    int getCubicCapacity();  
}
```

```
public class Car extends Vehicle {  
    int getSpeed(){ }  
    int getCubicCapacity() { }  
    boolean isHatchBack() {}  
}
```

```
public class Bus extends Vehicle {  
    int getSpeed() { }  
    int getCubicCapacity() { }  
    String getEmergencyExitLoc() { }  
}
```

```
// following the Liskov principle  
Vehicle vehicle = new Bus();  
Vehicle.getSpeed();  
Vehicle = new Car()  
Vehicle.getCubicCapacity
```

Example (cont'd)

- we can assign an object of type Car or that of type Bus to a reference of type Vehicle
- all the functionality in base class Vehicle can be invoked on a reference of type Vehicle
 - it is possible to invoke methods `getSpeed()` & `getCubicCapacity()` on a Vehicle reference which actually holds a Bus/Car object
 - the actual object's overridden implementation of these methods will be invoked
- this is exactly what the Liskov Substitution Principle states
 - subtype objects can replace super type objects without affecting the functionality inherent in the super type

Another Example (Circle-Ellipse)

- all circles are inherently ellipses with their major and minor axes being equal
- in terms of classes, Ellipse could be a base class and Circle a subclass
 - an object of type Circle can be assigned to a reference of type Ellipse
 - all the methods in Ellipse can be invoked on a Circle object
- one inherent functionality of an ellipse is that its stretchable
 - the length of the two axes of an ellipse can be changed
- so, we will add methods setLengthOfAxisX() and setLengthOfAxisY() to the Ellipse class
- calling any of these two methods on an object of type circle inside a reference of type ellipse would lead to a circle no longer being a circle
 - for a circle the length of the major and minor axes are equal
- this is a violation of the Liskov Substitution Principle
 - assigning a subtype object to a super type reference does not work

Interface Segregation

Jason Fedin

Overview

- the Interface Segregation Principle was defined by Robert C. Martin while consulting for Xerox to help them build the software for their new printer systems
- “Clients should not be forced to depend upon interfaces that they do not use.”
 - a client should not implement an interface if it does not use a method in that interface
 - happens mostly when one interface contains more than one functionality, and the client only needs one functionality and not the other
- the goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts
- interface design is a tricky job because once you release your interface you can not change it without breaking all implementation
- using the interface keyword in Java means that you have to implement all of the methods in the interface before any class can use it
 - if you follow this principle in Java, you will implement less methods because each interface will have a single functionality

Overview

- suppose there is a system which has multiple functionalities and various clients using those functionalities
 - we could create an API and implement it by using the Java “interface” keyword
- If we created a single interface then all clients will have to unnecessarily implement all other clients' methods just to make their interface compile
 - this is referred to as a “fat” interface
 - an object-oriented designer's nightmare
 - makes the overall design rigid due to the enormous effort required to manage changes across all clients when making a change to a method pertaining to only one client
- the Interface Segregation Principle avoids the design drawbacks associated with a fat interface by refactoring each fat interface into multiple segregated interfaces
 - each segregated interface is a lean interface as it only contains methods which are required for a specific client
- a lean interface does not mean one method per interface
- a lean interface caters to a consumers of a specific type of functionality or a specific set of customers all of whom have the same functional needs

Example (Shapes)

- suppose we have an interface that represents different types of shapes, defined as follows

```
interface ShapeInterface {  
    public function area();  
    public function volume();  
}
```

- all shapes must implement the volume method
 - however, squares are flat shapes and they do not have volumes
 - this interface would force the Square class to implement a method that it has no use of
- this interface violates the interface segregation principle
 - Instead, you should create another interface called SolidShapeInterface that has the volume contract
 - solid shapes like cubes can implement this interface

Example (Shapes) (cont'd)

```
interface ShapeInterface {  
    public double area();  
}
```

```
interface SolidShapeInterface {  
    public double volume();  
}
```

```
class Cube implements ShapeInterface, SolidShapeInterface {  
    public double area() {  
        // calculate the surface area of the cuboid  
    }  
  
    public double volume() {  
        // calculate the volume of the cuboid  
    }  
}
```

- this is a much better approach, but, still has an issue
 - you are now coupled to the type of shape
- an even better way is to create a “ManageShapeInterface”
 - implement it on both the flat and solid shapes
 - you can easily see that it has a single API for managing the shapes

Example (Shapes) (cont'd)

```
interface ManageShapeInterface {  
    public double calculate();  
}  
  
class Square implements ShapeInterface,  
ManageShapeInterface {  
    public function area() {  
        /_Do stuff here_  
    }  
  
    public double calculate() {  
        return this.area();  
    }  
}  
  
class Cube implements ShapeInterface,  
SolidShapeInterface, ManageShapeInterface {  
    public double area() {  
        /_Do stuff here_  
    }  
  
    public double volume() {  
        /_Do stuff here_  
    }  
  
    public double calculate() {  
        return this.area() + this.volume();  
    }  
}
```

Another Example (Restaurant)

- suppose there is a Restaurant interface which contains methods for
 - accepting orders from online customers
 - dial-in or telephone customers
 - walk-in customers
 - handling online payments
 - in-person payments

```
public interface RestaurantInterface {  
    public void acceptOnlineOrder();  
    public void takeTelephoneOrder();  
    public void payOnline();  
    public void walkInCustomerOrder();  
    public void payInPerson();  
}
```

Another Example (Restaurant)

```
public class OnlineClientImpl implements RestaurantInterface
{
    @Override
    public void acceptOnlineOrder() {
        //logic for placing online order
    }
    @Override
    public void takeTelephoneOrder() {
        //Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }
    @Override
    public void payOnline() {
        //logic for paying online
    }
    @Override
    public void walkInCustomerOrder() {
        //Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }
    @Override
    public void payInPerson() {
        //Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }
}
```

Another Example (Restaurant)

- problems with the current Restaurant interface design
 - the OnlineClientImpl class is for online orders and thus, the following methods are unnecessary
 - telephone order placement
 - walk-in order placement
 - in-person payment
- online, telephonic and walk-in clients use the Restaurant Interface implementation specific to each of them
 - similar to the above OnlineClientImpl, the implementation classes for Telephonic client and Walk-in client will have methods which will be unsupported
- since the 5 methods are part of the Restaurant Interface, the implementation classes have to implement all 5 of them
 - the methods that each of the implementation classes do not implement throw UnsupportedOperationException
 - implementing all methods is inefficient
- any change in any of the methods of the Restaurant Interface will be propagated to all implementation classes
 - maintenance of the code is a nightmare
- the Restaurant Interface breaks the Single Responsibility Principle because the logic for payments as well as that for order placement is grouped together in a single interface

Another Example (Restaurant)

- we need to apply the Interface Segregation principle to correct the problems with the current Restaurant interface design
- separate out payment and order placement functionalities into two separate lean interfaces
 - PaymentInterface and OrderInterface
- each of the clients use one implementation each of PaymentInterface and OrderInterface
 - OnlineClient uses OnlinePaymentImpl and OnlineOrderImpl and so on
- the Single Responsibility Principle is now adhered to as the Payment interface and Ordering interfaces are separate and have their client-specific implementations
- change in any one of the order or payment interfaces does not affect the other
 - they are independent now
- no need to do any implementation or throw an UnsupportedOperationException as each interface has only methods it will always use

Dependency Inversion

Jason Fedin

Overview

- dependency inversion is a principle that states that entities must depend on abstractions and not on concretions
 - the goal is to reduce dependencies on concrete classes
- abstractions should not depend upon details
 - details should depend upon abstractions
- high level classes must not depend on the low level classes
 - both high-level classes and low-level classes should depend upon abstractions
 - the lower-level class implementation is accessible to the higher-level class via an abstract interface
 - actual implementation of lower level class can then vary
- the “inversion” in the name “Dependency Inversion Principle” is there because it inverts the way you typically might think about your OO design
 - top-to-bottom dependency has inverted itself, with both high-level and low-level classes now depending on an abstraction
- sounds a lot like “Program to an interface, not an implementation”
 - similar, however, the Dependency Injection Principle makes an even stronger statement about abstraction
- dependency inversion is a central principle underlying the use of design patterns

Invert your thinking...

- lets say we need to implement a pizza store
 - What's the first thought that pops into your head?
- start at the top and follow things down to the concrete classes
 - however, you do not want your store to know about the concrete pizza types
 - pizza store will then be dependent on all those concrete classes
- let's "invert" your thinking...
 - instead of starting at the top, start at the Pizzas and think about what you can abstract
 - pizza is the abstraction
- your different concrete pizza types depend only on an abstraction and so does your store
 - the initial design where the store depended on concrete classes can be inverted to have the design abstract those dependencies

PizzaStore (Example)

- a PizzaStore could be a high-level class
 - its behavior is defined in terms of pizzas
 - it creates all the different pizza objects
 - It prepares, bakes, cuts, and boxes pizzas
- the pizzas it uses are low-level classes
 - pizza implementations are our “low-level classes”
 - VeggiePizza
 - NYStyle
 - ChicagoStyle
- the PizzaStore class is dependent on the concrete pizza classes
- this principle tells us we should write our code so that we are depending on abstractions, not concrete classes
 - applies to both our high-level classes and our low-level classes
 - we can create an abstract class named Pizza
 - the PizzaStore and the concrete pizzas both depend on the Pizza class (the abstraction)

PizzaStore (Example)

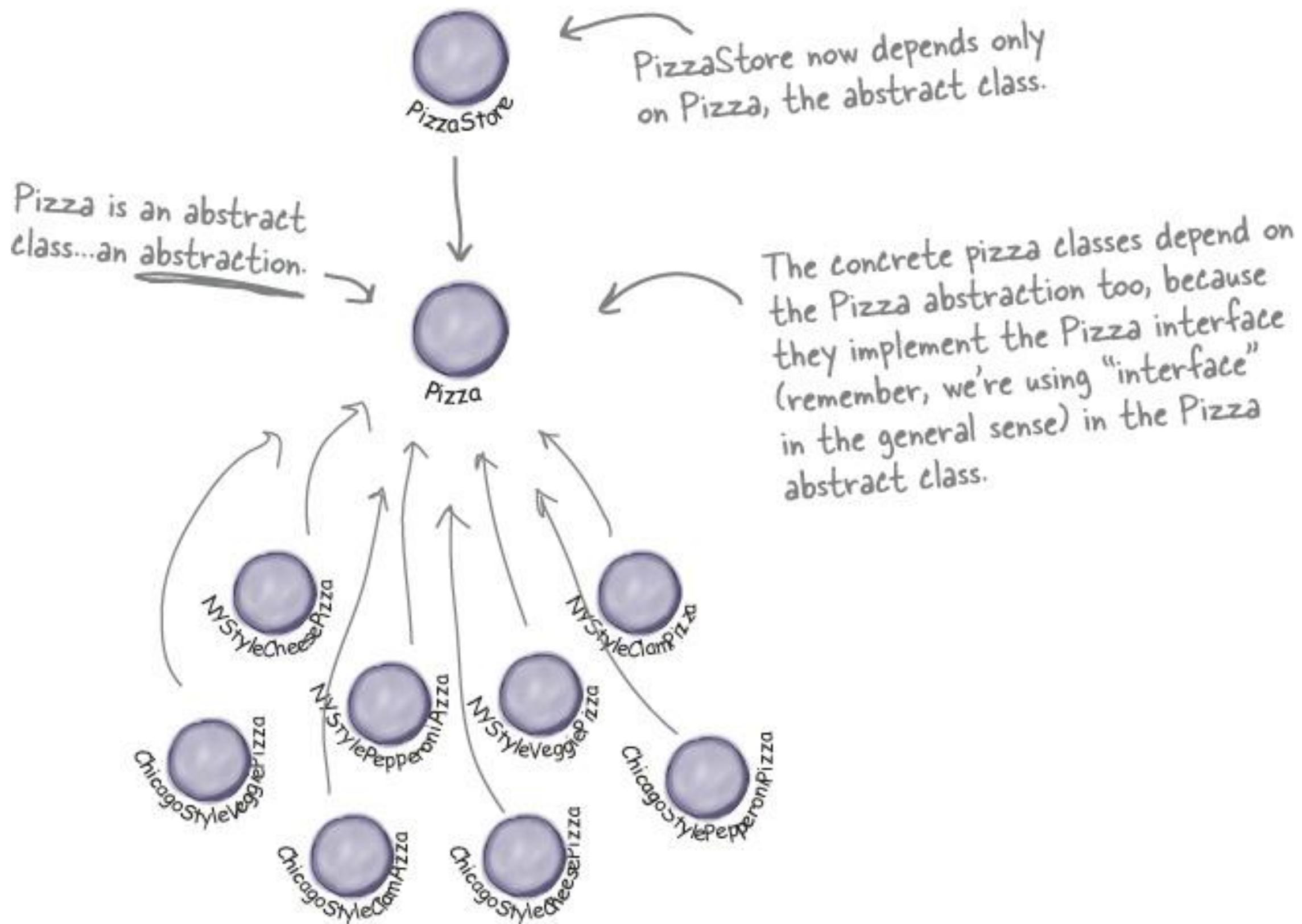


Image taken from:

"Head First Design Patterns
By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra"

Advantages of Dependency Inversion

- removes tight coupling that comes with a top-down design approach
 - each higher level class is tightly coupled with its lower level concrete class
 - any change in the lower level class will have a ripple effect in the next higher level class
 - makes it extremely difficult and costly to maintain and extend the functionality of the layers
- Dependency Inversion Principle introduces a layer of abstraction between each higher level class and lower level concrete class
 - higher-level classes depend only on a common abstraction
 - lower-level classes can then be modified or extended without the fear of disturbing higher-level classes
 - as long as it obeys the contract of the abstract interface
- dependency inversion provides loose coupling between higher and lower level classes by introducing an abstraction layer
 - highly beneficial for maintaining and extending the overall system

OO guidelines for adhering to DIP

- no variable should hold a reference to a concrete class
 - use the factory design pattern to avoid this
- no class should subclass from a concrete class
 - If you subclass from a concrete class, you are depending on a concrete class
 - subclass from an abstraction (an interface or an abstract class)
- no method should override an implemented method of any of its base classes
 - If you override an implemented method, then your base class was not really an abstraction to start with
 - methods implemented in the base class are meant to be shared by all your subclasses
- this is a guideline you should strive for, rather than a rule you should follow all the time
 - if you have a class that is not likely to change, and you know it, then it is ok to instantiate a concrete class
 - we instantiate String objects all the time and this violates the principle
 - however, the String class is very unlikely to change
- you should internalize these guidelines and have them in the back of your mind when you design

Another Example (demonstrate in intellij)

```
class PasswordReminder {  
    private dbConnection;  
  
    public PasswordReminder(MySQLConnection dbConnection) {  
        this.dbConnection = dbConnection;  
    }  
}
```

- the MySQLConnection is the low level module while the PasswordReminder is the high level module
 - according to the Dependency Inversion Principle, we must depend on abstraction, not on concretions
 - the above code violates this principle as the PasswordReminder class is being forced to depend on the MySQLConnection class
- if you were to change the database engine in the future, you would also have to edit the PasswordReminder class which violates the Open-close principle
- the PasswordReminder class should not care what database your application uses
 - since high level and low level classes should depend on abstraction, we can code to an “interface”

Another Example (demonstrate in intelliJ)

```
public interface DBConnectionInterface {  
    public function connect();  
}
```

- the interface has a connect method and the MySQLConnection class implements this interface
- the constructor of the PasswordReminder will now accept this interface as a parameter
 - no matter the type of database your application uses, the PasswordReminder class can easily connect to the database without any problems and OCP is not violated

Another Example (demonstrate in intellij)

```
class MySQLConnection implements DBConnectionInterface {  
    public String connect() {  
        return "Database connection";  
    }  
}  
  
class PasswordReminder {  
    private dbConnection;  
  
    public PasswordReminder(DBConnectionInterface dbConnection) {  
        this.dbConnection = dbConnection;  
    }  
}
```

- you can now see that both the high level and low level modules depend on abstraction

Dependency Injection

Jason Fedin

Dependencies

- a Java class has a dependency on another class, if it uses an instance of this class
 - referred to as a class dependency
 - a class which accesses a logger service has a dependency on this service class
- java classes should be as independent as possible from other Java classes
 - increases the possibility of reusing these classes and to be able to test them independently from other classes
- if a Java class creates an instance of another class via the new operator, it cannot be used (and tested) independently from this class
 - this is called a hard dependency
- dependency injection solves these “hard” dependencies

Example of a class that has a “hard” dependency (demo)

```
public class Client {  
    // Internal reference to the service used by this client  
    private ExampleService service;  
  
    // Constructor  
    Client() {  
        // Specify a specific implementation in the constructor instead of using dependency injection  
        service = new ExampleService();  
    }  
  
    // Method within this client that uses the services  
    public String greet() {  
        return "Hello " + service.getName();  
    }  
}
```

- the Client class contains a Service member variable that is initialized by the Client constructor
 - client controls which implementation of service is used and controls its construction
 - client has a hard-coded dependency on ExampleService

Problems with Example

- this kind of dependency is very trivial in programming
 - however, when the application's code gets bigger and more complex, the hard-coded dependency among classes introduces many problems
- the code is inflexible
 - hard to maintain and extend as when a class permanently depends on another class
 - change to the depending class may require change to the dependent class
- the code is hard to get under unit test
 - when you want to test only the functionalities of a class, you have to test other depending classes as well
- the code is hard for reuse because the classes are tightly coupled
- dependency injection solves these drawbacks by making the code more flexible to changes, easy for unit testing and reusable

Dependency Injection

- dependency injection is a technique whereby one object supplies the dependencies of another object
 - enables you to replace dependencies without changing the class that uses them
- a dependency is an object that can be used (a service)
- an injection is the passing of a dependency to a dependent object (a client) that would use it
- allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable
- dependency injection is one form of the broader technique of dependency inversion
 - supports the dependency inversion principle
- the client delegates the responsibility of providing its dependencies to external code (the injector)

Reminder (Dependency Inversion principle)

- you can introduce interfaces to break the dependencies between higher and lower level classes
 - both classes depend on the interface and no longer on each other
 - this is the central focus of the dependency inversion principle
- you still have a dependency on the lower level class with the dependency inversion principle
 - interface only decouples the usage of the lower level class but not its instantiation
 - you still need to instantiate the implementation of the interface
- goal of the dependency injection technique is to remove the above dependency by separating the usage from the creation of the object
 - reduces the amount of required boilerplate code and improves flexibility

4 roles in dependency injection

- if you want to use dependency injection, you need classes that fulfill four basic roles
 - the service you want to use
 - the client that uses the service
 - an interface that is used by the client and implemented by the service
 - the injector which creates a service instance and injects it into the client
- you already implement three of these four roles by following the dependency inversion principle
 - the service and the client are the two classes between which the dependency inversion principle intends to remove the dependency by introducing an interface
- the injector is the only role that is not required by the dependency inversion principle

Injection Types

- there are at least three ways an object can receive a reference to an external object
- constructor injection
 - the dependencies are provided through a class constructor
- setter injection
 - the client exposes a setter method that the injector uses to inject the dependency
- interface injection
 - the dependency provides an injector method that will inject the dependency into any client passed to it
 - clients must implement an interface that exposes a setter method that accepts the dependency

Constructor injection

- this method requires the client to provide a parameter in a constructor for the dependency

```
// Constructor  
Client(Service service) {  
    // Save the reference to the passed-in service inside this client  
    this.service = service;  
}
```

Setter injection

- this method requires the client to provide a setter method for the dependency

```
// Setter method  
public void setService(Service service) {  
    // Save the reference to the passed-in service inside this client  
    this.service = service;  
}
```

Interface injection

- this is simply the client publishing a role interface to the setter methods of the client's dependencies
 - can be used to establish how the injector should talk to the client when injecting dependencies

```
// Service setter interface.  
public interface ServiceSetter {  
    public void setService(Service service);  
}  
  
// Client class  
public class Client implements ServiceSetter {  
    // Internal reference to the service used by this client.  
    private Service service;  
  
    // Set the service that this client is to use.  
    @Override  
    public void setService(Service service) {  
        this.service = service;  
    }  
}
```

IntelliJ Example with 4 main roles

```
public interface Service {  
    void write(String message);  
}
```

```
public class ServiceA implements Service {  
    @Override  
    public void write(String message) {  
        System.out.println("Hello World");  
    }  
}
```

Example with 4 main roles

```
public class Client {  
    private Service myService;  
  
    // injects via the constructor  
    public Client(Service service) {  
        this.myService = service;  
    }  
  
    public void doSomething() {  
        myService.write("This is a message.");  
    }  
}
```

Example with 4 main roles

```
Service service = new ServiceA(); // the injector  
Client client = new Client(service); // injects via the constructor  
client.doSomething();
```

- setter injection could be used instead to pass the depending object to the dependent one

```
// add to client class, remove Service parameter from constructor  
public void setService(Service service) {  
    this.service = service;  
}
```

```
Service service = new ServiceA();  
Client client = new Client();  
Client.setService(service);  
client.doSomething();
```

Summary

- the dependency inversion principle introduces interfaces between a higher-level class and its dependencies
 - decouples the higher-level class from its dependencies so that you can change the code of a lower-level class without changing the code that uses it
 - only code that uses a dependency directly is the one that instantiates an object of a specific class that implements the interface
- the dependency injection technique enables you to improve this even further
 - provides a way to separate the creation of an object from its usage
 - you can replace a dependency without changing any code and it also reduces the boilerplate code in your business logic

UML

Jason Fedin

Overview

- UML is an acronym that stands for the “Unified Modelling Language”
- modeling is an activity that has been carried out over the years in software development during the design phase
- UML is a modeling language in the field of software engineering
 - it is intended to provide a standard way in visualizing the design of a system
- UML is a visual language (meaning a drawing notation with semantics) used to create models of programs
 - the term *models of programs* means diagrammatic representations of the programs that show the relationships among the objects in the code
 - It is a documenting tool that illustrates the objects in a software system
- UML has a direct relation with object oriented analysis and design and is often used to illustrate design pattern concepts
- the UML has several different diagrams
 - some for analysis and others for design
 - each diagram shows the relationships among the different sets of entities, depending on the purpose of the diagram

Types of UML Diagrams

- UML includes the following nine diagrams
 - Class
 - Object
 - Use case
 - Sequence
 - Collaboration
 - Activity
 - Statechart
 - Deployment
 - Component
- we will mainly focus on class diagrams in this class that will help us understand each design pattern that we study

Goals of UML

- the purpose of the UML is to both flesh out your designs and to communicate them
 - do not worry so much about creating diagrams the “right” way
 - think about the best way to communicate the concepts in your design
- UML is simple to use and all modelers can use it
 - simple enough that they can be used by business users, common people, and anybody interested in understanding the system
- UML diagrams aim to be clear and precise
 - this means you should not use the UML in nonstandard ways that does not communicate properly
- the goal of UML can be defined as a simple modeling mechanism to model all possible practical systems in today's complex environment

Why use the UML?

- the UML is used primarily for communication
 - with myself, my team members, and with my customers
- poor requirements (either incomplete or inaccurate) are ubiquitous in the field of software development
 - UML gives us tools to elicit better requirements
- the UML gives a way to help determine whether my understanding of the system is the same as others'
 - systems are complex and have different types of information that must be conveyed
 - UML offers different diagrams specializing in the different types of information
- one easy way to see the value of the UML is to recall your last several design reviews
 - very difficult to describe software design in a review without using a modeling language
 - very confusing and much longer than necessary
- the UML is a much better way of describing object-oriented designs
 - forces the designer to think through the relationships between classes in his or her approach because they actually write down the design

Object-Oriented Concepts

- UML can be described as the successor of object-oriented (OO) analysis and design
 - an object contains both data and methods that control the data
 - data represents the state of the object
 - a class describes an object and they also form a hierarchy to model the real-world system
- objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML
 - UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design
- UML can document the following:
 - Objects – represent an entity and the basic building block
 - Classes –the blue print of an object
 - Abstraction – represents the behavior of an real world entity
 - Encapsulation – the mechanism of binding the data together and hiding them from the outside world
 - Inheritance – the mechanism of making new classes from existing ones
 - Polymorphism – defines the mechanism to exists in different forms

OO Analysis and Design

- the most important purpose of OO analysis is to identify objects of a system to be designed
 - this analysis is also done for an existing system
- after identifying the objects, their relationships are identified and finally the design is produced using UML
- the purpose of OO analysis and design can be described as
 - identifying the objects of a system
 - identifying their relationships
 - making a design, which can be converted to executables using OO languages

Role of UML in OO Design and Design Patterns

- the relationship between OO design and UML is very important to understand
- OO design is transformed into UML diagrams according to the requirements
- before understanding the UML in detail, the OO concept should be learned properly
- once the OO analysis and design is done, the next step is very easy
 - documenting the design using UML
 - the input from OO analysis and design is the input to UML diagrams
- we need to understand UML in this class because we need to understand how a specific design pattern works
 - UML documents and clearly communicates how these patterns behave

Class Diagrams

Jason Fedin

Overview

- a Class represents a set of objects having similar responsibilities
- a class diagram is used to define a detailed design of the system
- class diagrams describe the attributes (member variables) and operations (methods) of a class
- class diagrams show a collection of classes, interfaces, associations, collaborations, and constraints
 - relationships between classes
 - known as a structural diagram
- widely used in the modeling of object oriented systems because they are the only UML diagrams which can be mapped directly with object-oriented languages
- class diagrams can also generate executable code of the software application

Purpose of Class Diagrams

- the purpose of class diagram is to model the static view of an application
 - a graphical representation
 - describes the vocabulary of the system
- a collection of class diagrams represent the whole system
- the most popular UML diagram in the coder community
- class diagrams describe the responsibilities of a system
 - the functionalities performed by the system
- class diagrams provide a base for component and deployment diagrams
- class diagrams provide forward and reverse code engineering

Drawing a class diagram

- the name of the class diagram should be meaningful to describe the aspect of the system
- each element and their relationships should be identified in advance
- responsibility (attributes and methods) of each class should be clearly identified
- for each class, a minimum number of properties should be specified, as unnecessary properties will make the diagram complicated
- use notes whenever required to describe some aspect of the diagram
 - the final diagram should be understandable to the developer/coder
- before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct

Class Relationships

- when one class is a “kind of” another class, this is referred to as a is-a relationship (inheritance)
- when there are associations between two classes
 - one class “contains” another class (has-a relationship)
 - one class “uses” another class (uses-a relationship)
 - one class “creates” another class
- there are variations on these themes
 - the contained item is a part of the containing item (such as an engine in a car) (composition)
 - i have a collection of things that can exist on their own (such as airplanes at an airport) (aggregation)

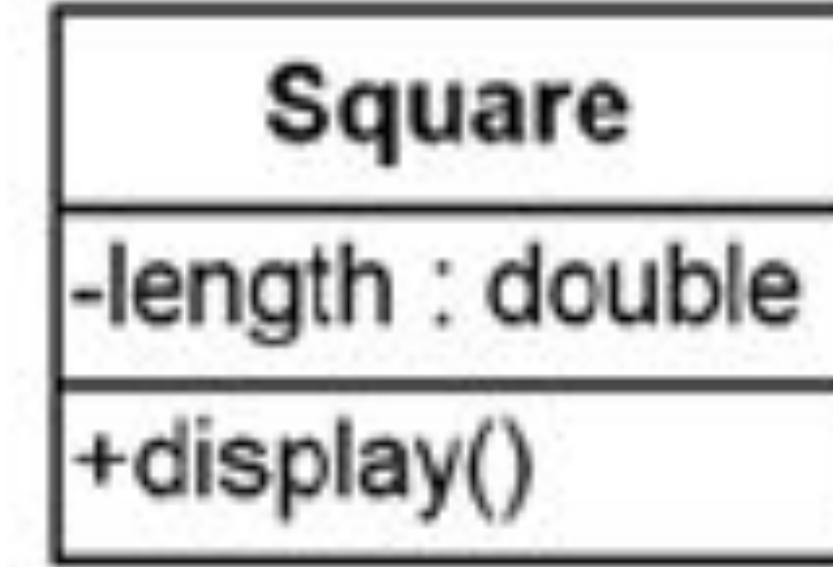
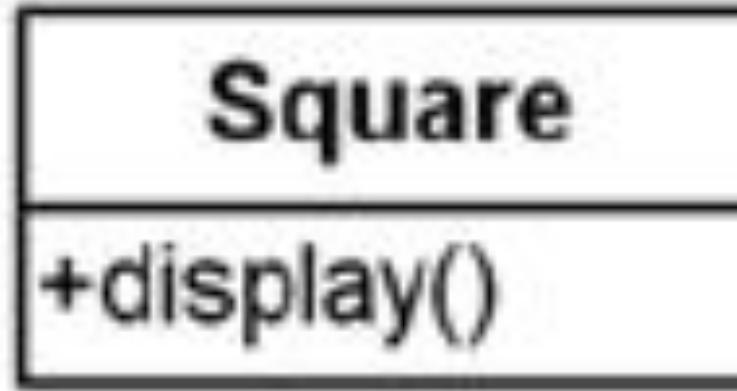
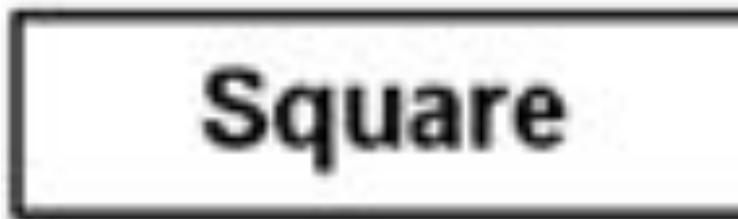
Drawing a Class Diagram

Jason Fedin

Overview

- UML is popular for its diagrammatic notations
 - visualization is the most important part which needs to be understood and remembered
- UML notations are the most important elements in modeling
 - efficient and appropriate use of notations is very important for making a complete and meaningful model
 - the model is useless, unless its purpose is depicted properly
- learning notations should be emphasized from the very beginning
 - different notations are available for classes and their relationships
- each rectangle in a class diagram represents a class
- you can represent up to three aspects of a class
 - the name of the class
 - the data members of the class
 - the methods (functions) of the class

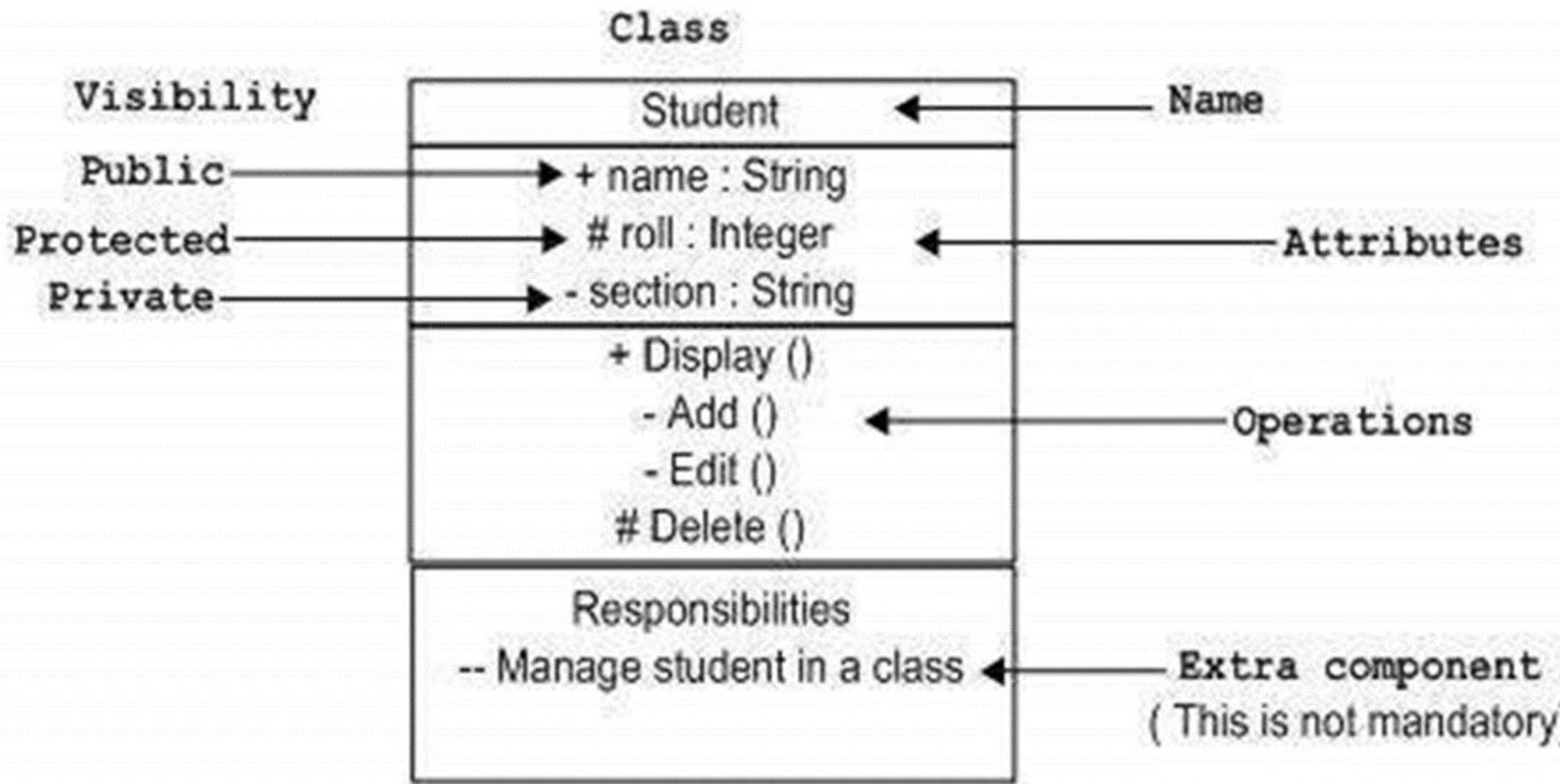
Basic notation



- the leftmost rectangle shows just the class name
 - use this type of class representation when more detailed information is not needed
- the middle rectangle shows both the name and the methods of the class
 - the Square has a method display
 - the plus sign (+) in front of display (the name of the method) means that this method is public
- the rightmost rectangle shows the name, methods of the class, and the data members of the class
 - the minus sign (-) before the data member length (which is of type double) indicates that this data member's value is private

a 4th section

- you will sometimes see a 4th section
 - optional to show any additional components



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition
By: Alan Shalloway; James R. Trott

UML Notation for Access

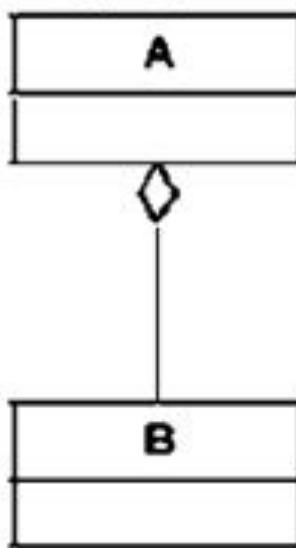
- you can control the accessibility of class data and method members
- public
 - notated with a plus sign (+)
 - all objects can access this data or method
- protected
 - notated with a pound sign (#)
 - means only this class and all of its subclasses can access this data or method
- private
 - notated with a minus sign (-)
 - means that only methods of this class can access this data or method

Class Relationships

- class diagrams can also show relationships between different classes
 - gives a proper meaning to a UML model
- the different types of relationships available in UML are:
 - extensibility - inheritance, the “is-a” relationship
 - association - describes how many elements are taking part in an interaction
 - aggregation or composition – “has a” relationship
 - dependency - represents the dependency between two classes in a system - “uses a” relationship

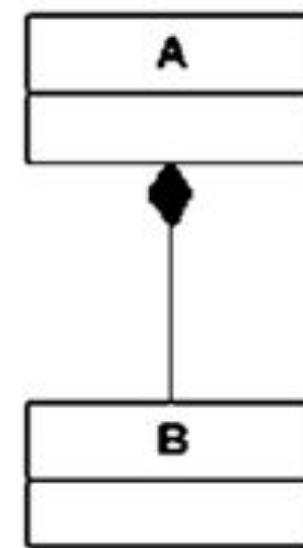
Class Relationships (illustration)

Aggregation



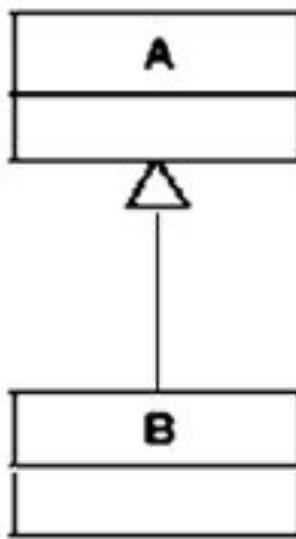
A aggregates B
B is a part of A

Composition



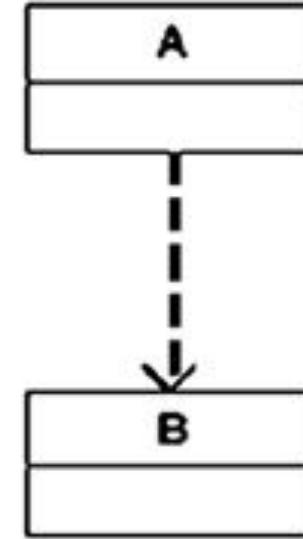
A is composed of B
A contains B type objects

Inheritance



B is derived from A
A generalizes B

Dependency

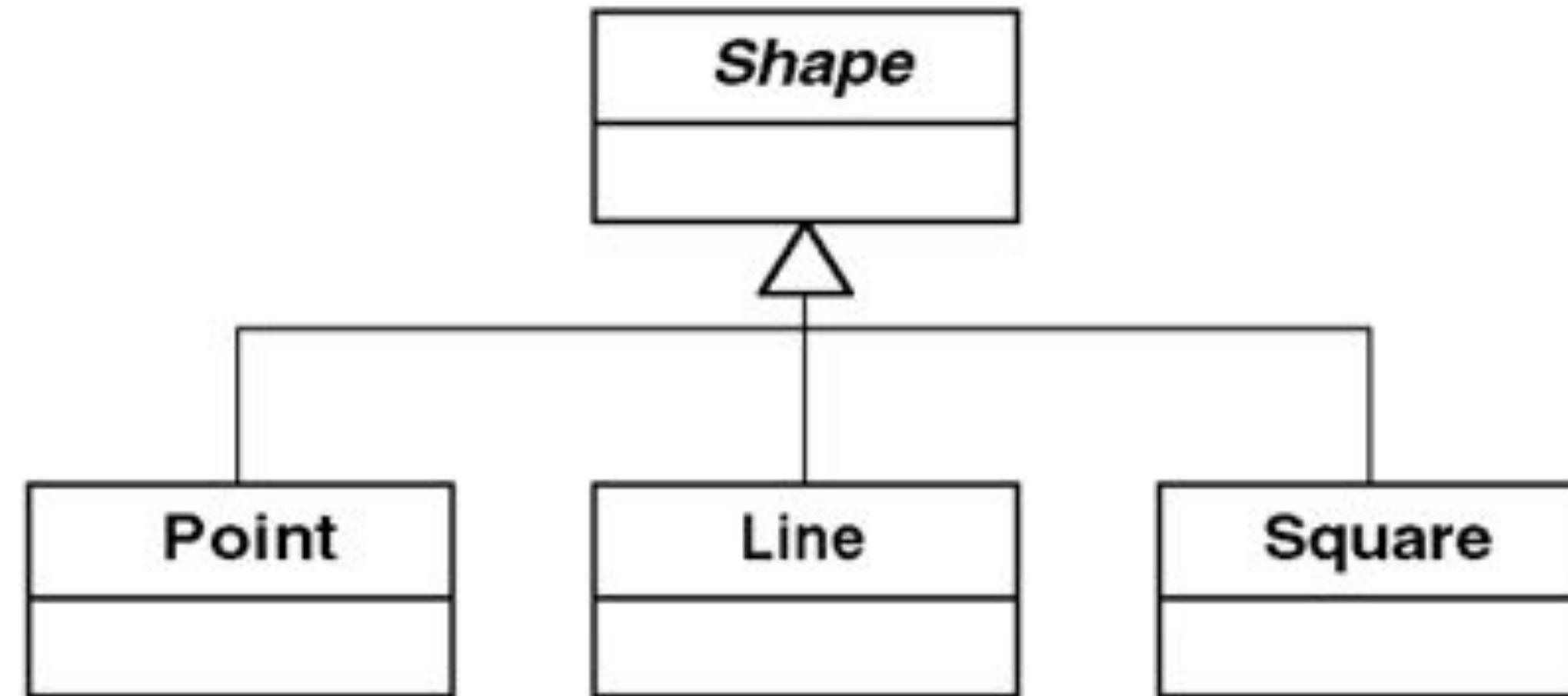


A depends upon B
A uses B

Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition
By: Alan Shalloway; James R. Trott

showing the is-a relationship

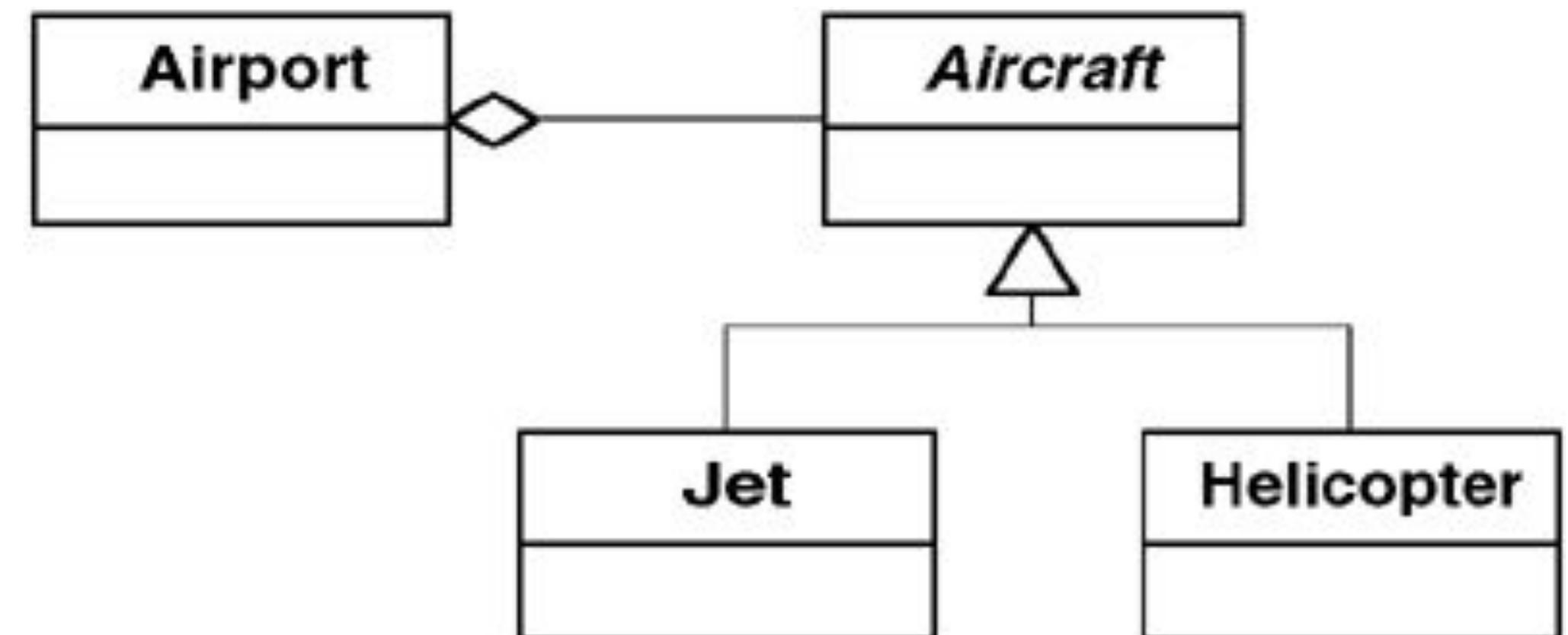
- the arrowhead under the Shape class means that those classes pointing to Shape derive from Shape
 - one end represents the parent element and the other end represents the child element
 - because Shape is *italicized*, that means it is an abstract class



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition
By: Alan Shalloway; James R. Trott

showing the has-a relationship (Aggregation)

- there are two different kinds of *has-a* relationships
 - Airport “having” Aircraft (aggregation)
 - Aircraft are not part of Airport, but I can still say the Airport has them
- the open (unfilled) diamond on the right of the Airport class indicates the aggregation relationship
- aircraft is either a Jet or a Helicopter (is-a relationship)
 - aircraft is an abstract class or an interface because its name is shown in italics

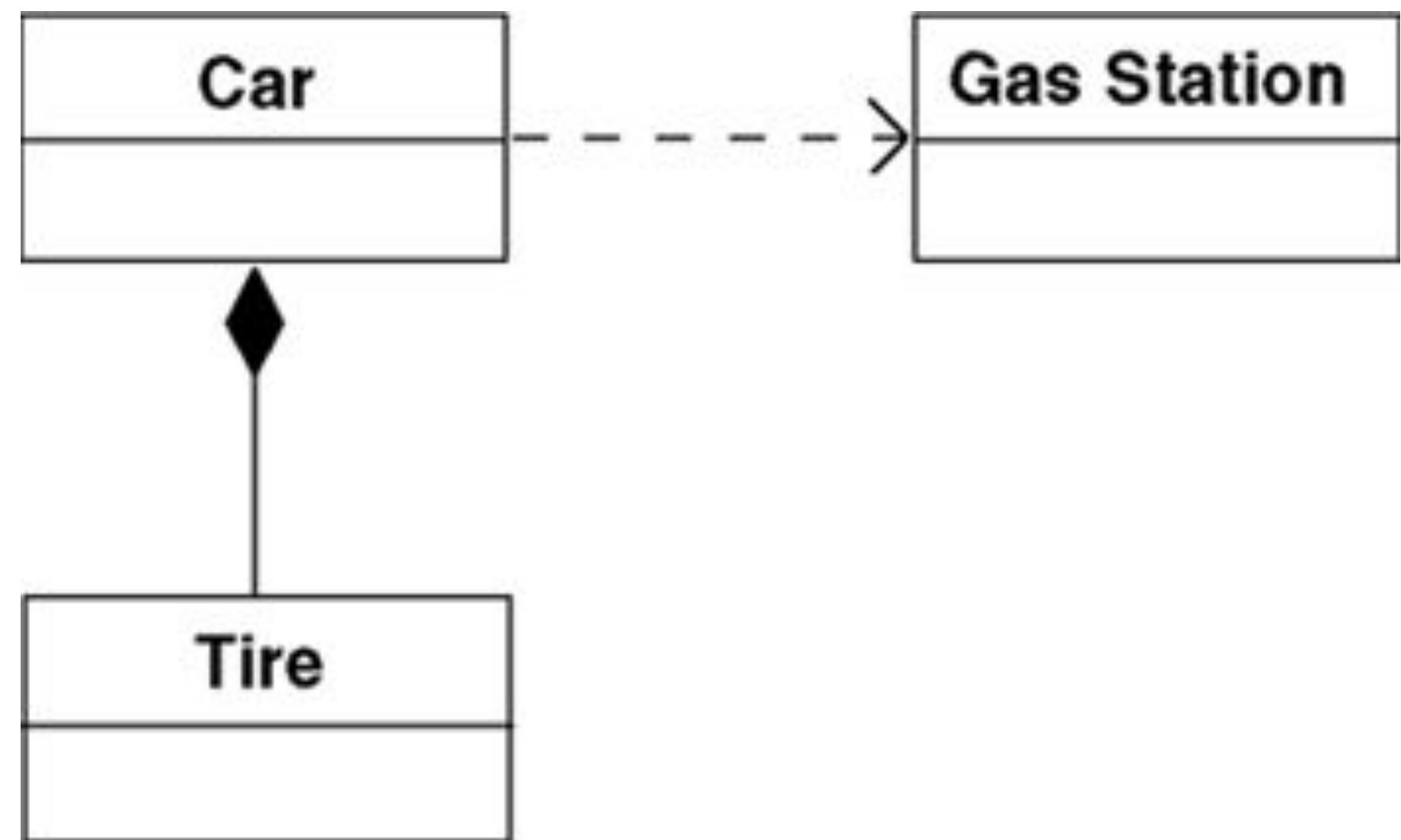


Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition

By: Alan Shalloway; James R. Trott

showing the has-a relationship (composition)

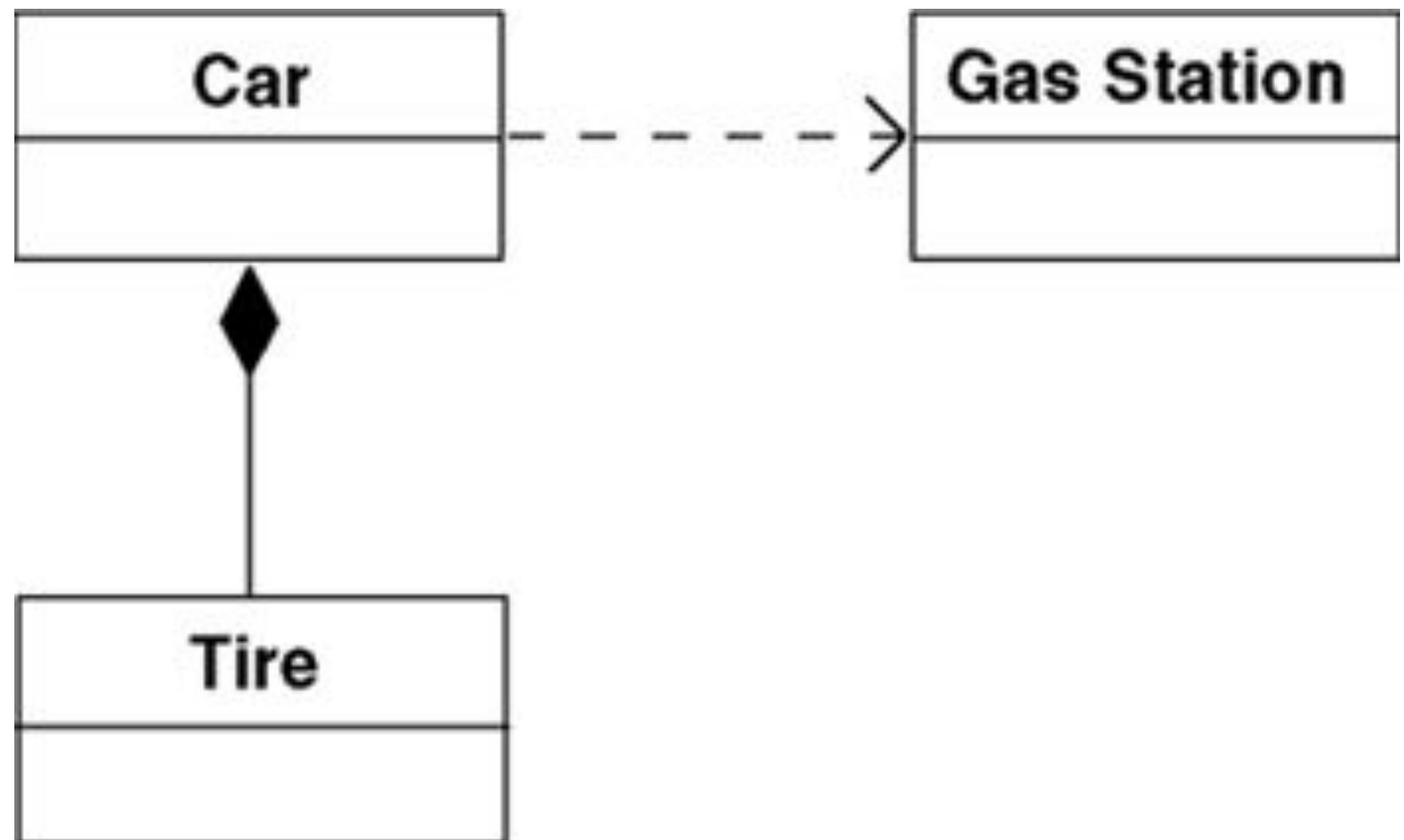
- the other type of has-a relationship is where the contained object is a part of the containing object (composition)
 - a Car has Tires as parts (the Car is made up of Tires and other things)
 - this type of relationship is depicted by the solid diamond



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition
By: Alan Shalloway; James R. Trott

showing a dependency relationship

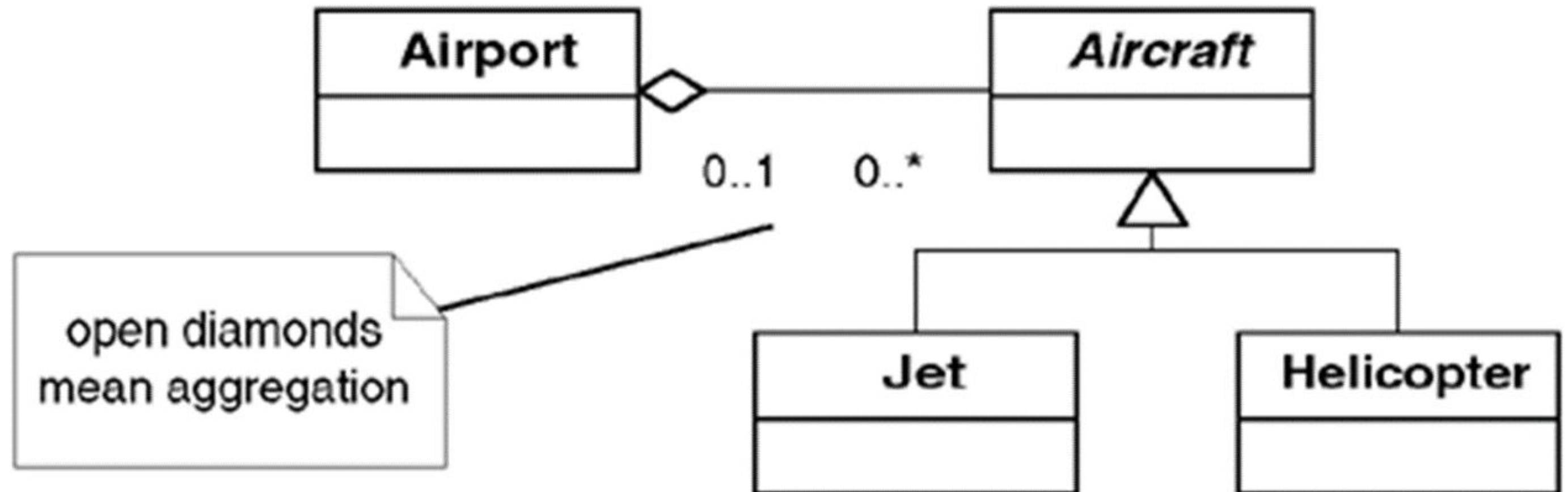
- the below diagram shows that a Car uses a Gas Station
- the uses relationship is depicted by a dashed line with an arrow
 - a dependency exists between the two



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition
By: Alan Shalloway; James R. Trott

Cardinality

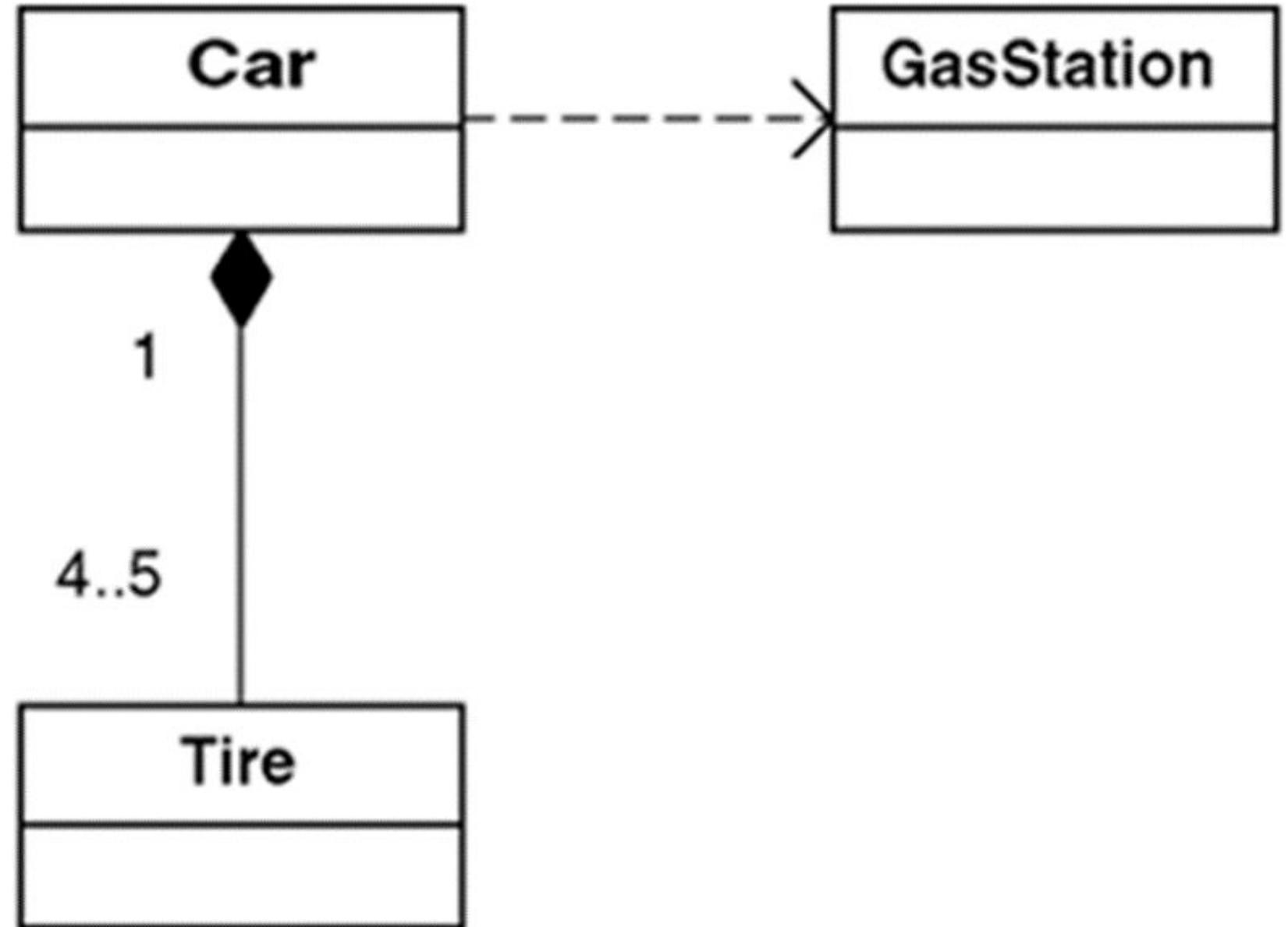
- the cardinality of the relationship between objects describe how many objects another one has
- the below shows us that when I have an Airport, it has from 0 to any number (represented by an asterisk here) of Aircraft
- 0..1 on the Airport side means that when I have an Aircraft, it can be contained by either 0 or 1 Airport (It may be in the air)



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition
By: Alan Shalloway; James R. Trott

Cardinality (cont'd)

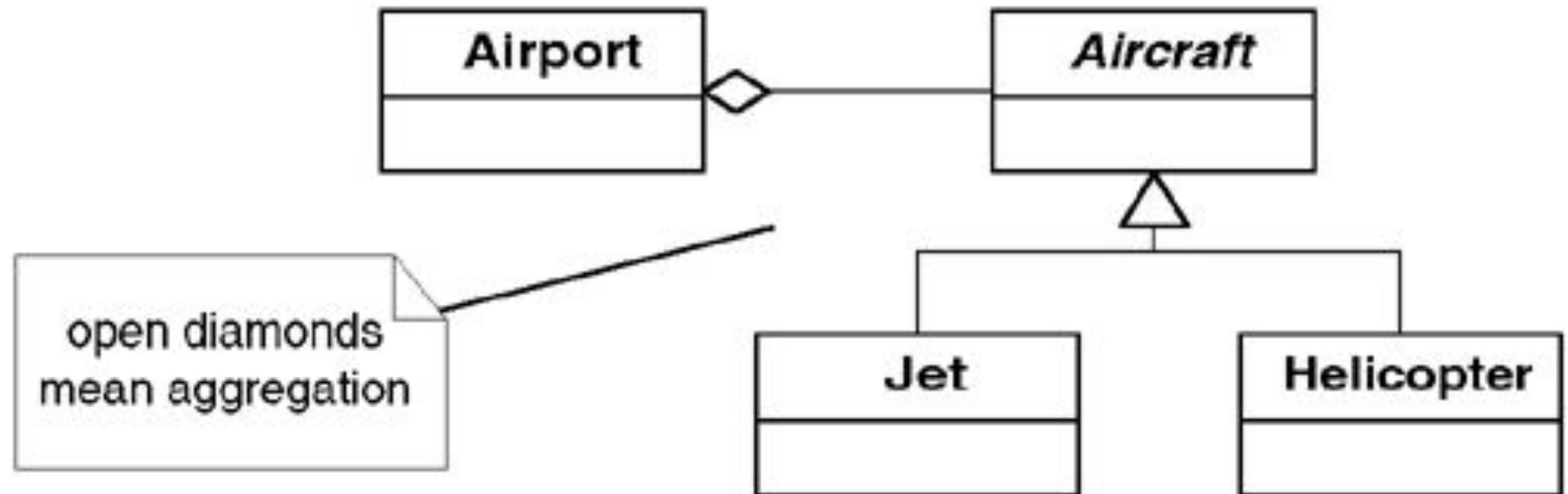
- the below shows us that when I have a Car, it has either 4 or 5 tires (it may or may not have a spare)
- tires are on exactly one car
- if cardinality is not specified, no assumption is made as to how many objects there are



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition
By: Alan Shalloway; James R. Trott

Notes

- the box containing the message “open diamonds mean aggregation” is a note
 - meant to look like pieces of paper with the right corner folded back
 - often see them with a line connecting them to a particular class, indicating they relate just to that class



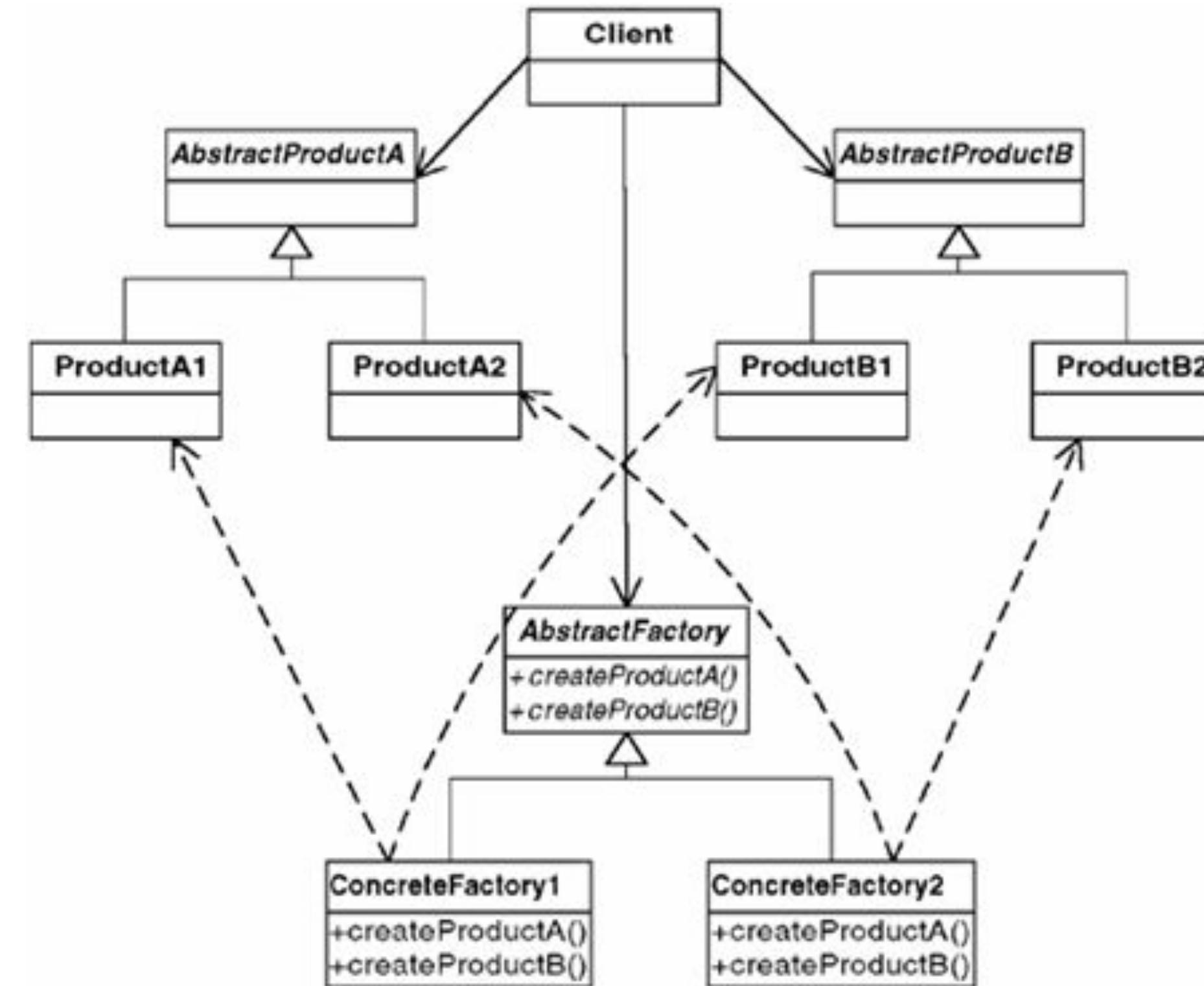
Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition

By: Alan Shalloway; James R. Trott

Some Example Class Diagrams

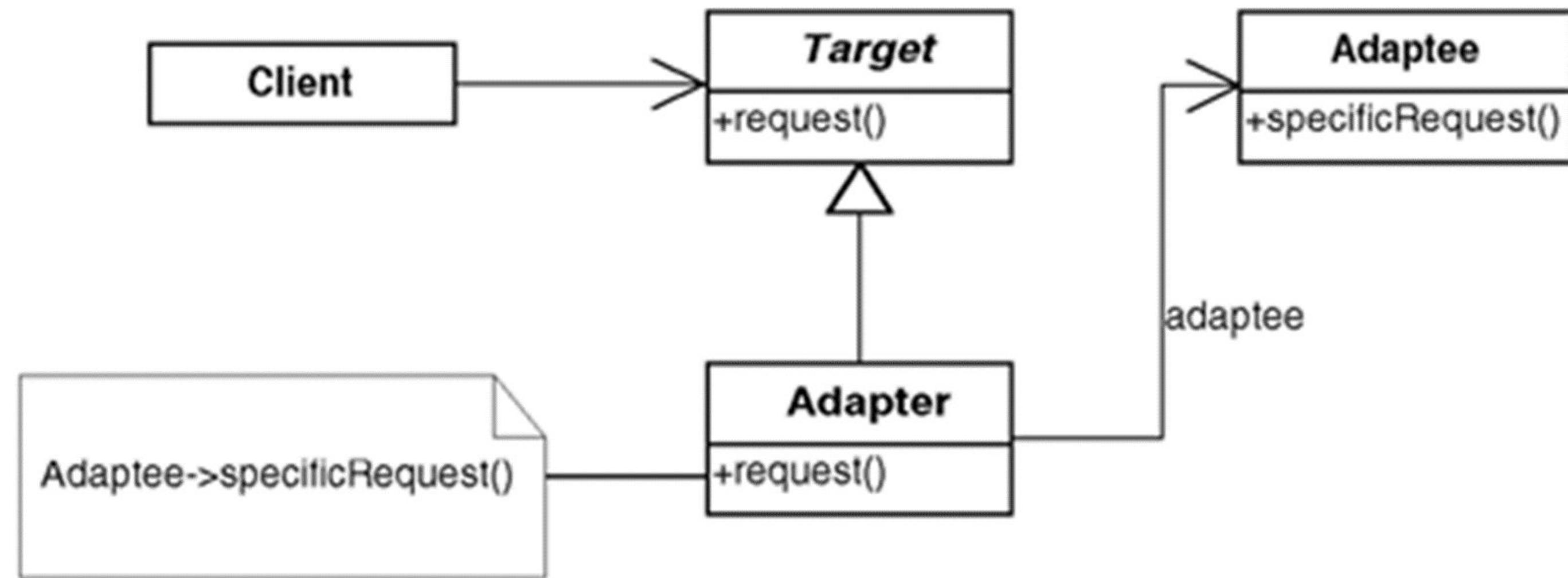
Jason Fedin

AbstractFactory Design Pattern



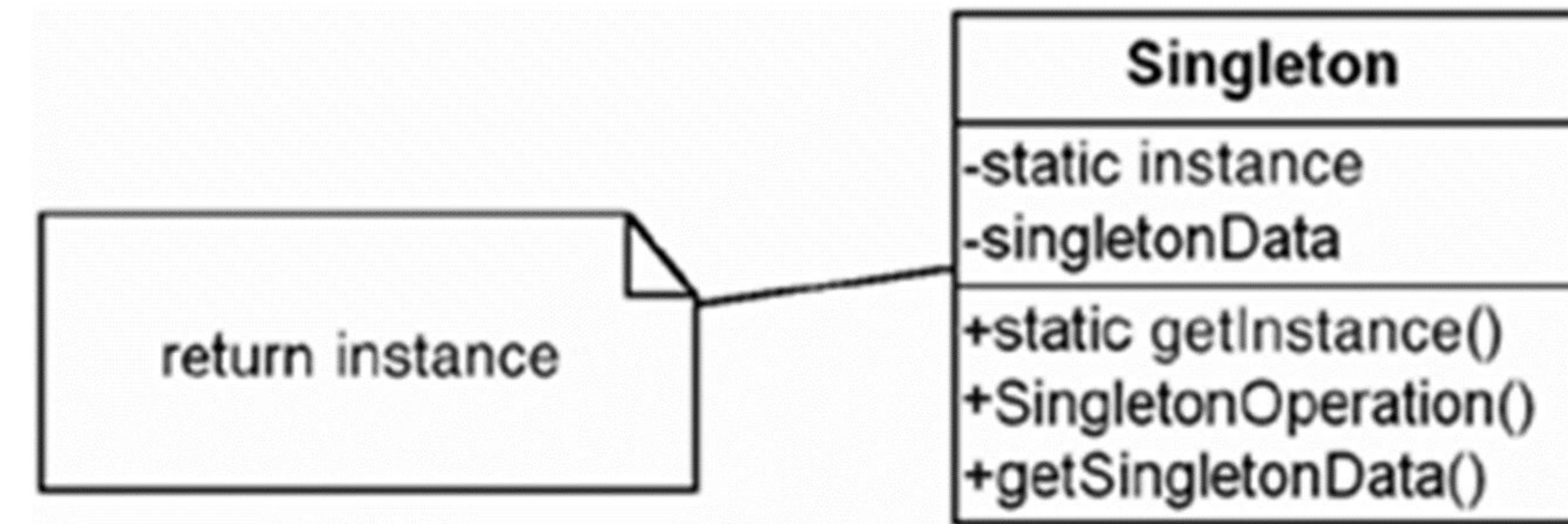
Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Adapter Design Pattern



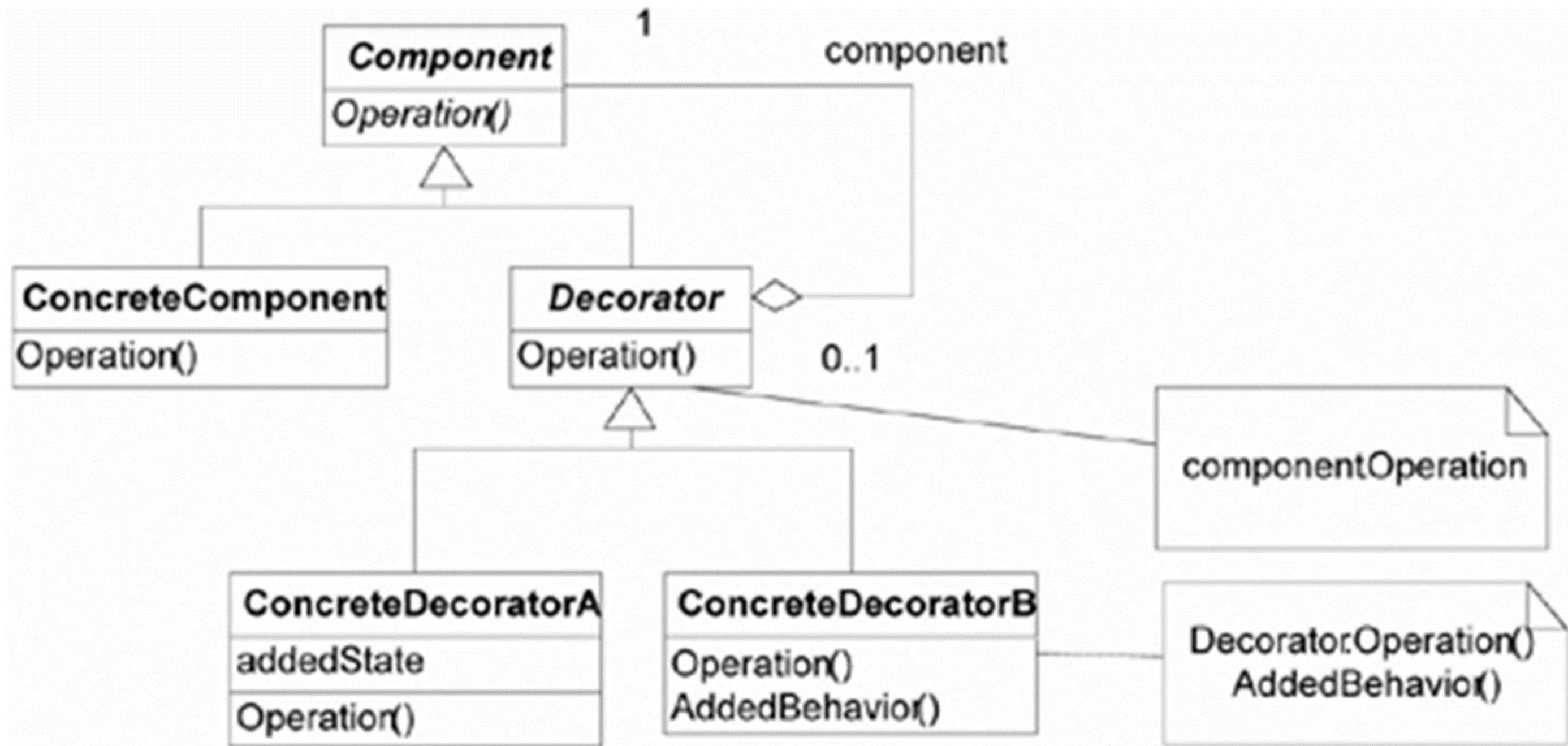
Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Singleton Design Pattern



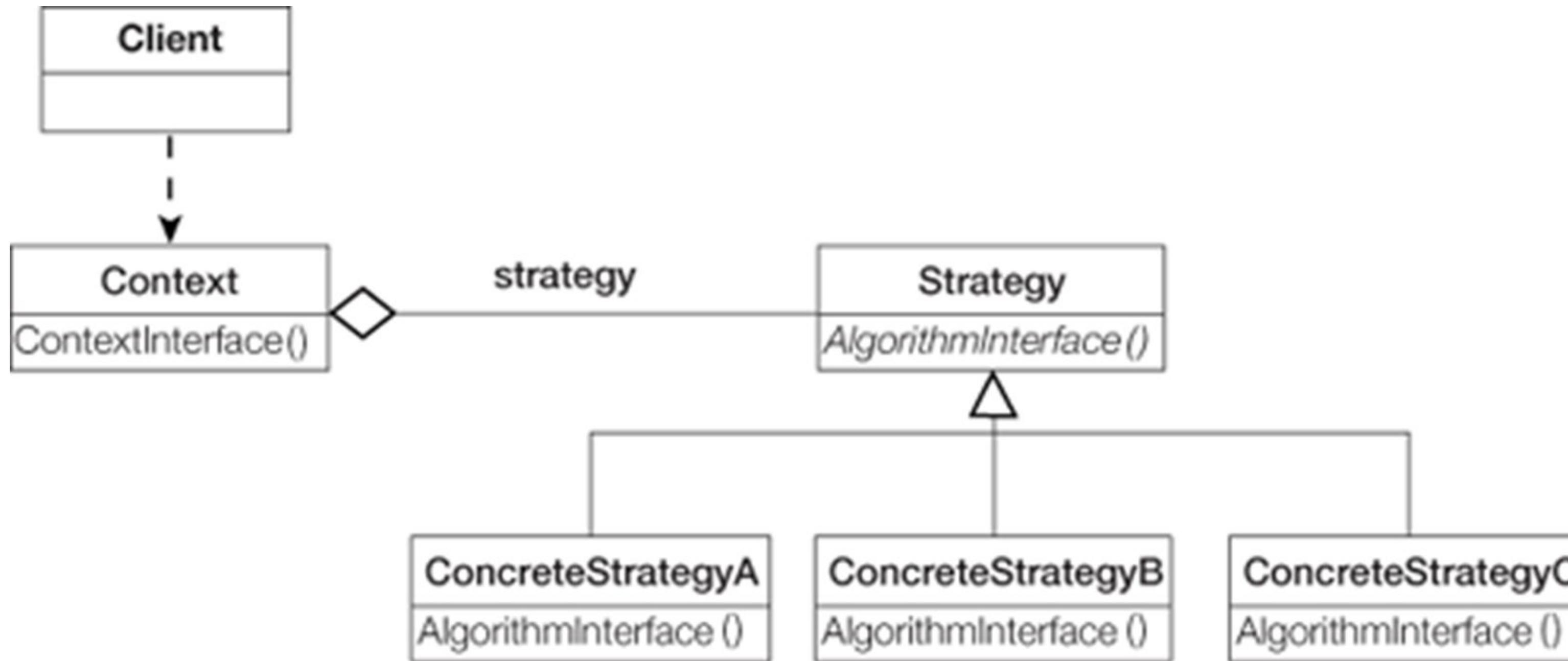
Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Decorator Design Pattern



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Strategy Design Pattern



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Creational Design Patterns

Jason Fedin

Creational Patterns

- a program should not depend on how objects are created and arranged
- instantiation is an activity that should not always be done in public and can often lead to coupling problems
- in Java, the simplest way to create an instance of an object is by using the new operator
 - fred = new Fred(); //instance of Fred class
 - creates a concrete class
- tying your code to a concrete class can make it more fragile and less flexible
 - code may have to be changed as new concrete classes are added
 - your code will not be “closed for modification”
 - to extend it with new concrete types, you will have to reopen it

Creational Patterns (cont'd)

- creational design patterns provide a way to create objects
- creational design patterns abstract the instantiation process
 - the creation logic is hidden
 - encapsulates knowledge about which concrete classes the system uses
 - programmer may call a method or use another object, rather than instantiating objects directly using the new operator
- all the system at large knows about the objects is their interfaces as defined by abstract classes
 - gives the programmer a lot of flexibility in what gets created, who creates it, how it gets created, and when
 - lets you configure a system with “product” objects that vary widely in structure and functionality
 - configuration can be static (compile-time) or dynamic (at run-time)

Creational Patterns (cont'd)

- by coding to an interface, you can insulate yourself from a lot of changes that might happen to a system down the road
- sometimes creational patterns are competitors
 - there are cases when either Prototype or Abstract Factory could be used profitably
- sometimes creational patterns are complementary
 - builder can use one of the other patterns to implement which components get built
 - prototype can use Singleton in its implementation

Class Patterns vs. Object Patterns (sub-categories)

- class patterns describe how relationships between classes are defined
 - use inheritance
 - relationships are established at compile time
 - Factory pattern
 - drawback of this approach is that it can require creating a new subclass just to change the class of the product
 - changes can cascade
 - when the product creator is itself created by a factory method, then you have to override its creator as well
- object patterns describe relationships between objects
 - use composition
 - relationships are typically created at runtime
 - more dynamic and flexible
 - abstract factory, singleton, builder, and prototype patterns
- there are five creational patterns that we will study
 - will highlight their similarities and differences

Creational Patterns that we will study

- Factory
- Abstract Factory
- Singleton
- Builder
- Prototype

Factory Method

Jason Fedin

Overview

- one of the most used design patterns in Java
 - a creational pattern
 - factories handle the details of object creation
- this pattern defines an interface for creating an object (Creator)
- when a class needs to instantiate a subclass of another class, but doesn't know which one
 - it lets subclasses decide which class to instantiate
- creates objects without exposing the creation logic to the client (Creator) and refers to the newly created object using a common interface (Product)
- gives us a way to encapsulate the instantiations of concrete types

Frameworks

- the factory method is used in frameworks
 - frameworks exist at an abstract level
- frameworks use abstract classes to define and maintain relationships between objects
 - often responsible for creating these objects as well
- the framework should not know and should not be concerned about instantiating specific objects
 - need to defer the decisions about specific objects to the users of the framework
- use the Factory Method pattern when
 - a class cannot anticipate the class of objects it must create
 - a class wants its subclasses to specify the objects it creates
- also useful when implementing parallel class hierarchies
 - when some of the responsibilities shift from one class to another

Example

- suppose you have two different types of televisions
 - one with an LED screen and another with an LCD screen
- If either of the tv's start to malfunction, you will call a TV repairman
- the repairman must ask first what kind of TV is broken
- as per your input, he will carry the required instruments with him to fix the tv

JDK Examples

- examples in the JDK that use the factory method pattern
 - java.util.Calendar, ResourceBundle and NumberFormat
 - getInstance() methods uses Factory pattern
 - valueOf() method in wrapper classes like Boolean, Integer etc.
- SAXParserFactory
 - a factory class which can be used to intantiate SAX based parsers to pares XML
 - method newInstance is the factory method which instantiates the sax parsers based on some predefined logic

Advantages

- decouples the business logic of creation of a class from the actual logic of the class (i.e. decouples the implementation of the product from its use)
 - you can add additional products or change a product's implementation and it will not affect your Creator
 - the Creator is not tightly coupled to any ConcreteProduct
- allows you to change the design of your application more readily
 - makes our code more robust, less coupled and easy to extend
- promotes the approach of coding to an interface rather than implementation
- provides abstraction between implementation and client classes through inheritance
- connects parallel class hierarchies

Disadvantages

- clients might have to subclass the Creator class just to create a particular ConcreteProduct object
 - the client now must deal with another point of evolution
- makes code more difficult to read as all of your code is behind an abstraction that may in turn hide abstractions
- can be classed as an anti-pattern when it is incorrectly used
 - some people use it to wire up a whole application
- sometimes making too many objects often can decrease performance

Master: Grasshopper, tell me how your training is going.

Student: Master, I have taken my study of “encapsulate what varies” further.

Master: Go on...

Student: I have learned that one can encapsulate the code that creates objects. When you have code that instantiates concrete classes, this is an area of frequent change. I've learned a technique called “factories” that allows you to encapsulate this behavior of instantiation.

Master: And these “factories,” of what benefit are they?

Student: There are many. By placing all my creation code in one object or method, I avoid duplication in my code and provide one place to perform maintenance. That also means clients depend only upon interfaces rather than the concrete classes required to instantiate objects. As I have learned in my studies, this allows me to program to an interface, not an implementation, and that makes my code more flexible and extensible in the future.

Master: Yes Grasshopper, your OO instincts are growing

Student: Master, I know that by encapsulating object creation I am coding to abstractions and decoupling my client code from actual implementations

Factory Method Implementation

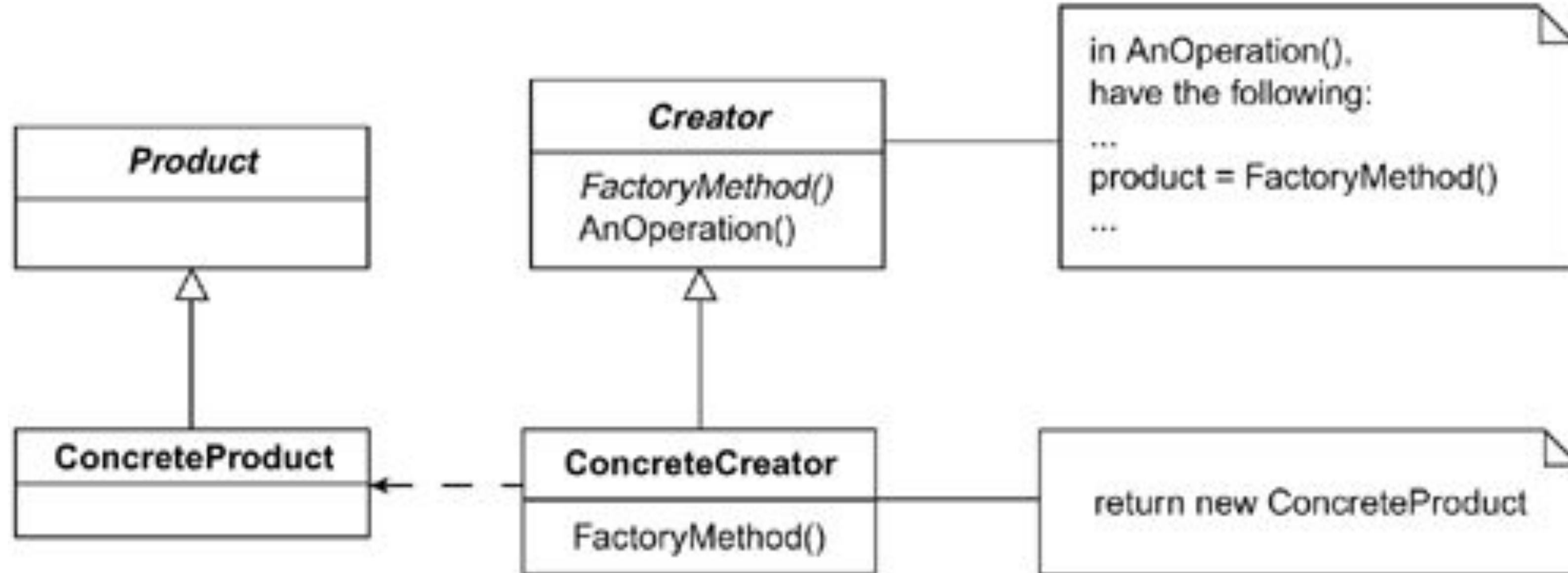
(Abstract Creator)

Jason Fedin

Overview

- there are 3 common implementations of the factory method pattern
- the first one I will describe is the “strictest” implementation of the pattern
 - the creator class is an abstract class
 - you create a subclass of the creator class for each product type which contains an implementation of the factory method
 - to use the factory method (create objects), you simply specify an instance of that type and invoke the factory method
- the disadvantage of this approach is that every new product has to subclass the creator class and implement its factory method

Class Diagram



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Most “strict” implementation

- product is the interface for the type of object that the Factory Method creates
- creator is the interface that defines the Factory Method
 - any other methods implemented here are written to operate on products produced by the factory method
 - the creator class is written without knowledge of the actual products that will be created
- clients will need to subclass the Creator class to make a particular concrete product
 - only subclasses actually implement the factory method and create products
- the actual products that will be created is decided purely by the choice of the subclass that is used

Example Overview

- we are going to create a Shape interface and concrete classes that implement this Shape interface (Products)
- we are then going to create an abstract class named ShapeFactory and subclasses that extend this ShapeFactory (Creators)
 - each subclass which will implement our factory method
- lastly, we then create a FactoryPatternDemo class
 - will use subclass ShapeFactory objects to get a Shape object

Demo walkthrough

- Step 1 - Create an interface (Shape.java)

```
public interface Shape {  
    void draw();  
}
```

- Step 2 - Create concrete classes implementing the same interface (Rectangle.java, Square.java, Circle.java)

```
public class Rectangle implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Inside Rectangle::draw() method.");  
    }  
}
```

Demo walkthrough

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Square::draw() method.");  
    }  
}
```

```
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Circle::draw() method.");  
    }  
}
```

Demo walkthrough

- Step 3 - Create an abstract class that is the interface of the Creator (AbstractShapeFactory.java)

```
public abstract class AbstractShapeFactory {  
    protected abstract Shape factoryMethod();  
    public Shape getShape() {  
        return factoryMethod();  
    }  
  
    // any other methods implemented here are written to operate on products produced by the factory method  
}
```

Demo walkthrough

- Step 4 - Create subclasses for each abstract factory that implements the factory method (RectangleFactory.java, SquareFactory.java, CircleFactory.java)

```
public class RectangleFactory extends AbstractShapeFactory
{
    private Shape factoryMethod()
    {
        return new Rectangle();
    }
}

// same for SquareFactory and CircleFactory
```

Demo walkthrough

- Step 5 - Use the Factory to get object of concrete class by creating the instance type of the correct factory (FactoryPatternDemo.java)

```
public class FactoryPatternDemo {  
    public static void main(String[] args) {  
        //get an object of Circle and call its draw method.  
        Shape shape1 = new CircleFactory.getShape();  
  
        //call draw method of Circle  
        shape1.draw();  
  
        //get an object of Rectangle and call its draw method.  
        Shape shape2 = new RectangleFactory.getShape();  
  
        //call draw method of Rectangle  
        shape2.draw();  
  
        //get an object of Square and call its draw method.  
        Shape shape3 = new SquareFactory.getShape();  
  
        //call draw method of square  
        shape3.draw();  
    }  
}
```

Demo walkthrough

- Step 6 - Verify the output.

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Another Example

- consider a framework for applications that can present multiple documents to the user
- two key abstractions in this framework are the classes Application (Creator) and Document (Product)
 - both classes are abstract
 - clients have to subclass them to realize their application-specific implementations
- if we create a drawing application
 - we define the classes DrawingApplication and DrawingDocument
- the Application class (Creator) is responsible for managing Documents (Product) and will create them as required
 - only knows when a new document should be created, not what kind of Document to create
 - the framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate
- the Factory Method pattern will encapsulate the knowledge of which Document subclass to create and moves this knowledge out of the framework
- application subclasses redefine an abstract CreateDocument operation on Application to return the appropriate Document subclass
- once an Application subclass is instantiated, it can then instantiate application-specific Documents without knowing their class
- we can call CreateDocument a factory method because it is responsible for “manufacturing” an object

Factory Method Implementation (Concrete Creator)

Jason Fedin

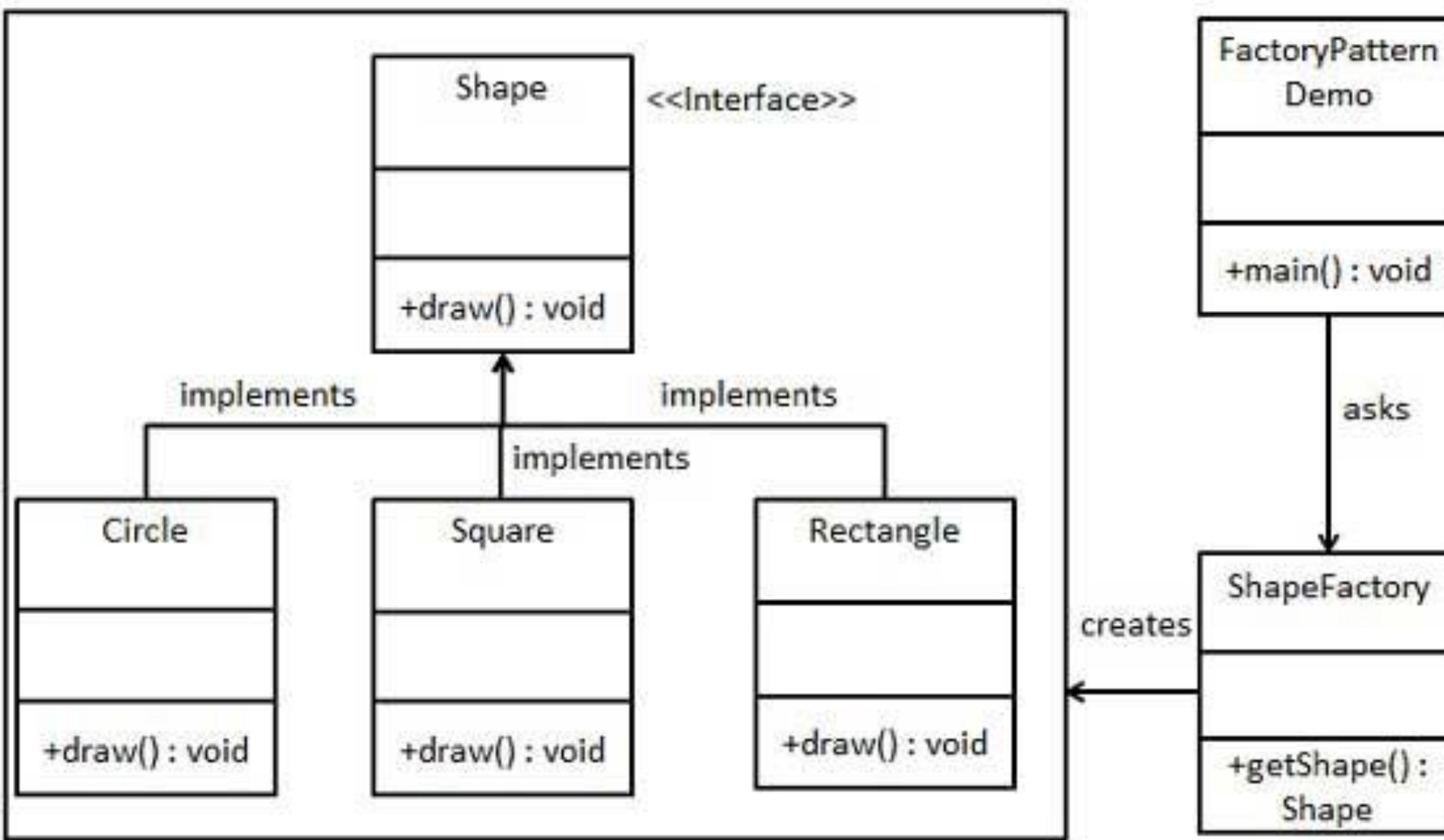
Most common implementation

- the second implementation of the factory method includes creating a single concrete creator class
 - the creator class is concrete class
 - you add implementation code to one factory method to create your product type based on a parameter passed to the method
 - to use the factory method (create objects), you create an instance of the creator class and invoke the factory method with an argument for your “class type”
- the advantage of this approach is that you do not need to create a new subclass of the abstract creator class and implement a new factory method

Overview

- we are going to create a Shape interface and concrete classes that implement this Shape interface
- we are then going to create a class named ShapeFactory which will implement our factory method
- lastly, we then create a FactoryPatternDemo class
 - will use ShapeFactory to get a Shape object
 - it will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs

Class Diagram



Demo walkthrough

- Step 1 - Create an interface (Shape.java)

```
public interface Shape {  
    void draw();  
}
```

- Step 2 - Create concrete classes implementing the same interface (Rectangle.java, Square.java, Circle.java)

```
public class Rectangle implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Inside Rectangle::draw() method.");  
    }  
}
```

Demo walkthrough

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Square::draw() method.");  
    }  
}
```

```
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Circle::draw() method.");  
    }  
}
```

Demo walkthrough

- Step 3 - Create a Factory to generate object of concrete class based on given information. (ShapeFactory.java)

```
public class ShapeFactory {  
    //use getShape method to get object of type shape  
    public Shape getShape(String shapeType){  
        if(shapeType == null){  
            return null;  
        }  
        if(shapeType.equalsIgnoreCase("CIRCLE")){  
            return new Circle();  
  
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
            return new Rectangle();  
  
        } else if(shapeType.equalsIgnoreCase("SQUARE")){  
            return new Square();  
        }  
        return null;  
    }  
}
```

Demo walkthrough

- Step 4 - Use the Factory to get object of concrete class by passing an information such as type. (FactoryPatternDemo.java)

```
public class FactoryPatternDemo {  
  
    public static void main(String[] args) {  
        ShapeFactory shapeFactory = new ShapeFactory();  
  
        //get an object of Circle and call its draw method.  
        Shape shape1 = shapeFactory.getShape("CIRCLE");  
  
        //call draw method of Circle  
        shape1.draw();  
  
        //get an object of Rectangle and call its draw method.  
        Shape shape2 = shapeFactory.getShape("RECTANGLE");  
  
        //call draw method of Rectangle  
        shape2.draw();  
  
        //get an object of Square and call its draw method.  
        Shape shape3 = shapeFactory.getShape("SQUARE");  
  
        //call draw method of square  
        shape3.draw();  
    }  
}
```

Demo walkthrough

- Step 5 - Verify the output.

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Factory Method Implementation (Static Method)

Jason Fedin

Static method implementation

- the third implementation of the factory method pattern includes the use of a static method
- define a factory as a static method is a common technique
 - often called a static factory
- this technique is sometimes used so that you do not need to instantiate an object to make use of the create method
- this technique has the disadvantage that you cannot subclass and change the behavior of the create method

Overview

- we are going to create a Shape interface and concrete classes that implement this Shape interface
- we are then going to create a class named ShapeFactory which will implement our factory method that is static
- lastly, we then create a FactoryPatternDemo class
 - will use ShapeFactory to get a Shape object
 - it will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs

Demo walkthrough

- Step 1 - Create an interface (Shape.java)

```
public interface Shape {  
    void draw();  
}
```

- Step 2 - Create concrete classes implementing the same interface (Rectangle.java, Square.java, Circle.java)

```
public class Rectangle implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Inside Rectangle::draw() method.");  
    }  
}
```

Demo walkthrough

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Square::draw() method.");  
    }  
}  
  
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Circle::draw() method.");  
    }  
}
```

Demo walkthrough

- Step 3 - Create a Factory to generate object of concrete class based on given information. (ShapeFactory.java)

```
public class ShapeFactory {  
    //use getShape method to get object of type shape  
    public static Shape getShape(String shapeType){  
        if(shapeType == null){  
            return null;  
        }  
        if(shapeType.equalsIgnoreCase("CIRCLE")){  
            return new Circle();  
  
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
            return new Rectangle();  
  
        } else if(shapeType.equalsIgnoreCase("SQUARE")){  
            return new Square();  
        }  
        return null;  
    }  
}
```

Demo walkthrough

- Step 4 - Use the Factory to get object of concrete class by passing an information such as type. (FactoryPatternDemo.java)

```
public class FactoryPatternDemo {  
  
    public static void main(String[] args) {  
        //get an object of Circle and call its draw method.  
        Shape shape1 = ShapeFactory.getShape("CIRCLE");  
  
        //call draw method of Circle  
        shape1.draw();  
  
        //get an object of Rectangle and call its draw method.  
        Shape shape2 = ShapeFactory.getShape("RECTANGLE");  
  
        //call draw method of Rectangle  
        shape2.draw();  
  
        //get an object of Square and call its draw method.  
        Shape shape3 = ShapeFactory.getShape("SQUARE");  
  
        //call draw method of square  
        shape3.draw();  
    }  
}
```

Demo walkthrough

- Step 5 - Verify the output.

Inside Circle::draw() method.

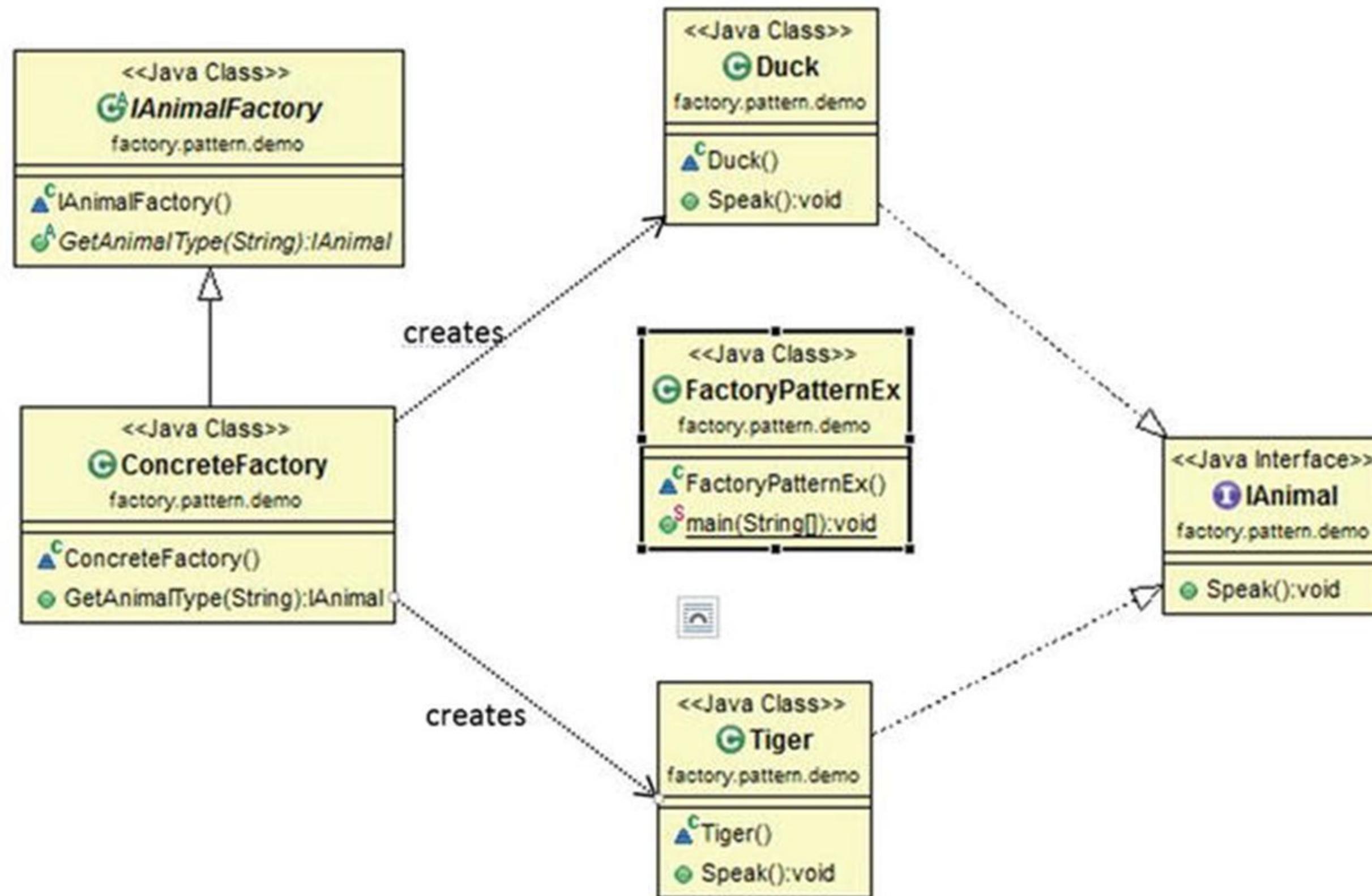
Inside Rectangle::draw() method.

Inside Square::draw() method.

(Demonstration) Applying the Factory Method Design Pattern

Jason Fedin

Overview



Code

```
interface IAnimal
{
    void Speak();
}

class Duck implements IAnimal
{
    @Override
    public void Speak()
    {
        System.out.println("Duck says Pack-pack");
    }
}

class Tiger implements IAnimal
{
    @Override
    public void Speak()
    {
        System.out.println("Tiger says: Halum..Halum");
    }
}
```

Code

```
abstract class IAnimalFactory
{
    /*if we cannot instantiate in later stage, we'll throw exception*/
    public abstract IAnimal GetAnimalType(String type) throws Exception;
}

class ConcreteFactory extends IAnimalFactory
{
    @Override
    public IAnimal GetAnimalType(String type) throws Exception
    {
        switch (type)
        {
            case "Duck":
                return new Duck();
            case "Tiger":
                return new Tiger();
            default:
                throw new Exception( "Animal type : "+type+" cannot be instantiated");
        }
    }
}
```

Code

```
class FactoryPatternDemo
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("/**Factory Pattern Demo***\n");
        IAnimalFactory animalFactory = new ConcreteFactory();
        IAnimal DuckType=animalFactory.GetAnimalType("Duck");
        DuckType.Speak();

        IAnimal TigerType = animalFactory.GetAnimalType("Tiger");
        TigerType.Speak();

        //There is no Lion type. So, an exception will be thrown
        IAnimal LionType = animalFactory.GetAnimalType("Lion");
        LionType.Speak();
    }
}
```

Abstract Factory

Jason Fedin

Overview

- the abstract factory provides an interface for creating families of related or dependent objects without specifying their concrete classes
 - “factory of factories”
 - super factory that creates other factories
- a pattern that creates objects via abstraction (does not care how its products are created)
- the methods of an Abstract Factory are implemented as factory methods
 - provides an encapsulation mechanism to a group of individual factories
 - factory method is a subset of this pattern
- there is often one concrete class implemented for each family

When to use this pattern?

- when a system should be independent of how its products are created, composed, and represented
- when we need to deal with multiple factories
- when the problem domain has different families of objects present and each family is used under different circumstances
- when a family of related product objects is designed to be used together, and you need to enforce this constraint
- when you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Advantages and Disadvantages

- isolates clients from concrete implementation classes
 - clients manipulate instances through their abstract interfaces
 - product class names are isolated in the implementation of the concrete factory; they do not appear in client code
- makes exchanging product families easy
 - the class of a concrete factory appears only once in an application (where it is instantiated)
 - makes it easy to change the concrete factory an application uses
- can support a complete family of products
 - enforces the use of products only from one family
- promotes consistency among products
- supporting new kinds of products is difficult
 - we need to extend the interface and, as a result, changes will be required in all of the subclasses that already implemented the interface

Relationship to factory method

- both encapsulate object creation to keep applications loosely coupled and less dependent on implementations
- the Abstract Factory delegates the responsibility of object instantiation to another object via composition
 - provides an abstract type for creating a family of products (get the “right” factory)
 - subclasses of this type define how those products are produced (the actual factory method)
 - interface has to change if new products are added
- the Factory Method uses inheritance and relies on a subclass to handle the desired object instantiation
 - usually implement code in the abstract creator that makes use of the concrete types the subclasses create
- use the abstract factory whenever you have families of products you need to create and you want to make sure your clients create products that belong together
- use the factory method when you want to decouple your client code from the concrete classes you need to instantiate
 - just subclass me and implement my factory method!

decorating room example (High Level Example)

- we need two different types of tables
 - one must be made of wood
 - one must be made of steel
- for the wooden table, we need to visit a carpenter shop
- for the other type, we can go to a readymade steel table shop
- both of these are table factories
- we decide what kind of factory we need

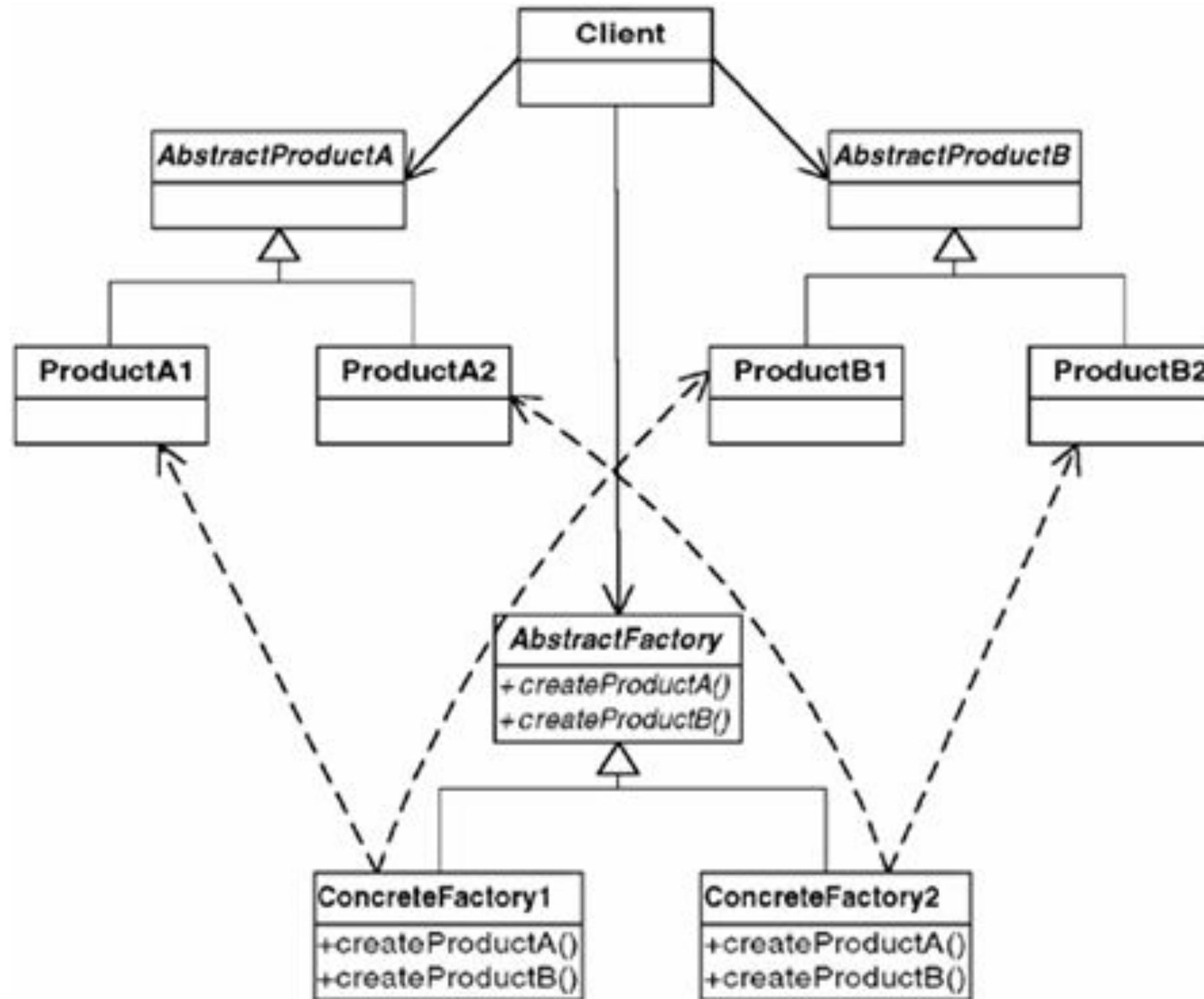
Summary

- all factories encapsulate object creation
- Factory Method relies on inheritance
 - object creation is delegated to subclasses, which implement the factory method to create objects
- Abstract Factory relies on object composition
 - object creation is implemented in methods exposed in the factory interface
- all factory patterns promote loose coupling by reducing the dependency of your application on concrete classes
- the intent of Factory Method is to allow a class to defer instantiation to its subclasses
- the intent of Abstract Factory is to create families of related objects without having to depend on their concrete classes

Abstract Factory Implementation

Jason Fedin

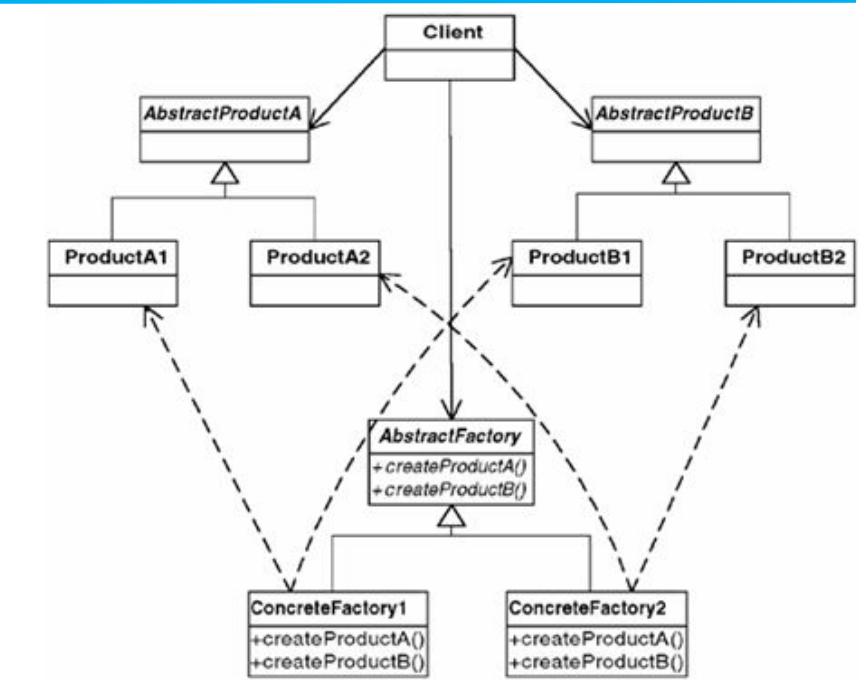
Class Diagram



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Class Diagram explanation

- a Client uses objects derived from two different Product classes
 - AbstractProductA and AbstractProductB
 - declares an interface for a type of product object
 - a design that simplifies, hides implementations, and makes a system more maintainable
- the Client object does not know which particular concrete implementations of the server objects it has because the factory object has the responsibility to create them
- the client object does not even know which particular factory it uses because it only knows that it has an AbstractFactory object
 - It has a ConcreteFactory1 or a ConcreteFactory2 object, but it doesn't know which one



Overview (Tips for Implementation)

- an application typically needs only one instance of a ConcreteFactory per product family
 - usually best implemented as a Singleton
- AbstractFactory only declares an interface for creating products
 - up to ConcreteProduct subclasses to actually create them (just like in the factory method)
 - most common way to do this is to define a factory method for each product
 - concrete factory will specify its products by overriding the factory method for each
 - requires a new concrete factory subclass for each product family, even if the product families differ only slightly
- If many product families are possible, the concrete factory can be implemented using the Prototype pattern
 - concrete factory is initialized with a prototypical instance of each product in the family, and it creates a new product by cloning its prototype
 - eliminates the need for a new concrete factory class for each new product family

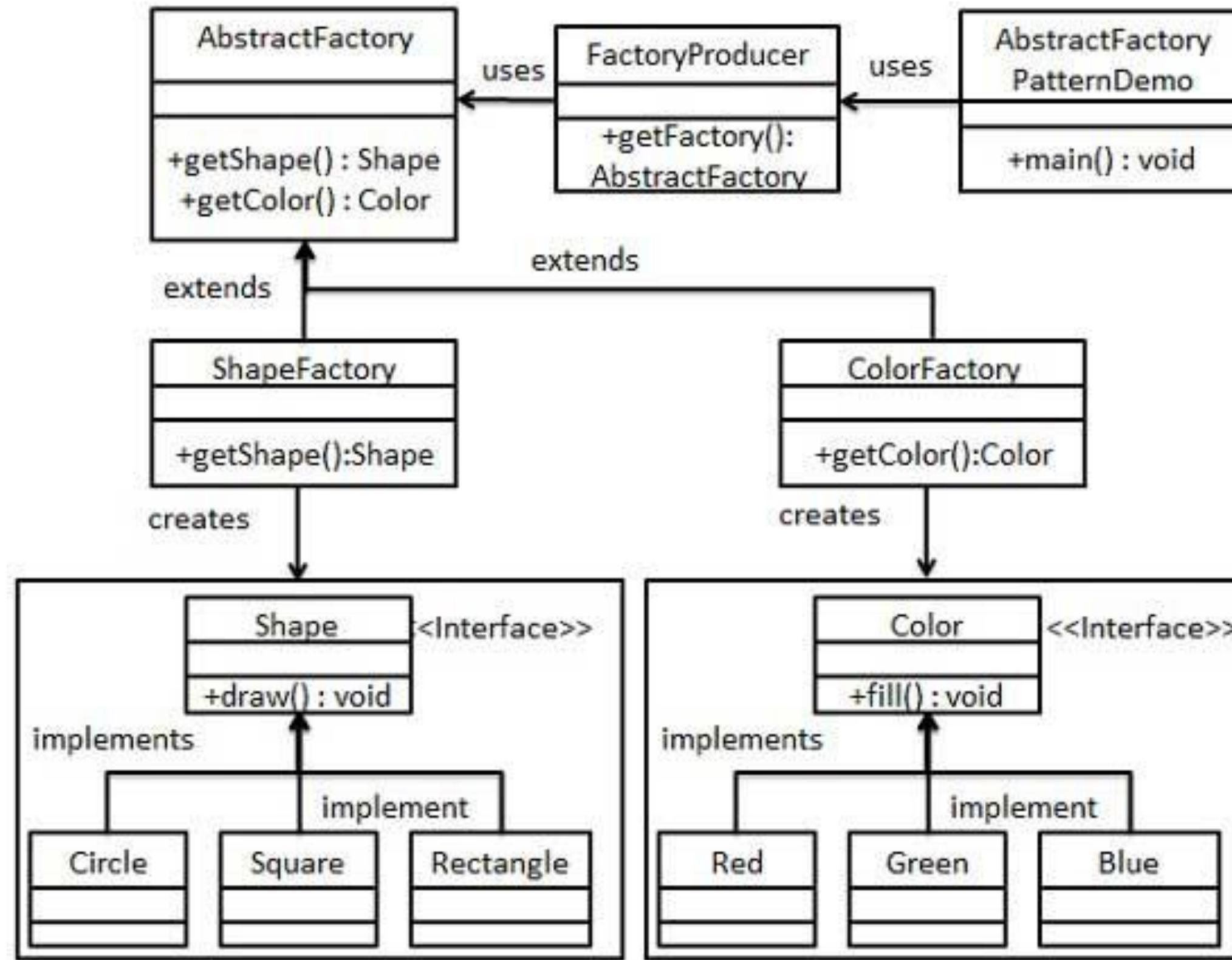
Overview (Tips for Implementation)

- AbstractFactory usually defines a different method for each kind of product it can produce
 - adding a new kind of product requires changing the AbstractFactory interface and all the classes that depend on it
- a more flexible design is to add a parameter to methods that create objects
 - parameter specifies the kind of object to be create
 - parameter could be a class identifier, an integer, a string, or anything else that identifies the kind of product
 - needs a single “Make” operation with a parameter indicating the kind of object to create

Example Overview

- we are going to create a Shape interface and concrete classes that implement this Shape interface (Products)
 - this is our first “family” of products
- we are going to create a Color interface and concrete classes that implement this Color interface (Products)
 - this is our second “family” of products
- we are then going to create an abstract factory class named AbstractFactory and subclasses that extend this AbstractFactory
 - this abstract factory class will return a particular family of related products (Shapes or Colors)
 - this abstract factory class contains factory methods for each product family
 - each subclass will implement its factory method for its family of related products
- we will also need to create a FactoryProducer class which will return the correct factory for each family of related products
 - shape factory or color factory
 - will utilize a static method to return the correct factory (could have required an instance)
- lastly, we then create a FactoryPatternDemo class
 - this class will use object composition (FactoryProducer) to get the “correct” AbstractFactory (family of products grouped together)
- all factory methods will include parameters which will determine the correct object instance to create (could have delegated to a new subclass)

Class Diagram (Our Example)



Demo walkthrough

- Step 1 - Create an interface (Shape.java)

```
public interface Shape {  
    void draw();  
}
```

- Step 2 - Create concrete classes implementing the same interface (Rectangle.java, Square.java, Circle.java)

```
public class Rectangle implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Inside Rectangle::draw() method.");  
    }  
}
```

Demo walkthrough

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Square::draw() method.");  
    }  
}  
  
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Circle::draw() method.");  
    }  
}
```

Demo walkthrough

Step 3 - Create an interface for Colors (Color.java)

```
public interface Color {  
    void fill();  
}
```

Step 4 - Create concrete classes implementing the same interface (Red.java, Green.java, Blue.java)

```
public class Red implements Color {  
  
    @Override  
    public void fill() {  
        System.out.println("Inside Red::fill() method.");  
    }  
}
```

Demo walkthrough

```
public class Green implements Color {  
  
    @Override  
    public void fill() {  
        System.out.println("Inside Green::fill() method.");  
    }  
}
```

```
public class Blue implements Color {  
  
    @Override  
    public void fill() {  
        System.out.println("Inside Blue::fill() method.");  
    }  
}
```

Demo walkthrough

Step 5 - Create an Abstract class to get factories for Color and Shape Objects (AbstractFactory.java)

```
public abstract class AbstractFactory {  
    abstract Color getColor(String color); // NOT A GOOD INTERFACE, SHOULD BE MORE COHESIVE  
    abstract Shape getShape(String shape);  
}
```

Step 6 - Create Factory classes extending AbstractFactory to generate object of concrete class based on given information (ShapeFactory.java, ColorFactory.java)

```
public class ShapeFactory extends AbstractFactory {  
  
    @Override  
    public Shape getShape(String shapeType){  
  
        if(shapeType == null){  
            return null;  
        }  
    }  
}
```

Demo walkthrough (ShapeFactory.java)

```
if(shapeType.equalsIgnoreCase("CIRCLE")){
    return new Circle();
}

}else if(shapeType.equalsIgnoreCase("RECTANGLE")){
    return new Rectangle();
}

}else if(shapeType.equalsIgnoreCase("SQUARE")){
    return new Square();
}

}

return null;
}

@Override
Color getColor(String color) {
    return null;          // BADDADDADDADDADDADDADDADDADDADDADDADD!!!!!!!
}

}
```

Demo walkthrough (ColorFactory.java)

Demo walkthrough (ColorFactory.java)

```
 }else if(color.equalsIgnoreCase("GREEN")){
    return new Green();

 }else if(color.equalsIgnoreCase("BLUE")){
    return new Blue();
}

return null;
}

}
```

Demo walkthrough

Step 7 - Create a Factory generator/producer class to get factories by passing an information such as Shape or Color
(FactoryProducer.java)

```
public class FactoryProducer {  
    public static AbstractFactory getFactory(String choice){  
  
        if(choice.equalsIgnoreCase("SHAPE")){  
            return new ShapeFactory();  
  
        }else if(choice.equalsIgnoreCase("COLOR")){  
            return new ColorFactory();  
        }  
  
        return null;  
    }  
}
```

Demo walkthrough

Step 8 - Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type (AbstractFactoryPatternDemo.java)

```
public class AbstractFactoryPatternDemo {  
    public static void main(String[] args) {  
  
        //get shape factory  
        AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");  
  
        //get an object of Shape Circle  
        Shape shape1 = shapeFactory.getShape("CIRCLE");  
  
        //call draw method of Shape Circle  
        shape1.draw();  
  
        //get an object of Shape Rectangle  
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
```

Demo walkthrough (AbstractFactoryPatternDemo.java)

```
//call draw method of Shape Rectangle  
shape2.draw();  
  
//get an object of Shape Square  
Shape shape3 = shapeFactory.getShape("SQUARE");  
//call draw method of Shape Square  
shape3.draw();  
  
//get color factory  
AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");  
  
//get an object of Color Red  
Color color1 = colorFactory.getColor("RED");  
  
//call fill method of Red  
color1.fill();
```

Demo walkthrough (AbstractFactoryPatternDemo.java)

```
//get an object of Color Green  
Color color2 = colorFactory.getColor("Green");  
  
//call fill method of Green  
color2.fill();  
  
//get an object of Color Blue  
Color color3 = colorFactory.getColor("BLUE");  
  
//call fill method of Color Blue  
color3.fill();  
}  
}
```

Demo walkthrough

Step 9 Verify the output

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Inside Red::fill() method.

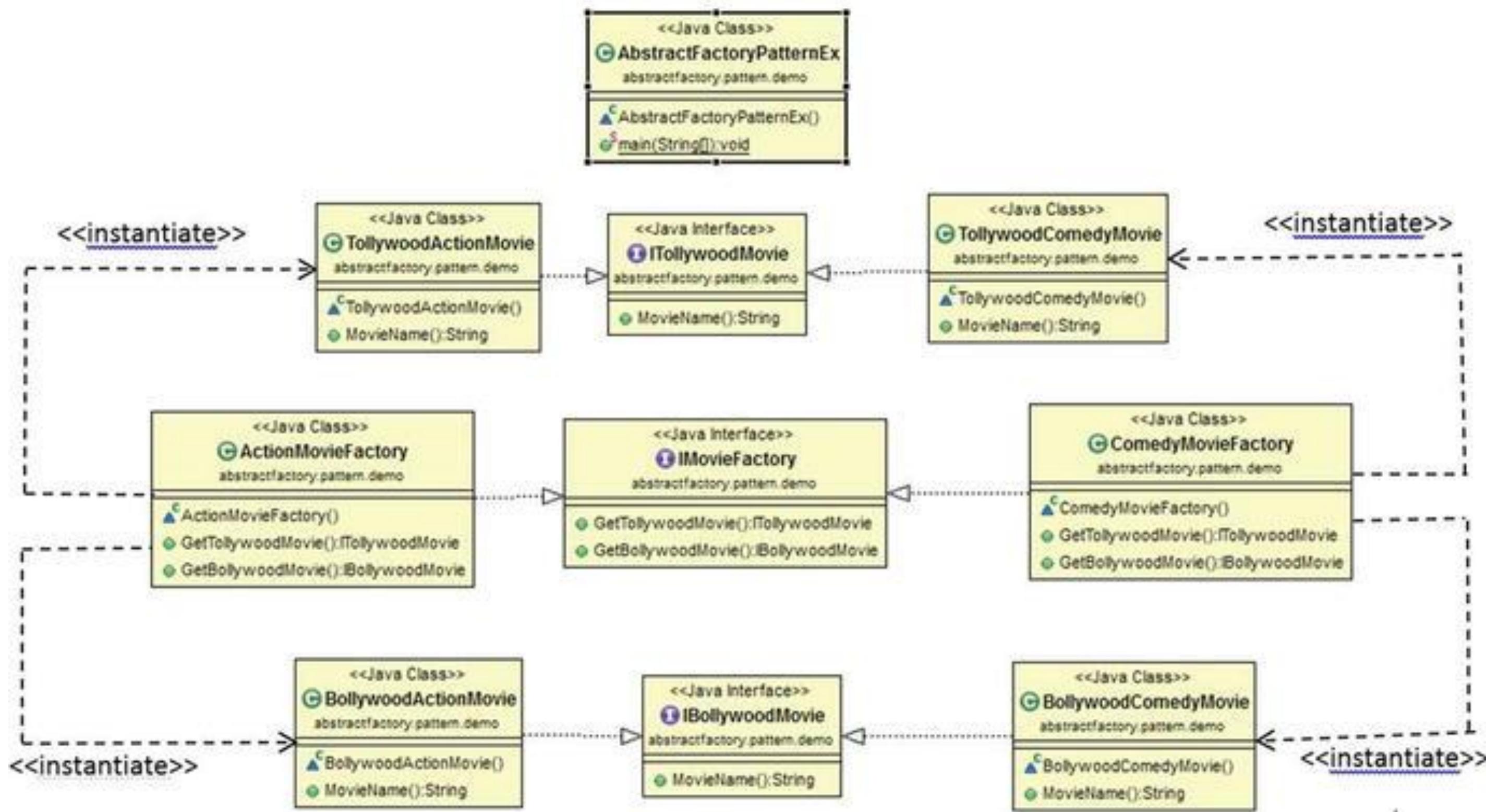
Inside Green::fill() method.

Inside Blue::fill() method.

(Demonstration) Abstract Factory Method Design pattern

Jason Fedin

Overview



Code

```
interface IHollywoodMovie
{
    String getMovieName();
}
```

```
interface IBollywoodMovie
{
    String getMovieName();
}
```

Code

```
class HollywoodActionMovie implements IHollywoodMovie
{
    @Override
    public String getMovieName()
    {
        return "True Lies is a Hollywood Action Movie";
    }

}

class HollywoodComedyMovie implements IHollywoodMovie
{
    @Override
    public String getMovieName()
    {
        return "The Jerk is a Hollywood Comedy Movie";
    }
}
```

Code

```
class BollywoodActionMovie implements IBollywoodMovie
{
    @Override
    public String getMovieName()
    {
        return "Bang Bang is a Bollywood Action Movie";
    }
}

class BollywoodComedyMovie implements IBollywoodMovie
{
    @Override
    public String getMovieName()
    {
        return "Munna Bhai MBBS is a Bollywood Comedy Movie";
    }
}
```

Code

```
interface IMovieFactory
{
    IHollywoodMovie GetHollywoodMovie();
    IBollywoodMovie GetBollywoodMovie();
}

//Action Movie Factory
class ActionMovieFactory implements IMovieFactory
{
    public IHollywoodMovie GetHollywoodMovie()
    {
        return new HollywoodActionMovie();
    }

    public IBollywoodMovie GetBollywoodMovie()
    {
        return new BollywoodActionMovie();
    }
}
```

Code

```
//Comedy Movie Factory
class ComedyMovieFactory implements IMovieFactory
{
    public IHollywoodMovie GetHollywoodMovie()
    {
        return new HollywoodComedyMovie();
    }

    public IBollywoodMovie GetBollywoodMovie()
    {
        return new BollywoodComedyMovie();
    }
}
```

Code

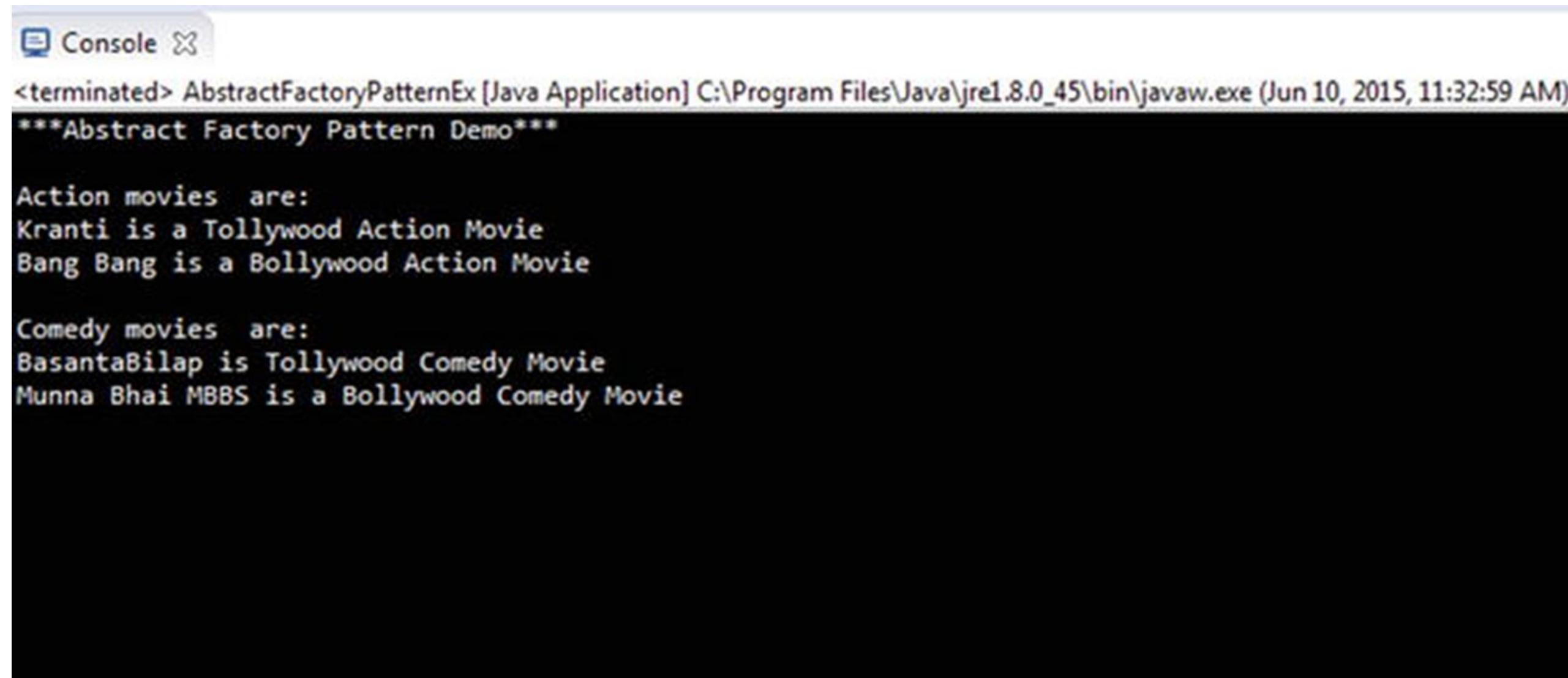
```
class AbstractFactoryPatternEx
{
    IMovieFactory actionMovies = new ActionMovieFactory();
    IHollywoodMovie hAction = actionMovies.GetHollywoodMovie();
    IBollywoodMovie bAction = actionMovies.GetBollywoodMovie();

    System.out.println("\nAction movies are:");
    System.out.println(hAction.getMovieName());
    System.out.println(bAction.getMovieName());

    IMovieFactory comedyMovies = new ComedyMovieFactory();
    IHollywoodMovie hComedy = comedyMovies.GetHollywoodMovie();
    IBollywoodMovie bComedy = comedyMovies.GetBollywoodMovie();

    System.out.println("\nComedy movies are:");
    System.out.println(hComedy.getMovieName());
    System.out.println(bComedy.getMovieName());
}
```

Output



The screenshot shows a Java application console window titled "Console". The output text is:
<terminated> AbstractFactoryPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Jun 10, 2015, 11:32:59 AM)
Abstract Factory Pattern Demo

Action movies are:
Kranti is a Tollywood Action Movie
Bang Bang is a Bollywood Action Movie

Comedy movies are:
BasantaBilap is Tollywood Comedy Movie
Munna Bhai MBBS is a Bollywood Comedy Movie

Another implementation

- different methods for each product
 - GetMovieName for Hollywood movies vs. GetMovieActors for Bollywood movies
- HollywoodMovieFactory and BollyWoodMovieFactory
 - as opposed to factories organized around comedy and action
 - each factory method returns null for other method not relevant
 - each factory method takes a parameter to determine object creation (“action” vs. “comedy”)
- FactoryProducer class which has a static method to determine which factory is created
 - HollywoodFactory or BollyWoodFactory

Code

```
interface IHollywoodMovie
{
    String getMovieName();
}
```

```
interface IBollywoodMovie
{
    String getMovieActors();
}
```

Code

```
class HollywoodActionMovie implements IHollywoodMovie
{
    @Override
    public String getMovieName()
    {
        return "True Lies is a Hollywood Action Movie";
    }

}

class HollywoodComedyMovie implements IHollywoodMovie
{
    @Override
    public String getMovieName()
    {
        return "The Jerk is a Hollywood Comedy Movie";
    }
}
```

Code

```
class BollywoodActionMovie implements IBollywoodMovie
{
    @Override
    public String getMovieActors()
    {
        return "whatever";
    }
}
```

```
class BollywoodComedyMovie implements IBollywoodMovie
{
    @Override
    public String getMovieActors()
    {
        return "whatever";
    }
}
```

Code

Code

Code

```
public class FactoryProducer {  
    public static AbstractFactory getFactory(String choice){  
  
        if(choice.equalsIgnoreCase("HollyWoodMovie")){  
            return new HollyWoodMovieFactory();  
  
        }else if(choice.equalsIgnoreCase("BollyWoodMovie")){  
            return new BollyWoodMovieFactory();  
        }  
  
        return null;  
    }  
}
```

Code

```
class AbstractFactoryPatternEx
{
    IMovieFactory hollywoodMovieFactory = FactoryProducer.getFactory("HollyWoodMovie");

    IHollywoodMovie hAction = hollywoodMovieFactory.GetHollywoodMovie("action");
    System.out.println(hAction.getMovieName());

    IHollywoodMovie hComedy = hollywoodMovieFactory.GetHollywoodMovie("comedy");
    System.out.println(hComedy.getMovieName());

    IMovieFactory bollywoodMovieFactory = FactoryProducer.getFactory("BollyWoodMovie");

    IBollywoodMovie bAction = bollywoodMovieFactory.GetBollywoodMovie("action");
    System.out.println(bAction.getMovieActors());

    IBollywoodMovie bComedy = bollywoodMovieFactory.GetBollywoodMovie("comedy");
    System.out.println(bComedy.getMovieActors());
}
```

Singleton Overview

Jason Fedin

Overview

- the Singleton pattern is one of the simplest design patterns in Java
 - comes under creational pattern as this pattern provides one of the best ways to create an object
- ensures a class only has one instance, and provide a global point of access to it
- we are taking a class and letting it manage a single instance of itself
 - also preventing any other class from creating a new instance on its own
 - to get an instance, you have got to go through the class itself
- we are also providing a global access point to the instance
 - whenever you need an instance, just query the class and it will hand you back the single instance
 - a global variable makes an object accessible, but it does not keep you from instantiating multiple objects
- it is important for some classes to have exactly one instance

Examples

- although there can be many printers in a system, there should be only one printer spooler
- there should be only one file system and one window manager
- an accounting system will be dedicated to serving one company
- logging, drivers objects, caching, and thread pool would also use a singleton
- the singleton design pattern is used in core java classes also, for example `java.lang.Runtime`, `java.awt.Desktop`

Advantages/Disadvantages of the singleton

- controlled access to sole instance
 - because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it
- reduced name space
 - an improvement over global variables
 - avoids polluting the name space with global variables that store sole instances
- permits a variable number of instances
 - makes it easy to change your mind and allow more than one instance of the Singleton class
- singletons hinder unit testing
 - might cause issues for writing testable code if the object and the methods associated with it are so tightly coupled that it becomes impossible to test without writing a fully-functional class dedicated to the Singleton
- singletons create hidden dependencies
 - because it is readily available throughout the code base, it can be overused
 - since its reference is not completely transparent while passing to different methods, it becomes difficult to track

Overview of implementation

- the Singleton pattern works by having a special method that is used to instantiate the desired object
- when this method is called, it checks to see whether the object has already been instantiated
 - If it has, the method just returns a reference to the object
 - If not, the method instantiates it and returns a reference to the new instance
- to ensure that this is the only way to instantiate an object of this type, you should define the constructor of this class to be protected or private

Summary

- use the Singleton pattern when
- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point

Singleton vs. Dependency Injection

Jason Fedin

Overview

- there is some discussion in the software community on when you should utilize a singleton and when you should utilize dependency injection
- we know dependency injection is a technique whereby one object supplies the dependencies of another object
 - enables you to replace dependencies without changing the class that uses them
- dependency injection can also be used to avoid statics (one of the most common reasons to use it)
- we know that singletons ensure only one instance of an object
- using DI, you can use constructor or setter injection to pass around a single object
 - have the injector create a single object and then inject it via the constructor or setter of any dependent objects
 - implements the singleton with less dependencies

Singleton dependencies

- we know that there are some disadvantages of utilizing the singleton pattern
- singletons create hidden dependencies
 - because it is readily available throughout the code base, it can be overused
 - since its reference is not completely transparent while passing to different methods, it becomes difficult to track
- singletons hinder unit testing
 - might cause issues for writing testable code if the object and the methods associated with it are so tightly coupled that it becomes impossible to test without writing a fully-functional class dedicated to the Singleton
 - use of static makes hard to test
- so, the question becomes, can dependency injection solve the disadvantage of the Singleton pattern's introduction of new dependencies?
 - can DI reduce/eliminate these dependencies?
- the answer is “it depends on the situation”, in some cases, dependency injection is preferred over singletons

when to use DI over the Singleton

- when you want your software to be under unit test
 - it is much easier to write unit tests for your code when using DI (less coupling)
- when you want to avoid using statics
 - statics make code harder to test
 - singletons use statics
- when you have a non-stable dependency
 - a non-stable dependency is a dependency which refers to or affects the global state, such as an external service, file system, or a database
 - it is good practice to inject that dependency to the dependent class
 - helps the class explicitly specify everything it needs in order to perform properly
- using DI over Singletons is mostly, but not entirely, about testing
- sometimes you do not want a singleton, just more flexible code

when to use a singleton over DI

- there still are dependencies which are better represented using a Singleton
- ambient dependencies are dependencies which span across multiple classes and often multiple layers
 - the singleton pattern is better suited to handle this
 - you do not want to pass the injector object to all of these multiple classes
- a good example of when to use a singleton over DI is a logger service
 - if you tend to log a lot of activities throughout your code base, it is just not practical to pass the logger instance to every class that needs it as a dependency
- It is important to keep a balance between the dependencies represented as Singletons and the ones injected using the DI principles
- if a dependency is ambient, meaning that it is used by many classes and/or multiple layers, use a Singleton
- otherwise, inject it to the dependent classes using the Dependency Injection principle

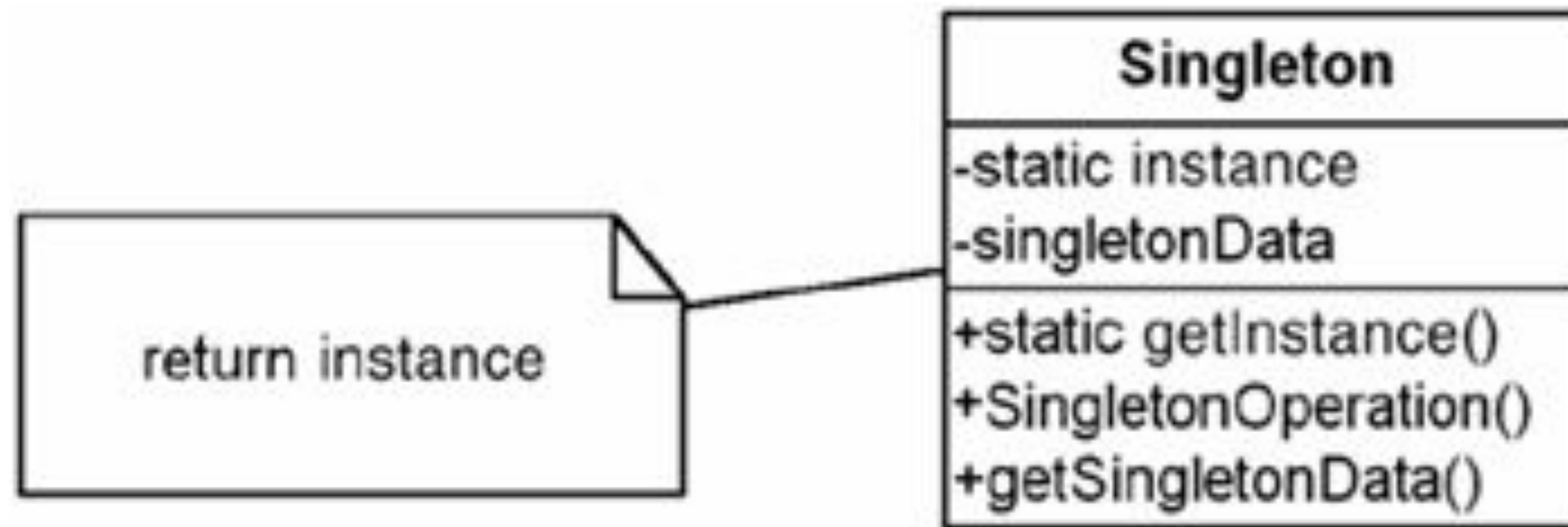
Singleton Implementation (Overview)

Jason Fedin

Overview

- to implement the Singleton pattern, there are different approaches but all of them have the following common concepts
 - private constructor to restrict instantiation of the class from other classes
 - private static variable of the same class that is the only instance of the class
 - public static method that returns the instance of the class
 - is the global access point for outer world to get the instance of the singleton class

Class Diagram



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Approaches

- I will discuss 5 main approaches when implementing the singleton pattern
- Lazy evaluation approach
 - it is not multi-thread safe
 - use this approach if you are not worried about multiple threads
 - this is not a recommended approach
- Synchronized approach
 - thread safe
 - use when performance is not critical to your application, but, it is multi-threaded
 - straightforward and effective

Approaches (cont'd)

- double-checked locking principle approach
 - thread safe
 - increases performance from the synchronized approach
- eager evaluation approach
 - if your application always creates and uses an instance of the Singleton
 - does not use a lot of resources
 - thread safe
 - the instance is created even though client application might not be using it

Approaches (cont'd)

- Bill Pugh approach
 - thread safe
 - high performance
 - ensures that the instance is only created if a client needs it
 - create the Singleton class using a inner static helper class
 - regarded as the standard method to implement singletons in Java

Singleton Implementation (Lazy Evaluation Approach)

Jason Fedin

Singleton with Lazy Initialization (Demo using intellij)

- the first implementation technique we will discuss is to use lazy initialization
 - this method implements the Singleton pattern by creating the instance in a global access method

```
public class Singleton {  
  
    // the private reference to the one and only instance  
    private static Singleton uniqueInstance = null;  
  
    // An instance attribute  
    private int data = 0;  
  
    /**  
     * The Singleton Constructor  
     * Note that it is private!  
     * No client can instantiate a Singleton object!  
     */  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
  
        return uniqueInstance;  
    }  
  
    // add a setData here  
}
```

Singleton with Lazy Initialization

```
public class TestSingleton {  
  
    public static void main(String args[]) {  
        // Get a reference to the single instance of Singleton.  
        Singleton s = Singleton.instance();  
  
        // Set the data value.  
        s.setData(34);  
        System.out.println("First reference: " + s);  
        System.out.println("Singleton data value is: " + s.getData());  
  
        // Get another reference to the Singleton.  
        // Is it the same object?  
        s = null;  
        s = Singleton.instance();  
        System.out.println("\nSecond reference: " + s);  
        System.out.println("Singleton data value is: " + s.getData());  
    }  
}
```

Singleton with Lazy Initialization (output)

First reference: Singleton@1cc810

Singleton data value is: 34

Second reference: Singleton@1cc810

Singleton data value is: 34

Problems with Lazy Initialization approach

- the implementation on the previous slides are not thread safe
- suppose two calls to getInstance() are made at virtually the same time
- the first thread checks to see whether the instance exists. It does not, it goes into the part of the code that will create the first instance
- however, before it has done that, suppose a second thread also looks to see whether the instance member is null
 - because the first thread has not created anything yet, the instance is still equal to null, so the second thread also goes into the code that will create an object
- both threads now perform a new on the Singleton object, thereby creating two objects

Problems with Lazy Initialization approach (cont'd)

- if the Singleton is absolutely stateless, then thread safety may not be a problem
- If the Singleton has state, and if you expect that when one object changes the state, all other objects should see the change, then this could become a serious problem
 - the first thread will be interacting with a different object than all other threads do
- inconsistent state between threads using the different Singleton objects
- if the object creates a connection, there will actually be two connections (one for each object)
- if a counter is used, there will be two counters
- it may be very difficult to find these problems
 - dual creation is very intermittent - it usually won't happen
 - it may not be obvious why the counts are off, because only one client object will contain one of the Singleton objects while all the other client objects will refer to the other Singleton

Singleton Implementation (Thread Safe)

Jason Fedin

Singleton thread safe (synchronized approach)

- the easiest way to create a thread-safe singleton class is to make the global access method synchronized
 - only one thread can execute this method at a time

```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    private Singleton() {}  
  
    // by adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the  
    // method  
    public static synchronized Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
}
```

problem with synchronized approach

- one big problem is that the synchronization may end up being a severe bottleneck
 - all the threads will have to wait for the check on whether the object already exists
 - reduces the performance because of cost associated with the synchronized method
- the only time synchronization is relevant is the first time through this method
 - once we have set the uniqueInstance variable to an instance of Singleton, we have no further need to synchronize this method
 - after the first time through, synchronization is totally unneeded overhead
- for most Java applications, we need to ensure that the Singleton works in the presence of multiple threads and does not have performance issues
 - use the double checked locking principle

Singleton Implementation (Double Checked Locking principle)

Jason Fedin

Double Checked Locking principle

- this approach will use a synchronized block inside the if condition with an additional check to ensure that only one instance of the singleton class is created
 - intent is to optimize away unnecessary locking, increase performance
 - the synchronization check happens at most one time, so it will not be a bottleneck
- use “double-checked locking” to reduce the use of synchronization in getInstance()
- with double-checked locking, we first check to see if an instance is created, and if not, THEN we synchronize
 - we only synchronize the first time through, just what we want

double checked locking example

```
public class Singleton {  
    private volatile static Singleton uniqueInstance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
  
        // check for instance and if there isn't one, enter a synchronized block  
        If (uniqueInstance == null) {  
            synchronized (Singleton.class) {  
                // we only synchronize the first time through  
                if (uniqueInstance == null) {  
                    uniqueInstance = new Singleton();  
                }  
            }  
        }  
    }  
}
```

double checked locking example explained

- the volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance
- if performance is an issue in your use of the getInstance() method then this method of implementing the Singleton can drastically reduce the overhead.
- double-checked locking does not work in Java 1.4 or earlier!
- unfortunately, in Java version 1.4 and earlier, many JVMs contain implementations of the volatile keyword that allow improper synchronization for double-checked locking
 - If you must use a JVM earlier than Java 5, consider other methods of implementing your Singleton
- again, double-checked locking provides for unnecessary locking and is avoided by adding another test before creating the object
- it supports multithreaded environments

Singleton Implementation (Eager Evaluation approach)

Jason Fedin

Singleton (eager Initialization approach)

- in eager initialization, the instance of Singleton Class is created at the time of class loading
 - the easiest method to create a singleton class
 - it has a drawback that the instance is created even though client application might not be using it
- using this approach, we rely on the JVM to create the unique instance of the Singleton when the class is loaded
 - the JVM guarantees that the instance will be created before any thread accesses the static uniqueInstance variable (threadsafe)

```
public class Singleton {  
    // create an instance of singleton in a static initializer, code is guaranteed to be thread safe  
    private static Singleton uniqueInstance = new Singleton();  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        // we already got an instance so just return it  
        return uniqueInstance;  
    }  
}
```

Singleton (eager Initialization) problems

- If your singleton class is not using a lot of resources, then the eager initialization is the approach to use
- in most of the scenarios, Singleton classes are created for resources such as File System, Database connections etc
 - we should avoid the instantiation until/unless client calls the getInstance method
- so, Bill Pugh came up with a different approach to create the Singleton class using a inner static helper class
- this is the most widely used approach for the Singleton class
 - does not require synchronization, is thread safe and only creates the instance when the client needs it
 - it is easy to understand and implement
 - regarded as the standard method to implement singletons in Java

Singleton Implementation (Bill Pugh Approach)

Jason Fedin

Overview

- in most of the scenarios, Singleton classes are created for resources such as File System, Database connections etc
 - we should avoid the instantiation until/unless client calls the getInstance method
- so, Bill Pugh came up with a different approach to create the Singleton class using a inner static helper class
- this is the most widely used approach for the Singleton class
 - does not require synchronization, is thread safe and only creates the instance when the client needs it
 - it is easy to understand and implement
 - regarded as the standard method to implement singletons in Java

Bill Pugh Singleton Implementation

```
public class SingleObject
{
    private SingleObject() {}

    private static class SingletonHelper {
        //Nested class is referenced after getInstance() is called
        private static final SingleObject INSTANCE = new SingleObject();
    }

    public static SingleObject getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

- notice the private inner static class that contains the instance of the singleton class
- when the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the getInstance method, this class gets loaded and creates the Singleton class instance

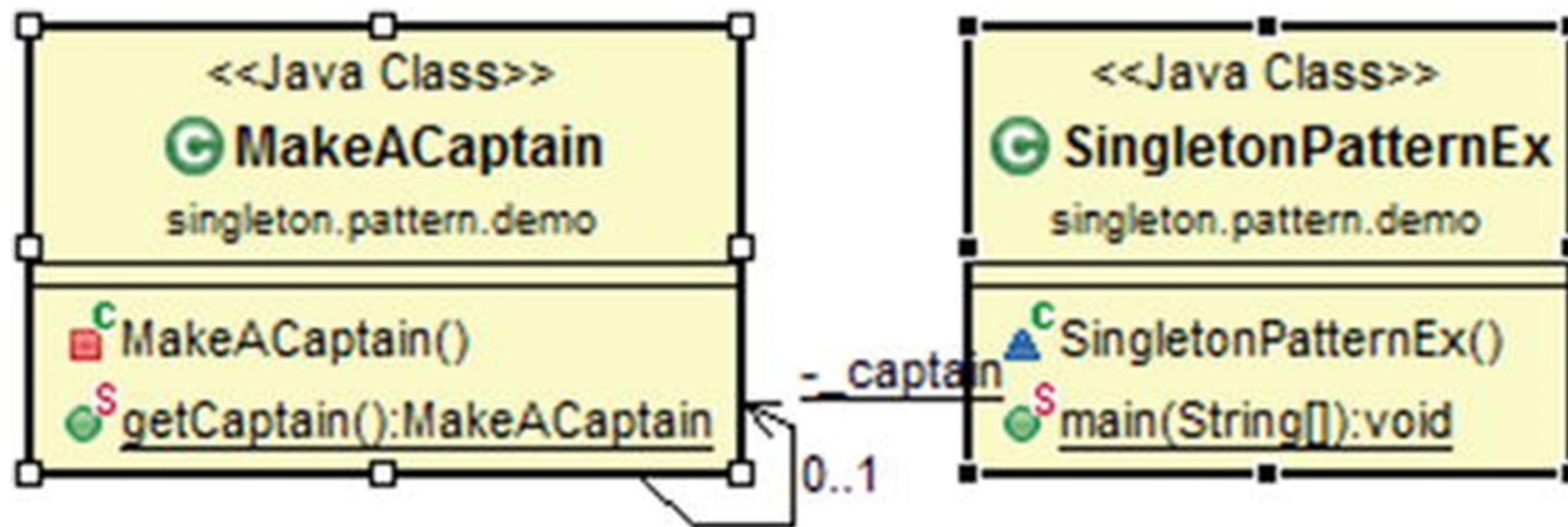
Summary

- use the lazy evaluation approach for creating a Singleton if you are not worried about multiple threads
 - not recommended
- for most Java applications, you will need ensure that the Singleton works in the presence of multiple threads
 - add the synchronized keyword to the getInstance() method if performance is not critical to your application
 - when calling the getInstance() method is not causing substantial overhead for your application
 - synchronizing getInstance() is straightforward and effective
 - just keep in mind that synchronizing a method can decrease performance by a factor of 100, so if a high-traffic part of your code begins using getInstance(), you may have to reconsider
 - Use the double-checked locking principle to increase performance in the getInstance() method
- If your application always creates and uses an instance of the Singleton and is not using a lot of resources
 - move to an eagerly created instance approach rather than a lazily created one
 - will be thread safe
 - has a drawback that the instance is created even though client application might not be using it
- so, the recommended approach is the Bill Pugh method
 - create the Singleton class using a inner static helper class

(Demonstration) Singleton

Jason Fedin

Class Diagram



code

```
class MakeACaptain
{
    private static MakeACaptain _captain;
    private MakeACaptain() { }

    //Bill Pugh solution

    private static class SingletonHelper{
        //Nested class is referenced after getCaptain() is called
        private static final MakeACaptain _captain = new MakeACaptain();
    }

    public static MakeACaptain getCaptain()
    {
        return SingletonHelper._captain;
    }
}
```

main() method

```
class SingletonPatternEx
{
    public static void main(String[] args)
    {
        System.out.println("***Singleton Pattern Demo***\n");
        System.out.println("Trying to make a captain for our team");
        MakeACaptain c1 = MakeACaptain.getCaptain();
        System.out.println("Trying to make another captain for our team");
        MakeACaptain c2 = MakeACaptain.getCaptain();
        if (c1 == c2)
        {
            System.out.println("c1 and c2 are same instance");
        }
    }
}
```

Output

```
Console X
<terminated> SingletonPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 9, 2015, 9:05:05 PM)
***Singleton Pattern Demo***

Trying to make a captain for our team
New Captain selected for our team
Trying to make another captain for our team
You already have a Captain for your team. Send him for the toss.
c1 and c2 are same instance
```

Builder Design Pattern

Jason Fedin

Overview

- the builder design pattern separates the construction of a complex object from its representation
 - uses the same construction processes to create the same object
 - however, these processes can create different representations of the object
 - uses simple objects and a step by step approach to create the object
 - the builder class is independent of other objects
- useful when creating the object is very complex and is independent of the assembly of the parts of the object
- an example would be creating your own computer
 - different parts are assembled depending upon the order received by the customer
 - a customer can demand a 500 GB hard disk with an Intel processor
 - another customer can choose a 250 GB hard disk with an AMD processor

Another Example - Vacation Planner for Disney World

- building a vacation planner for Disney World
 - guests can choose a hotel and various types of admission tickets
 - make restaurant reservations, and even book special events
- you will need a flexible design
 - each guest's planner can vary in the number of days and types of activities it includes
 - a local resident might not need a hotel, but wants to make dinner and special event reservations
 - another guest might be flying into Orlando and needs a hotel, dinner reservations, and admission tickets
- we encapsulate the creation of the trip planner in an object (a builder)
 - have our client ask the builder to construct the trip planner structure for it

Another Example - Vacation Planner for Disney World(cont'd)

- you need a flexible data structure that can represent guest planners and all their variations
- you also need to follow a sequence of potentially complex steps to create the planner
- the builder design pattern can provide a way to create the complex structure without mixing it with the steps for creating it
- the java development kit also uses the builder pattern in
 - `java.lang.StringBuilder#append()`

Why the Builder Pattern?

- this pattern was introduced to solve problems with the Factory and Abstract Factory design patterns
 - these patterns do not work well when the Object to be created contains a lot of attributes
- there are three major issues
 - too many arguments to pass from the client to the Factory class
 - can be error prone
 - its hard to maintain the order of the arguments on the client side
 - some of the parameters might be optional
 - in the Factory pattern we are forced to send all the parameters
 - optional parameters need to be sent as NULL
 - if the object is heavy and its creation is complex
 - all that complexity will be part of factory classes which can cause major confusion

Why the Builder Pattern (cont'd)

- one way to solve the optional parameters problem
 - provide a constructor with required parameters and then different setter methods to set the optional parameters
 - however, the Object state will be inconsistent until/unless all the attributes are set explicitly
- a better approach is to use the Builder pattern
 - provides a way to build the object step-by-step
 - provides a method that will actually return the final complex object

Advantages

- it encapsulates the way a complex object is constructed
 - separates the code of assembling from its representation
 - hides the complex construction process and represents it as a simple process
- allows objects to be constructed in a multistep and varying process (as opposed to one-step factories)
- hides the internal representation of the product from the client
- product implementations can be swapped in and out because the client only sees an abstract interface
- focuses on “how the product will be made”

Disadvantages

- often used for building composite structures
- constructing objects requires more domain knowledge of the client than when using a Factory
- requires some amount of code duplication

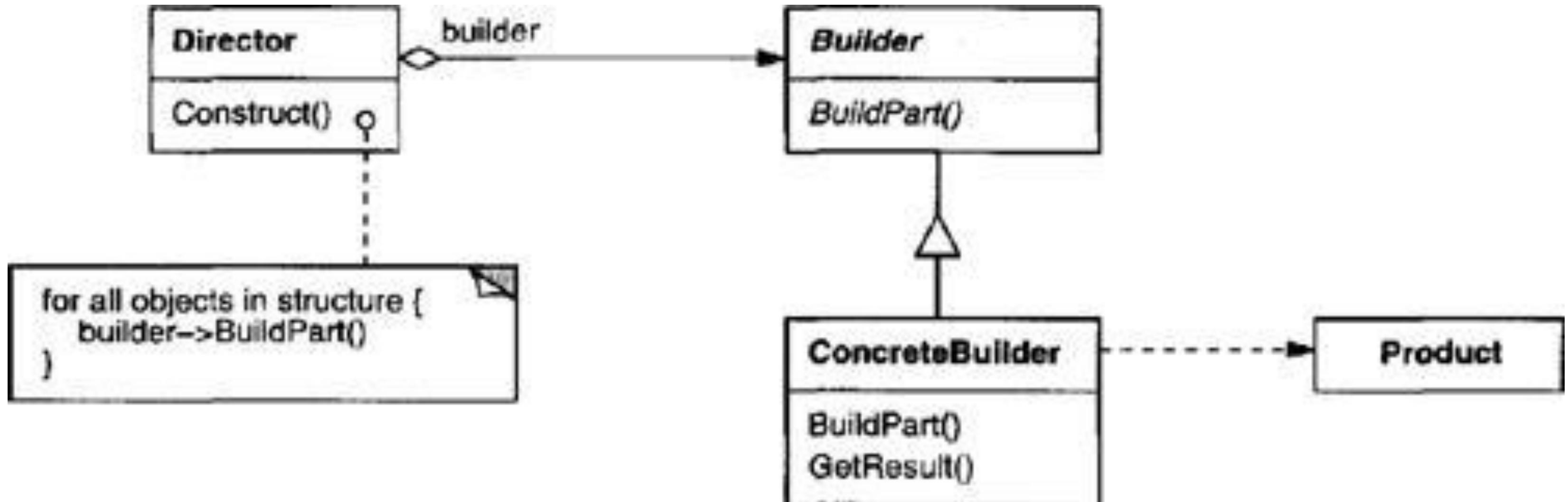
Summary

- use the Builder pattern when
 - the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
 - the construction process must allow different representations for the object that is constructed
- we should not use this pattern if we want a mutable object
 - an object which can be modified after the creational process is over

Implementing the Builder pattern

Jason Fedin

Class Diagram



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Overview

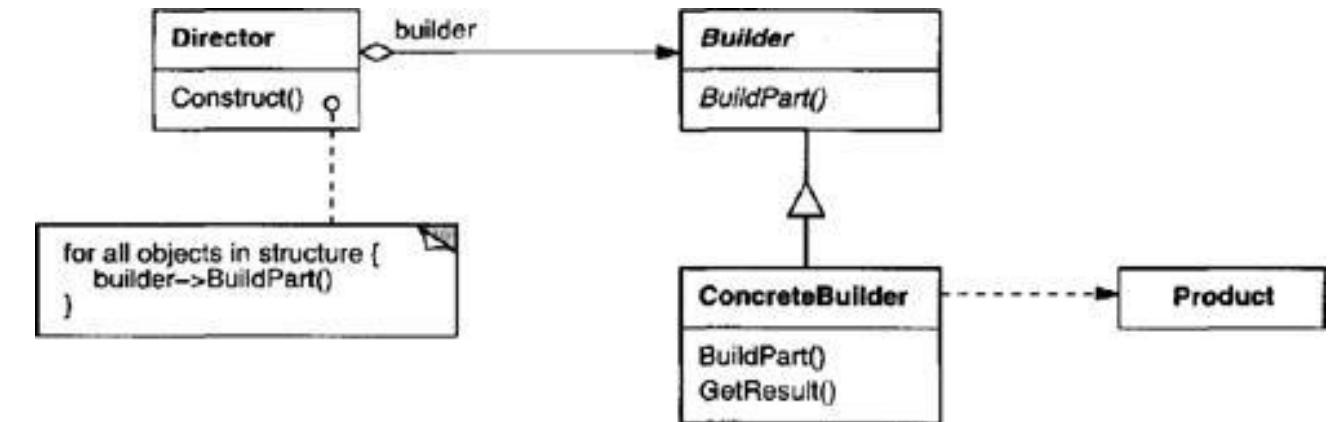
- the main participants when implementing the builder pattern are the following

- **Builder**

- specifies an abstract interface for creating parts of a Product object
- defines an operation for each component that a director may ask it to create
- must be general enough to allow the construction of products for all kinds of concrete builders

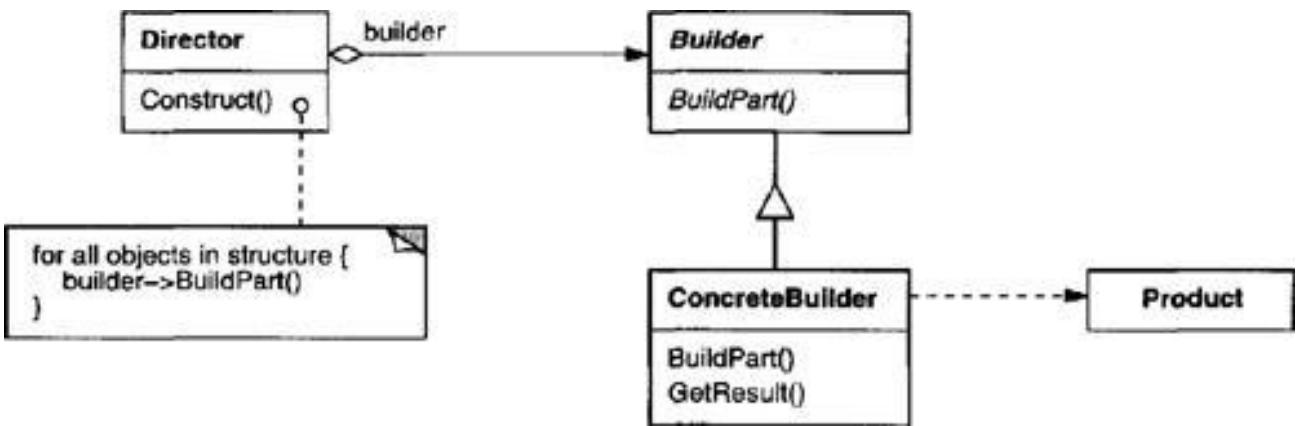
- **ConcreteBuilder**

- constructs and assembles parts of the product by implementing the Builder interface
- overrides operations for components it is interested in creating



Overview

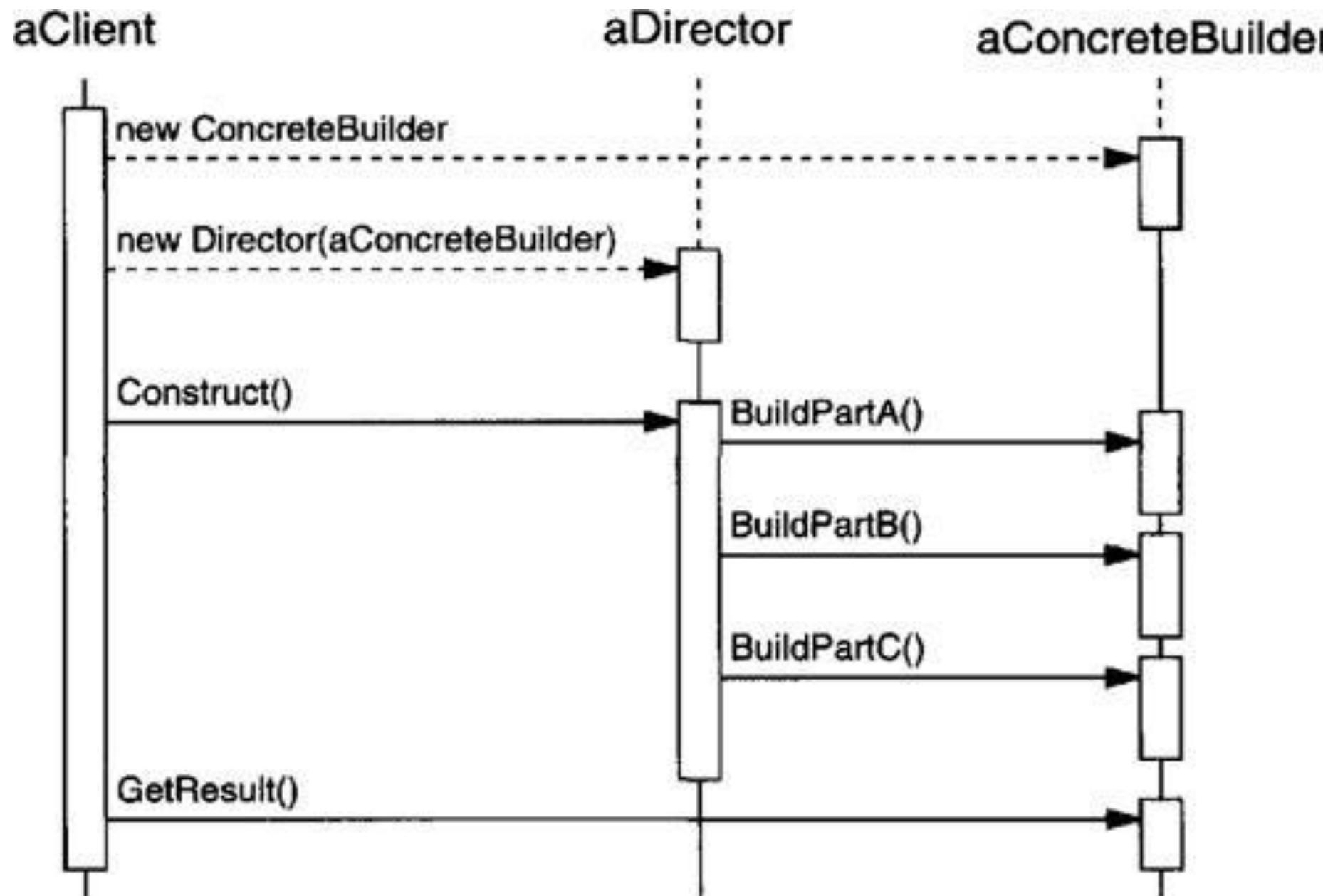
- ConcreteBuilder (cont'd)
 - defines and keeps track of the representation it creates
 - usually appended to the product via some type of list
 - sometimes you might need access to parts of the product constructed earlier
 - builder would return child nodes to the director, which then would pass them back to the builder to build the parent nodes
 - provides an interface for retrieving the product (`getProduct()`)
- Director
 - constructs an object using the Builder interface
- Product
 - represents the complex object under construction
 - ConcreteBuilder builds the product's internal representation and defines the process by which it is assembled
 - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result



Overview

- the client creates the Director object
 - configures it with the desired Builder object
- the Director notifies the builder whenever a part of the product should be built
- the Builder handles requests from the director and adds parts to the product
- the client retrieves the product from the builder

Sequence Diagram



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Implementation advantages

- lets you vary a product's internal representation
 - the Builder object provides the director with an abstract interface for constructing the product
 - the interface lets the builder hide the representation and internal structure of the product
 - also hides how the product gets assembled
 - because the product is constructed through an abstract interface, all you have to do to change the product's internal representation is define a new kind of builder
- it isolates code for construction and representation
 - improves modularity by encapsulating the way a complex object is constructed and represented
 - clients do not need to know anything about the classes that define the product's internal structure
 - classes do not appear in the Builder's interface
- each ConcreteBuilder contains all the code to create and assemble a particular kind of product
 - code is written once
 - different Directors can reuse it to build Product variants from the same set of parts

Implementation advantages (cont'd)

- gives you finer control over the construction process
- constructs the product step by step under the director's control
- only when the product is finished does the director retrieve it from the builder
- the Builder interface reflects the process of constructing the product more than other creational patterns
 - gives you finer control over the construction process and consequently the internal structure of the resulting product

Implementing the Builder pattern

Jason Fedin

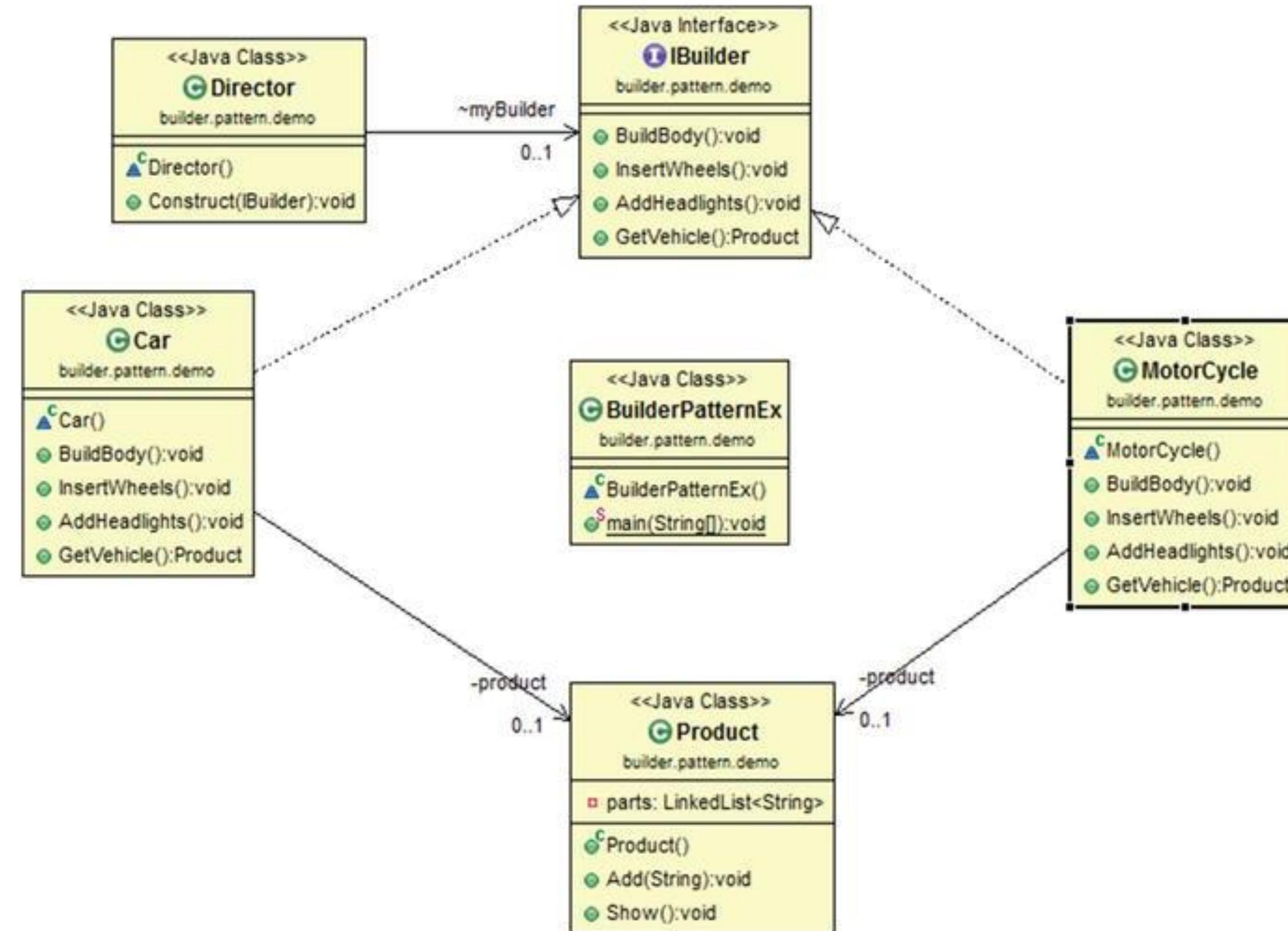
Example (walk through in intellij)

- here the participants are *IBuilder*, *Car*, *MotorCycle*, *Product*, and *Director*
- the first three are very straightforward
- *IBuilder* is used to create parts of the *Product* object
- *Car* and *MotorCycle* are implementing the *IBuilder* interface
 - concrete implementations that implement the following methods:
 - *BuildBody()*
 - build the body of the vehicle
 - *InsertWheels()*
 - insert the number of wheels into it
 - *AddHeadlights()*
 - add headlights to the vehicle
 - *GetVehicle()*
 - returns the ultimate product

Example (walk through in intellij)

- Product represents the complex object under construction
 - the assembly process is described in Product
 - we use the Linked List data structure in Product for this assembly operation
- Director will be responsible for constructing the ultimate vehicle
 - builds the product with IBuilder interface
 - calling the same Construct() method to create different types of vehicles

Class Diagram



Builder interface and concrete implementations

```
import java.util.LinkedList;

// Builders common interface
interface Ibuilder {
    void BuildBody();
    void InsertWheels();
    void AddHeadlights();
    Product GetVehicle();
}

// Car is ConcreteBuilder
class Car implements Ibuilder {
    private Product product = new Product();

    @Override
    public void BuildBody()
    {
        product.Add("This is a body of a Car");
    }
}
```

Concrete Implementations (Car)

```
@Override  
    public void InsertWheels() {  
        product.Add("4 wheels are added");  
    }  
  
@Override  
    public void AddHeadlights() {  
        product.Add("2 Headlights are added");  
    }  
  
@Override  
    public Product GetVehicle() {  
        return product;  
    }  
}
```

Concrete Implementations (Motorcycle)

```
// Motorcycle is a ConcreteBuilder
class MotorCycle implements Ibuilder {
    private Product product = new Product();

    @Override
    public void BuildBody() {
        product.Add("This is a body of a Motorcycle");
    }

    @Override
    public void InsertWheels() {
        product.Add("2 wheels are added");
    }

    @Override
    public void AddHeadlights() {
        product.Add("1 Headlights are added");
    }

    @Override
    public Product GetVehicle() {
        return product;
    }
}
```

Product

```
class Product {  
    // We can use any data structure that you prefer. We have used LinkedList here.  
    private LinkedList<String> parts;  
    public Product() {  
        parts = new LinkedList<String>();  
    }  
  
    public void Add(String part) {  
        //Adding parts  
        parts.addLast(part);  
    }  
  
    public void Show() {  
        System.out.println("\n Product completed as below :");  
        for(int i=0;i<parts.size();i++) {  
            System.out.println(parts.get(i));  
        }  
    }  
}
```

Director

```
class Director {  
    IBuilder myBuilder;  
  
    // A series of steps—for the production  
    public void Construct(IBuilder builder) {  
        myBuilder=builder;  
        myBuilder.BuildBody();  
        myBuilder.InsertWheels();  
        myBuilder.AddHeadlights();  
    }  
}
```

Demo Class

```
class BuilderPatternEx {  
    public static void main(String[] args) {  
        System.out.println("***Builder Pattern Demo***\n");  
  
        Director director = new Director();  
  
        IBuilder carBuilder = new Car();  
        IBuilder motorBuilder = new MotorCycle();  
  
        // Making Car  
        director.Construct(carBuilder);  
        Product p1 = carBuilder.GetVehicle();  
        p1.Show();  
  
        //Making MotorCycle  
        director.Construct(motorBuilder);  
        Product p2 = motorBuilder.GetVehicle();  
        p2.Show();  
    }  
}
```

Output

```
Console X
<terminated> BuilderPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Jun 10, 2015, 3:56:43 PM)
***Builder Pattern Demo***

Product completed as below :
This is a body of a Car
4 wheels are added
2 Headlights are added

Product completed as below :
This is a body of a Motorcycle
2 wheels are added
1 Headlights are added
```

Summary

- we separated the code of assembling from its representation
 - hides the complex construction process and represents it as a simple process
- we focus on “how the product will be made”
- we have only one method which will finally return the complete object
 - other methods will be responsible for partial creation process only
- one drawback is that it requires some amount of code duplication

(Demonstration) Builder

Jason Fedin

Demonstrate in intelliJ

Step 1 - Create interfaces Item and packing (Item.java, Packing.java)

```
public interface Item {  
    public String name();  
    public Packing packing();  
    public float price();  
}
```

```
public interface Packing {  
    public String pack();  
}
```

Demonstrate in intelliJ

Step 2 - Create concrete classes implementing the Packing interface (Wrapper.java, Bottle.java)

```
public class Wrapper implements Packing {
```

```
    @Override
```

```
    public String pack() {
```

```
        return "Wrapper";
```

```
}
```

```
}
```

```
public class Bottle implements Packing {
```

```
    @Override
```

```
    public String pack() {
```

```
        return "Bottle";
```

```
}
```

```
}
```

Demonstrate in intellij

Step 3 - Create abstract classes implementing the item interface providing default functionalities (Burger.java, ColdDring.java)

```
public abstract class Burger implements Item {
```

```
    @Override
```

```
    public Packing packing() {
```

```
        return new Wrapper();
```

```
}
```

```
    @Override
```

```
    public abstract float price();
```

```
}
```

```
public abstract class ColdDrink implements Item {
```

```
    @Override
```

```
    public Packing packing() {
```

```
        return new Bottle();
```

```
}
```

```
    @Override
```

```
    public abstract float price();
```

```
}
```

Demonstrate in intelliJ

Step 4 - Create concrete classes extending Burger and ColdDrink classes (VegBurger.java, ChickenBurger.java, Coke.java, Pepsi.java)

```
public class VegBurger extends Burger {
```

```
    @Override
```

```
    public float price() {  
        return 25.0f;
```

```
}
```

```
    @Override
```

```
    public String name() {  
        return "Veg Burger";
```

```
}
```

```
}
```

Demonstrate in intellij

```
public class ChickenBurger extends Burger {
```

```
    @Override  
    public float price() {  
        return 50.5f;  
    }
```

```
    @Override  
    public String name() {  
        return "Chicken Burger";  
    }  
}
```

Demonstrate in intellij

```
public class Coke extends ColdDrink {
```

```
    @Override  
    public float price() {  
        return 30.0f;  
    }
```

```
    @Override  
    public String name() {  
        return "Coke";  
    }  
}
```

Demonstrate in intellij

```
public class Pepsi extends ColdDrink {
```

```
    @Override  
    public float price() {  
        return 35.0f;  
    }
```

```
    @Override  
    public String name() {  
        return "Pepsi";  
    }  
}
```

Demonstrate in intellij

Step 5 - Create a Meal class having Item objects defined in previous slides. THIS IS THE PRODUCT (Meal.java)

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
    private List<Item> items;

    public Meal()
    {
        items = new ArrayList<Item>();
    }

    public void addItem(Item item){
        items.add(item);
    }
}
```

Demonstrate in intellij (Meal.java)

```
public float getCost(){  
    float cost = 0.0f;  
  
    for (Item item : items) {  
        cost += item.price();  
    }  
    return cost;  
}  
  
public void showItems(){  
    for (Item item : items) {  
        System.out.print("Item : " + item.name());  
        System.out.print(", Packing : " + item.packing().pack());  
        System.out.println(", Price : " + item.price());  
    }  
}
```

Demonstrate in intellij

- Step 6 - Create an interface for the abstractBuilder (IMealBuilder.java)

interface IMealBuilder

```
{  
    void buildBurger();  
    void buildDrink();  
    Meal getMeal();  
}
```

Demonstrate in intellij

Step 7 - Create concrete VegMealBuilder and NonVegMealBuilder classes, the actual builder class responsible to create Meal objects (VegMealBuilder.java, NonVegMealBuilder.java)

```
public class VegMealBuilder implements IMealBuilder
{
    private Meal vegMeal = new Meal();

    @Override
    public void buildBurger(){
        vegMeal.addItem(new VegBurger());
    }

    @Override
    public void buildDrink() {
        vegMeal.addItem(new Coke());
    }

    @Override
    public Meal getMeal() {
        return vegMeal;
    }
}
```

Demonstrate in intellij

```
public class NonVegMealBuilder implements IMealBuilder
{
    private Meal nonVegMeal = new Meal();

    @Override
    public void buildBurger(){
        nonVegMeal.addItem(new ChickenBurger());
    }

    @Override
    public void buildDrink() {
        nonVegMeal.addItem(new Pepsi());
    }

    @Override
    public Meal getMeal() {
        return nonVegMeal;
    }
}
```

Demonstrate in intellij

Step 8 - Create the director

```
// "Director"  
class Director  
{  
    IMealBuilder myBuilder;  
  
    // A series of steps—for the production  
    public void Construct(IMealBuilder builder)  
    {  
        myBuilder=builder;  
        myBuilder.buildBurger();  
        myBuilder.buildDrink();  
    }  
}
```

Demonstrate in intellij

Step 9 - BuiderPatternDemo uses MealBuider to demonstrate builder pattern (BuilderPatternDemo.java)

```
class BuilderPatternEx {  
    public static void main(String[] args) {  
        System.out.println("/**Meal Pattern Demo**\n");  
  
        Director director = new Director();  
  
        IMealBuilder vegMealBuilder = new VegMealBuilder();  
        IMealBuilder nonVegMealBuilder = new NonVegMealBuilder();  
  
        // Making veg meal  
        director.Construct(vegMealBuilder);  
        Meal vegMeal = vegMealBuilder.getMeal();  
        System.out.println("Veg Meal");  
        vegMeal.showItems();  
        System.out.println("Total Cost: " + vegMeal.getCost());  
    }  
}
```

Demonstrate in intellij

```
//Making nonVegMeal
director Construct(nonVegMealBuilder);
Meal nonVegMeal = nonVegMealBuilder.getMeal();
System.out.println("\n\nNon-Veg Meal");
nonVegMeal.showItems();
System.out.println("Total Cost: " + nonVegMeal.getCost());
}
}
```

Demonstrate in intellij

Step 10 - Verify the output

Veg Meal

Item : Veg Burger, Packing : Wrapper, Price : 25.0

Item : Coke, Packing : Bottle, Price : 30.0

Total Cost: 55.0

Non-Veg Meal

Item : Chicken Burger, Packing : Wrapper, Price : 50.5

Item : Pepsi, Packing : Bottle, Price : 35.0

Total Cost: 85.5

Prototype design pattern

Jason Fedin

Overview

- prototype pattern refers to creating a duplicate object while keeping performance in mind
 - specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype
- used when creation of an object is costly, requires a lot of time and resources and you have a similar object already existing
 - creating a new instance is normally treated as an expensive operation
 - focus here is to reduce the expense of this creational process of a new instance
- provides a mechanism to copy the original object to a new object and then modify it according to our needs
 - uses java cloning to copy the object (shallow) or de-serialization when you need deep copies
- a key aspect of this pattern is that the client code can make new instances without knowing which specific class is being instantiated
- mandates that the Object which you are copying should provide the copying feature
 - should not be done by any other class
 - whether to perform a shallow or deep copy of the Object depends on the requirements and design

Examples

- suppose we have a master copy of a valuable document
 - we want to make some changes to it, in order to get a different feel
 - we can make a photocopy of this document and then try to edit our changes
- suppose we have made an application
 - we want to create a similar application with some small changes
 - start with a copy from our master copy application and make the changes
 - saves a lot of time and resources
- suppose we have an Object that loads data from database
 - we need to modify this data in our program multiple times
 - not a good idea to create the Object using the new keyword and load all the data again from database
 - we can cache the object, returns a clone of the object on the next request
 - update the database when needed
 - reduces the number of database calls

When should we use a prototype?

- when a system should be independent of how its products are created, composed, and represented
 - does not care about the creational mechanism of the products
- we can use this pattern when we need to instantiate classes at runtime
 - dynamic loading
- when a system must create new objects of many types in a complex class hierarchy
 - you want to build a class hierarchy of factories that parallels the class hierarchy of products
- when instances of a class can have one of only a few different combinations of state
 - may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state

Advantages

- hides the complexities of making new instances from the client
- provides the option for the client to generate objects whose type is not known
- in some circumstances, copying an object can be more efficient than creating a new object
- we can include or discard products at runtime
- we can create new instances with a cheaper cost

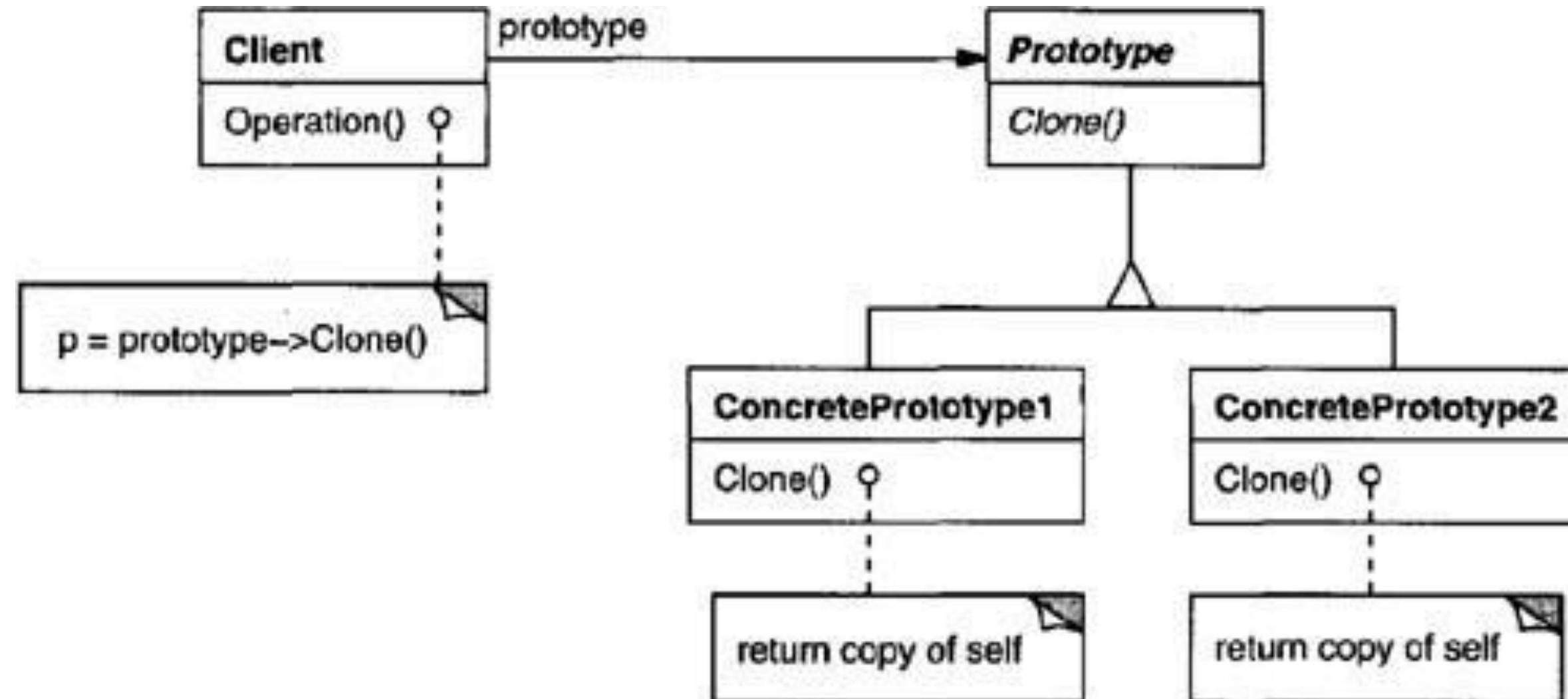
Drawbacks

- each subclass has to implement the cloning mechanism
- implementing the cloning mechanism can be challenging
 - if the objects under consideration do not support copying
 - if there is any kind of circular reference
- the java cloneable interface has some problems
 - will discuss in an upcoming lecture

Implementing the Prototype design pattern

Jason Fedin

Class Diagram



Design Patterns: Elements of Reusable Object-Oriented Software By: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

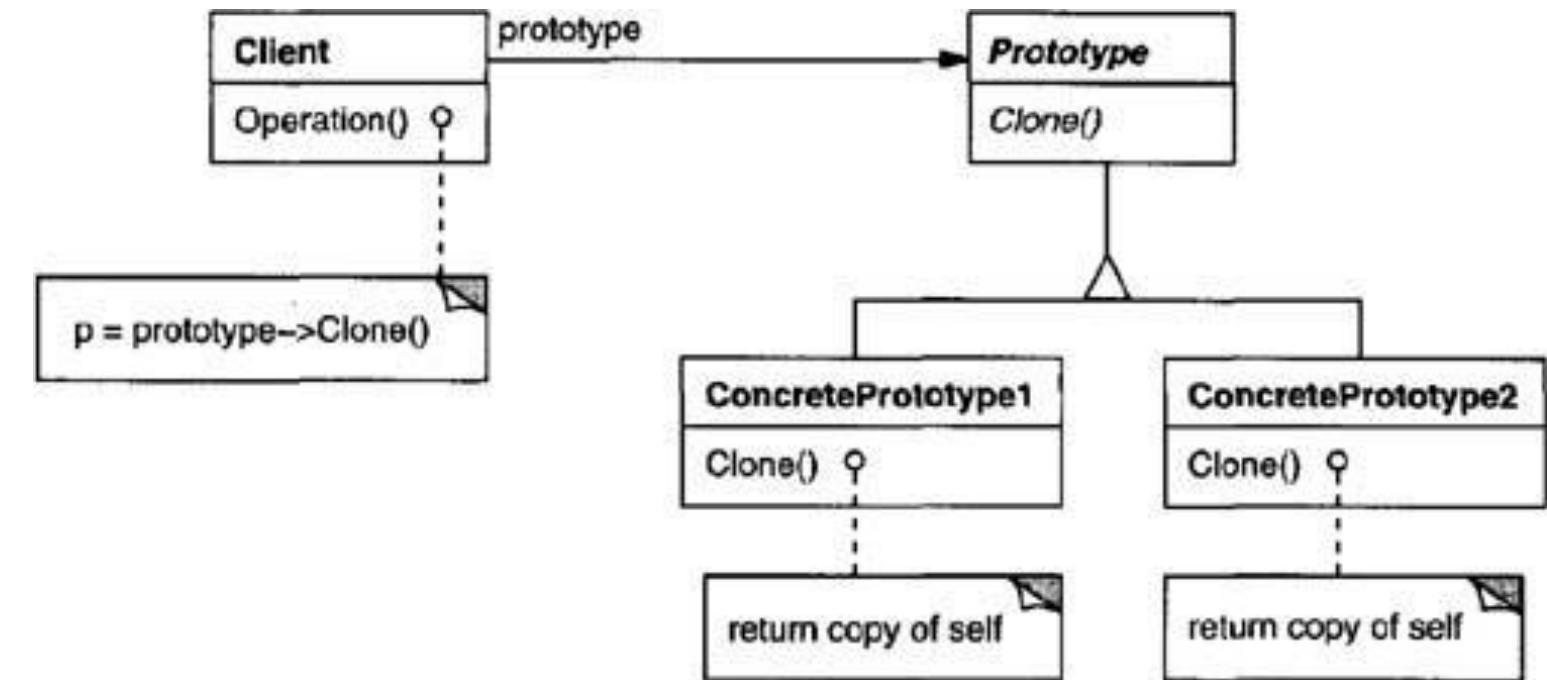
Overview

- when implementing the prototype pattern the following participants are included

- Prototype
 - declares an interface for cloning itself

- ConcretePrototype
 - implements an operation for cloning itself

- Client
 - creates a new object by asking a prototype to clone itself



Implementation advantages

- it hides the concrete product classes from the client
 - reduces the number of names clients know about
 - let a client work with application-specific classes without modification
- let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client
 - more flexible than other creational patterns, because a client can install and remove prototypes at run-time
- let you define new behavior through object composition
 - by specifying values for an object's variables, for example and not by defining new classes
 - you define a new kinds of objects by instantiating existing classes and registering the instances as prototypes of client objects
 - a client can exhibit new behavior by delegating responsibility to the prototype

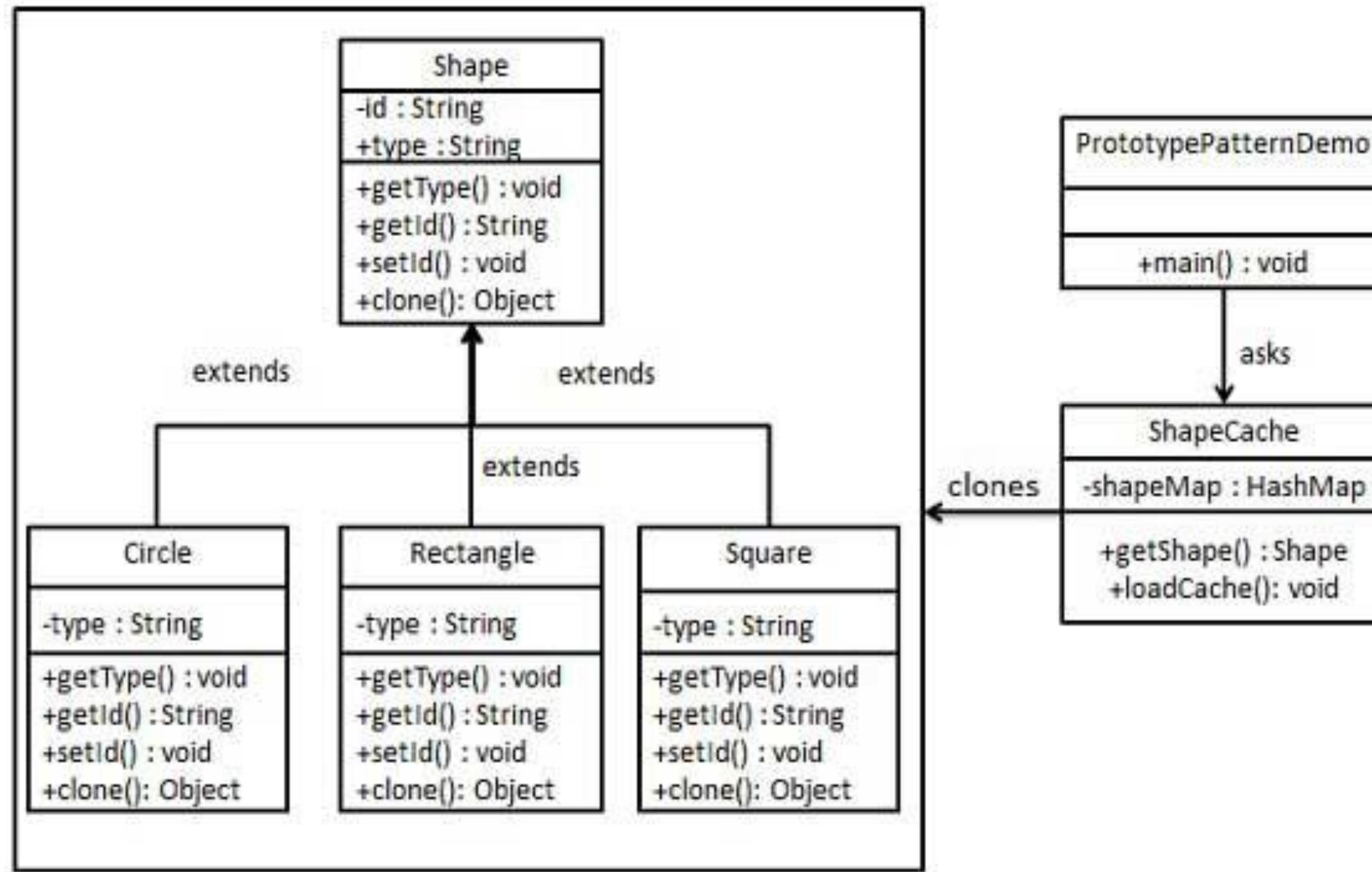
Implementation advantages (cont'd)

- lets users define new “classes” without programming
 - cloning a prototype is similar to instantiating a class
 - can greatly reduce the number of classes a system needs
- reduces subclassing
 - factory Method often produces a hierarchy of Creator classes that parallels the product class hierarchy
 - the prototype pattern lets you clone a prototype instead of asking a factory method to make a new object
 - do not need a Creator class hierarchy at all

Example demo in intellij

- we are going to create an abstract class Shape and concrete classes extending the Shape class
- a class ShapeCache will be defined which stores shape objects in a Hashtable and returns their clone when requested
- PrototypPatternDemo is our demo class
 - will use ShapeCache class to get a Shape object

Class Diagram



Code

Step 1 - Create an abstract class implementing Clonable interface (Shape.java)

```
public abstract class Shape implements Cloneable {  
  
    private String id;  
    protected String type;  
  
    abstract void draw();  
  
    public String getType(){  
        return type;  
    }  
}
```

Code (Shape.java)

```
public String getId() {  
    return id;  
}  
  
public void setId(String id) {  
    this.id = id;  
}  
  
public Object clone() {  
    Object clone = null;  
  
    try {  
        clone = super.clone();  
  
    } catch (CloneNotSupportedException e) {  
        e.printStackTrace();  
    }  
  
    return clone;  
}
```

Code

Step 2 - Create concrete classes extending the above class (Rectangle.java, Square.java, Circle.java)

```
public class Rectangle extends Shape {  
    public Rectangle(){  
        type = "Rectangle";  
    }
```

```
@Override  
public void draw() {  
    System.out.println("Inside Rectangle::draw() method.");  
}
```

```
public class Square extends Shape {  
    public Square(){  
        type = "Square";  
    }
```

```
@Override  
public void draw() {  
    System.out.println("Inside Square::draw() method.");  
}
```

Code (Circle.java)

```
public class Circle extends Shape {  
  
    public Circle(){  
        type = "Circle";  
    }  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Circle::draw() method.");  
    }  
}
```

Code

Step 3 - Create a class to get concrete classes from database and store them in a Hashtable
(ShapeCache.java)

```
import java.util.Hashtable;

public class ShapeCache {

    private static Hashtable<String, Shape> shapeMap = new Hashtable<String, Shape>();

    public static Shape getShape(String shapeld) {
        Shape cachedShape = shapeMap.get(shapeld);
        return (Shape) cachedShape.clone();
    }
}
```

Code

```
// for each shape run database query and create shape  
// shapeMap.put(shapeKey, shape);  
// for example, we are adding three shapes
```

```
public static void loadCache() {  
    Circle circle = new Circle();  
    circle.setId("1");  
    shapeMap.put(circle.getId(),circle);  
  
    Square square = new Square();  
    square.setId("2");  
    shapeMap.put(square.getId(),square);  
  
    Rectangle rectangle = new Rectangle();  
    rectangle.setId("3");  
    shapeMap.put(rectangle.getId(), rectangle);  
}  
}
```

Code

Step 4 - PrototypePatternDemo uses ShapeCache class to get clones of shapes stored in a Hashtable
(PrototypePatternDemo.java)

```
public class PrototypePatternDemo {  
    public static void main(String[] args) {  
        ShapeCache.loadCache();  
  
        Shape clonedShape = (Shape) ShapeCache.getShape("1");  
        System.out.println("Shape : " + clonedShape.getType());  
  
        Shape clonedShape2 = (Shape) ShapeCache.getShape("2");  
        System.out.println("Shape : " + clonedShape2.getType());  
  
        Shape clonedShape3 = (Shape) ShapeCache.getShape("3");  
        System.out.println("Shape : " + clonedShape3.getType());  
    }  
}
```

Output

Step 5 - Verify the output.

Shape : Circle

Shape : Square

Shape : Rectangle

Problems with the Cloneable interface

Jason Fedin

Overview

- from our previous lecture, we know that the implementation of the prototype design pattern involves implementing the cloneable interface
- java provides a mechanism for cloning of objects that is very easy to implement
- you first need to Implement the Cloneable interface
- you then need to define a clone() method that should handle CloneNotSupportedException
 - returns a shallow copy of the object
 - a shallow copy means if the copied object contains references to other objects, these objects are not cloned
 - a deep copy would clone even referenced objects
- lastly, we call the clone() method of the superclass

Example of using the Cloneable interface

```
class Person implements Cloneable { // step 1
    private String name;
    private City city; // deep copy

    // no @override, means we are not overriding clone
    public Person clone() throws CloneNotSupportedException { // Step 2
        Person clonedObj = (Person) super.clone(); // step 3
        clonedObj.city = this.city.clone(); // Making deep copy of city
        return clonedObj;
    }
}
```

Advantages of using the Cloneable interface

- Object.clone() is the most popular and easiest way of copying objects
 - this is why we use it when implementing the prototype design pattern
 - we need to define a parent class
 - implement Cloneable in it
 - provide the definition of the clone() method
 - every child of our parent will get the cloning feature
- cloning requires very few lines of code
 - an abstract class with 4 or 5 lines in the clone() method
 - also need to override it if we need deep cloning
- we should use clone to copy arrays because that is generally the fastest way to do it

Problems with the Cloneable interface

- some academics think that cloning is deeply broken in Java
- the Cloneable interface lacks the `clone()` method
 - `Cloneable` is a marker interface and does not have any methods in it
 - we still need to implement it just to tell the JVM that we can perform `clone()` on our object
- `Object.clone()` is protected
 - we have to provide our own `clone()` and indirectly call `Object.clone()` from it
- we do not have any control over object construction because `Object.clone()` does not invoke any constructor
 - there are no guarantees that it preserves the invariants established by the constructors

Problems with the Cloneable interface (cont'd)

- If we are writing a clone method in a child class then all of its superclasses should define the clone() method
 - otherwise, the super.clone() chain will fail
- Object.clone() supports only shallow copying
 - does not clone the reference fields of the object to be cloned
 - we need to implement clone() in every class whose reference our class is holding
 - then call their clone separately in our clone() method
- we can not manipulate final fields in Object.clone()
 - final fields can only be changed through constructors
 - if we want every object to be unique by including an id, we will get the duplicate object if we use Object.clone()
 - Object.clone() will not call the constructor, and final id field can not be modified from invoking the clone() method

Problems with the Cloneable interface (cont'd)

- you can not do a polymorphic clone operation
 - if I have an array of Cloneable, you would think that I could run down that array and clone every element to make a deep copy of the array
 - this does not work, you cannot cast something to Cloneable and call the clone method
 - Cloneable does not have a public clone method and neither does Object
 - If you try to cast to Cloneable and call the clone method, the compiler will say you are trying to call the protected clone method on object
- the clone generally shares state with the object being cloned
 - if that state is mutable, you do not have two independent objects
 - if you modify one, the other changes as well and all of a sudden, you get random behavior
- Cloneable is a weak spot, and you should be aware of its limitations

Alternatives to using Clonable (Copy Constructor)

- one option to provide copy functionality is to provide a copy constructor(s)
 - like a regular constructor, which returns a new instance of the class
 - as an input, it has an object, which is supposed to be copied
 - inside the body of the constructor, you implement your custom cloning logic
- this method of copying objects is one of the most popular among the developer community
 - overcomes every design issue of Object.clone()
 - provides better control over object construction

Copy Constructor Example

```
public Person(Person original)
{
    this.id = original.id + 1;
    this.name = new String(original.name);
    this.city = new City(original.city);
}
```

Advantages of Copy Constructors vs clone()

- does not force us to implement any interface or throw any exception
- does not require any casts
- does not require us to depend on an unknown object creation mechanism
- does not require parent classes to follow any contract or implement anything
- allows us to modify final fields
- allows us to have complete control over object creation
 - we can write our own initialization logic in it
- we can also create conversion constructors
 - allow us to convert one object to another object

Alternatives to using Cloneable (Serialization)

- another way to copy an object is to use a serialize/deserialize approach
 - instead of cloning, you can serialize an object and then immediately deserialize it
 - would result in a new instance created
- we will still not be able to modify the final fields
- we still do not have any control on object construction
- we still need to implement Serializable, which is similar to Cloneable
- the serialization process is slower than Object.clone()

Serialize/Deserialize approach

```
public Person copy(Person original)
{
    try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("data.obj"));
        ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj")))
    {
        out.writeObject(original);
        return (Person) in.readObject();
    }
    catch (Exception e)
    {
        throw new RuntimeException(e);
    }
}
```

Advantages of cloning using serialization

- simple alternative to cloning
 - especially when using library such as Apache Commons
- provides deep cloning
- suitable even for complex object graphs
- can be used on existing classes that currently provide just shallow copy

Summary

- implementing the prototype design pattern using the clone method provided by java is still the preferred approach
 - implement the Cloneable interface
 - override the clone() method
 - call super.clone() if a shallow copy is sufficient
 - implement custom cloning logic, if a deep copy is required
- alternatively, you can implement copying an object in the following ways
 - implement your own copy constructor
 - use various existing third-party libraries to implement a serialization / deserialization approach

(Demonstration) Prototype

Jason Fedin

Code (BasicCar.java)

```
import java.util.Random;

public abstract class BasicCar implements Cloneable {
    public String modelname;
    public int price;

    public String getModelname()
    {
        return modelname;
    }

    public void setModelname(String modelname)
    {
        this.modelname = modelname;
    }
}
```

Code (BasicCar.java)

```
public static int setPrice() {  
    int price = 0;  
    Random r = new Random();  
    int p = r.nextInt(100000);  
    price = p;  
    return price;  
}  
  
public BasicCar clone() throws CloneNotSupportedException {  
    return (BasicCar)super.clone();  
}  
}
```

Code (Ford.java)

```
public class Ford extends BasicCar {  
  
    public Ford(String m)  
    {  
        modelname = m;  
    }  
  
    @Override  
    public BasicCar clone() throws CloneNotSupportedException  
    {  
        return (Ford)super.clone();  
    }  
}
```

Code (Nano.java)

```
public class Nano extends BasicCar {  
  
    public Nano(String m)  
    {  
        modelname = m;  
    }  
  
    @Override  
    public BasicCar clone() throws CloneNotSupportedException  
    {  
        return (Nano)super.clone();  
    }  
}
```

Code (BasicCarCache.java)

```
import java.util.Hashtable;

public class BasicCarCache {

    private static Hashtable<String, BasicCar> basicCarMap = new Hashtable<String,
    BasicCarMap>();

    public static BasicCarMap getCar(String carName) {
        BasicCar cachedBasicCar = basicCarMap.get(carName);
        return (BasicCar) cachedBasicCar.clone();
    }
}
```

Code (BasicCarCache.java)

```
public static void loadCache() {  
    BasicCar nano_base = new Nano("Green Nano");  
    nano_base.price=100000;  
    basicCarMap.put("Green Nano", nano_base);  
  
    BasicCar ford_basic = new Ford("Ford Yellow");  
    ford_basic.price=500000;  
    basicCarMap.put("Ford Yellow", ford_basic);  
}  
}
```

Code (PrototypePatternEx.java)

```
public class PrototypePatternEx {  
    public static void main(String[] args) throws CloneNotSupportedException {  
        System.out.println("/**Prototype Pattern Demo**\n");  
        BasicCarCache loadCache();  
  
        BasicCar bc1 = (BasicCar ) BasicCarCache.getCar("Green Nano");  
  
        //Price will be more than 100000 for sure  
        bc1.price = bc1.price+BasicCar.setPrice();  
        System.out.println("Car is: "+ bc1.modelname+" and it's price is Rs."+bc1.price);  
  
        bc1 = (BasicCar ) BasicCarCache.getCar("Ford Yellow");  
  
        //Price will be more than 500000 for sure  
        bc1.price = bc1.price+BasicCar.setPrice();  
        System.out.println("Car is: "+ bc1.modelname+" and it's price is Rs."+bc1.price);  
    }  
}
```

Output

The screenshot shows a Java application running in an IDE's console. The title bar indicates it is a Java Application named 'PrototypePatternEx' (version 2) running on Jun 7, 2015, at 9:49:47 AM. The console tab is active, displaying the following output:

```
<terminated> PrototypePatternEx (2) [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Jun 7, 2015, 9:49:47 AM)
|***Prototype Pattern Demo***

Car is: Green Nano and it's price is Rs.189818
Car is: Ford Yellow and it's price is Rs.561925
```

Notes

- may want to also include in the demonstration an example of a deep copy
 - look at example from “problems with cloneable” slides
 - could also show how to utilize a copy constructor

Summary

- we have used the default clone() method in Java
 - which is a shallow copy
 - it is inexpensive compared to a deep copy
- advantages of this pattern are:
 - can include or discard products at runtime
 - can create new instances with a cheaper cost

Structural Design Patterns

Jason Fedin

Structural patterns

- describes how classes and objects can be combined to form larger structures
 - utilizes inheritance to compose interfaces or implementations
 - structural object patterns describe ways to assemble objects
 - e.g. complex user interfaces and accounting data
- these design patterns concern class and object composition
- the composite design pattern
 - describes how to build a class hierarchy made up of classes for two kinds of objects
- the proxy design pattern acts as a convenient surrogate or placeholder for another object.
 - provide a level of indirection to specific properties of objects

Class Patterns vs. Object Patterns (sub-category)

- class patterns describe how relationships between classes are defined
 - use inheritance to compose interfaces or implementations
 - relationships are established at compile time
 - adapter
- object patterns describe relationships between objects
 - describe ways to compose objects to realize new functionality
 - use composition
 - relationships are typically created at runtime
 - more dynamic and flexible
 - bridge, composite, decorator, façade, flyweight and proxy patterns
- there are seven structural patterns that we will study
 - will highlight their similarities and differences

Structural Patterns that we will study

- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator Pattern
- Facade Pattern
- Flyweight Pattern
- Proxy Pattern

Adapter Design Pattern

Jason Fedin

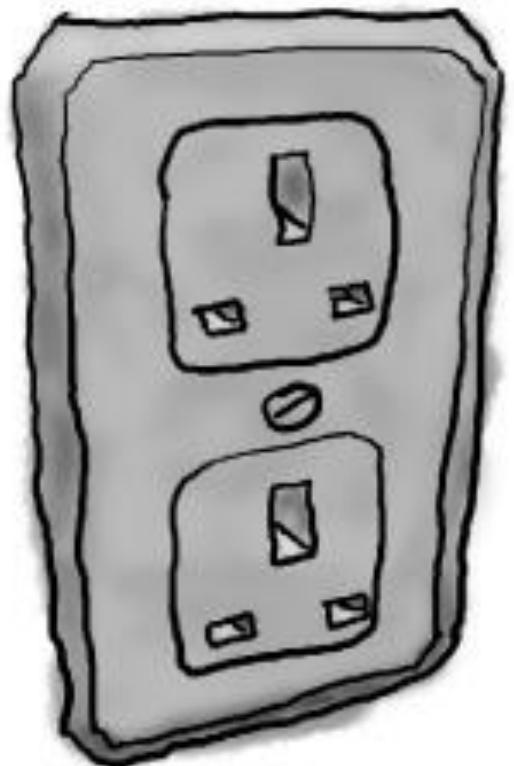
Structural patterns

- the adapter design pattern converts an interface of a class into another interface that clients expect
 - works as a bridge between two incompatible interfaces
 - “adapter” does the conversion
 - lets classes work together that could not otherwise
 - also known as a “wrapper”
- comes under the structural pattern classification as this pattern combines the capability of two independent interfaces
- the adapter acts to decouple the client from the implemented interface
 - encapsulates any future changes
 - client does not need to be modified each time it needs to operate against a different interface
- full of good OO design principles
 - use of object composition to wrap the adaptee with an altered interface
 - can use an adapter with any subclass of the adaptee
 - binds the client to an interface, not an implementation

Example

- have you ever needed to use a US-made laptop in Great Britain? Then you've probably needed an AC power adapter...

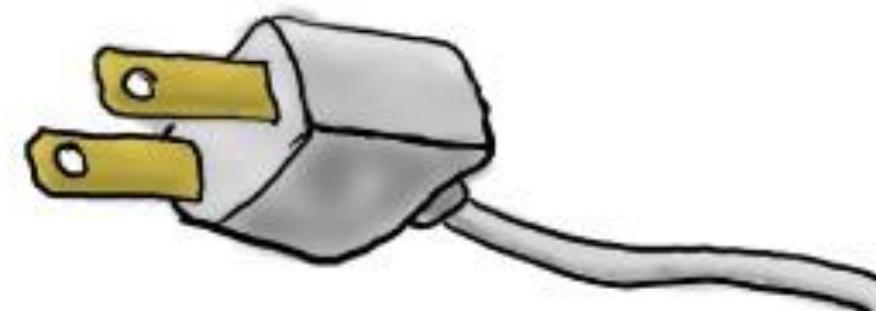
British Wall Outlet



AC Power Adapter



Standard AC Plug



The British wall outlet exposes one interface for getting power.

The US laptop expects another interface.



The adapter converts one interface into another.

- the adapter sits in between the plug of your laptop and the British AC outlet
- its job is to adapt the British outlet so that you can plug your laptop into it and receive power
- the adapter changes the interface of the outlet into one that your laptop expects

Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Real World Examples

- simple AC adapters
 - only change the shape of the outlet so that it matches your plug
 - pass the AC current straight through
- complex AC adapters
 - may need to step the power up or down to match your devices' needs
- card reader
 - acts as an adapter between memory card and a laptop
 - plug the memory card into the card reader
 - plug the card reader into the laptop
 - the memory card can now be read via laptop

Real World Examples (cont'd)

- mobile charging devices
 - an adapter is needed If a charger is not supported by a particular kind of switchboard
 - mobile battery needs 3 volts to charge
 - normal socket produces either 120V (US) or 240V (India)
 - mobile charger works as an adapter between mobile charging socket and the wall socket
- a translator
 - translating a language for one person to another language for another person

Software Examples

- you have an existing software system
 - you need to add a new vendor class library
 - the new vendor designed their interfaces differently than the last vendor
- you do not want to solve the problem by changing your existing code and you cannot change the vendor's code
 - write a class that adapts the new vendor interface into the one you are
- adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes

Software Examples (cont'd)

- old-world Enumerators
 - early collection types (Vector, Stack, Hashtable, etc.) implement a method (`elements()`), which returns an Enumeration
 - the Enumeration interface allows you to step through the elements of a collection
 - do not need to know the specifics of how they are managed in the collection
- new-world Iterators
 - newer Collection classes use an Iterator interface
 - like Enumeration
 - allows you to iterate through a set of items in a collection
 - allows you to remove items
 - both of these examples use adapters

Examples in the JDK

- `java.util.Arrays#asList()`
- `java.io.InputStreamReader(InputStream)` (returns a Reader)
- `java.io.OutputStreamWriter(OutputStream)` (returns a Writer)

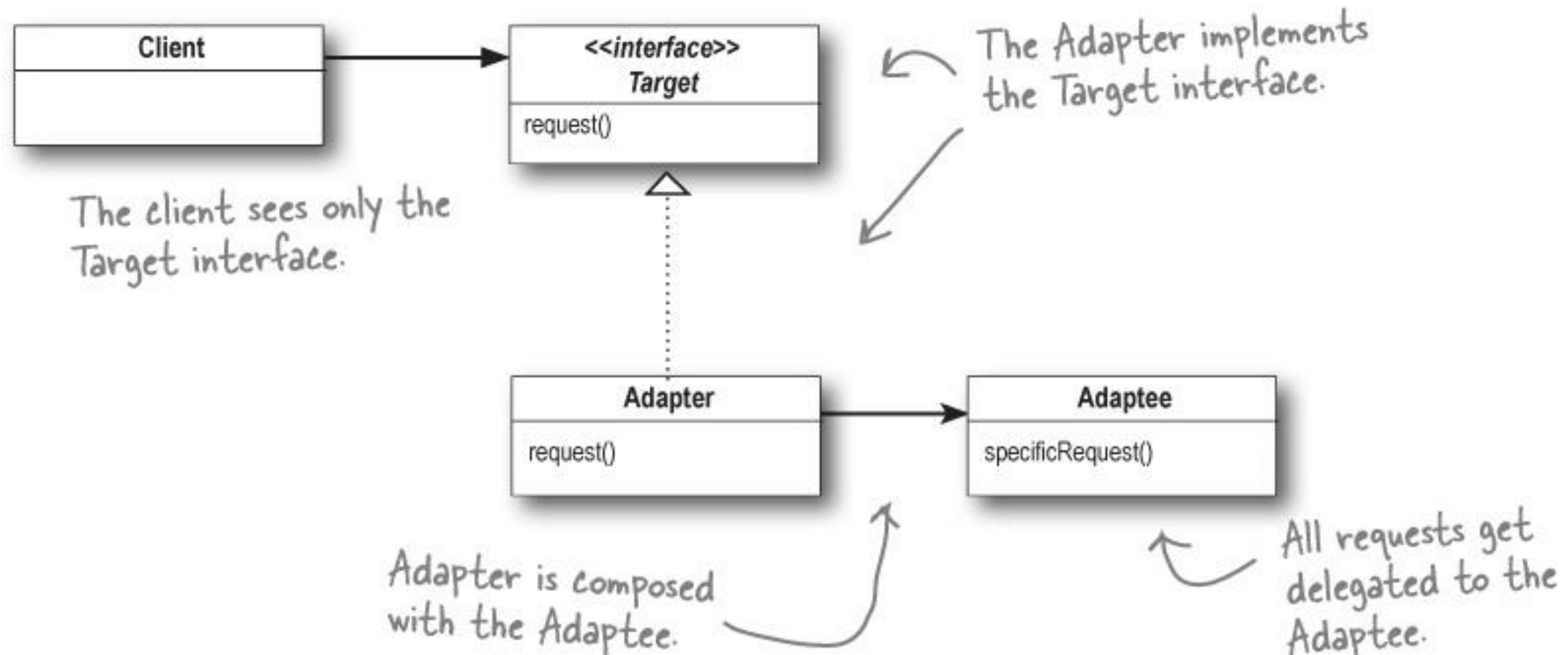
When to use the Adapter Pattern

- when you want to use an existing class, and its interface does not match the one you need
- when you want to create a reusable class that cooperates with unrelated or unforeseen classes
 - classes that do not necessarily have compatible interfaces
- when you need to use several existing subclasses, but it is impractical to adapt their interface by sub-classing every one
 - an object adapter can adapt the interface of its parent class

Object Adapter Implementation

Jason Fedin

Class Diagram

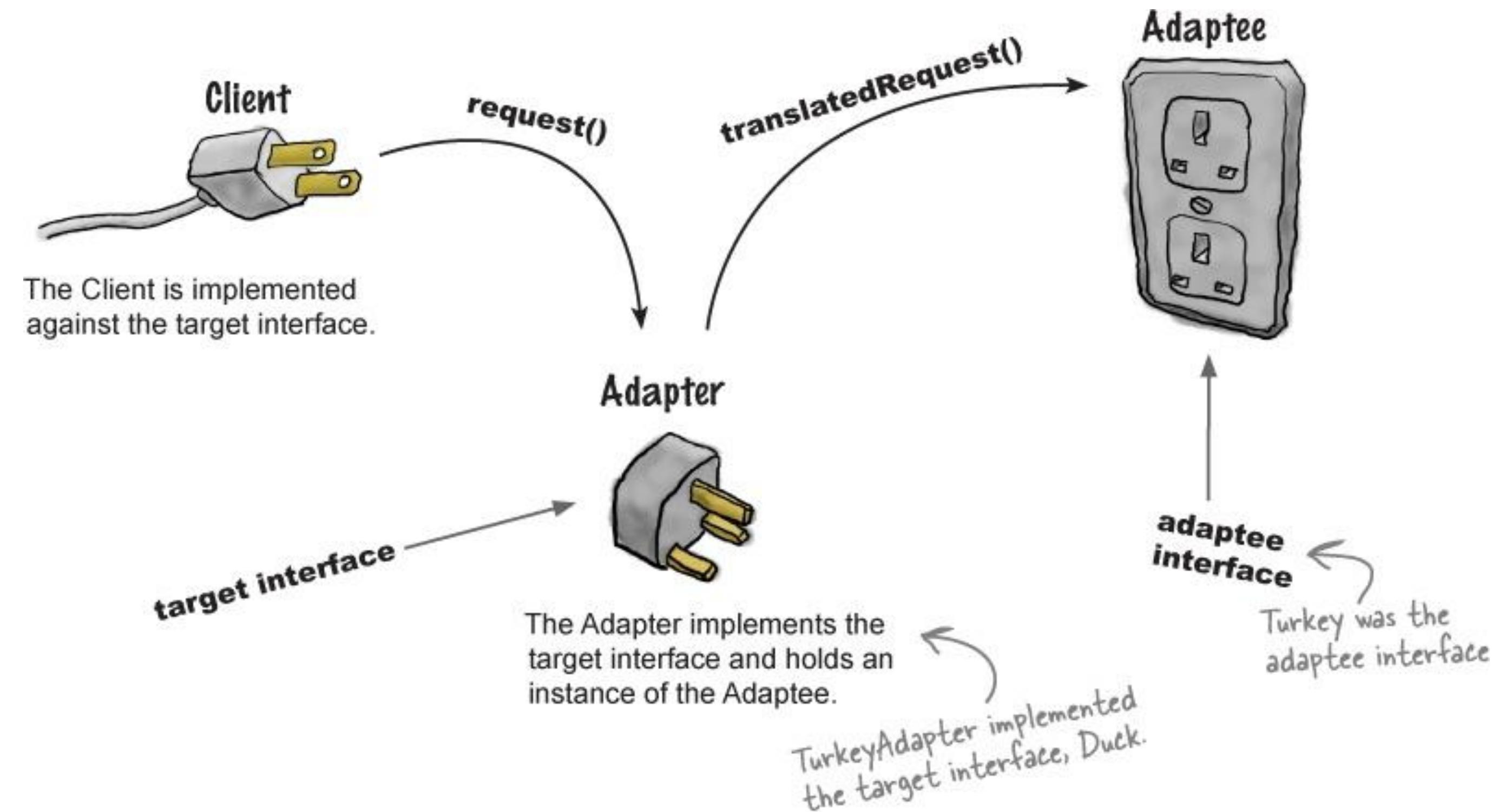


Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Participants

- Target
 - defines the domain-specific interface that Client uses
- Client
 - collaborates with objects conforming to the Target interface
- Adaptee
 - defines an existing interface that needs adapting
- Adapter
 - adapts the interface of Adaptee to the Target interface
 - involves a single class which is responsible to join functionalities of independent or incompatible interfaces

How the client uses the adapter



Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Client usage

- client makes a request to the adapter by calling a method on it using the target interface
- the adapter translates the request into one or more calls on the adaptee using the adaptee interface
- the client receives the results of the call and never knows there is an adapter doing the translation
- essentially, clients call operations on an Adapter instance
 - in turn, the adapter calls Adaptee operations that carry out the request

Summary

- adapters vary in the amount of work they do to adapt Adaptee to the Target interface
 - simple interface conversion (changing the names of operations to supporting an entirely different set of operations)
 - the amount of work the Adapter does depends on how similar the Target interface is to the Adaptee's
- adapter frees you from worrying about existing interfaces
- If I have a class that does what I need, I know that I can always use the adapter pattern to give it the correct interface

Object Adapter Example

Jason Fedin

Code

```
public interface Duck {  
    public void quack();  
    public void fly();  
}
```



This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

Code

- Here is a subclass of Duck, the MallardDuck

```
public class MallardDuck implements Duck {  
    public void quack() {  
        System.out.println("Quack");  
    }  
  
    public void fly() {  
        System.out.println("I'm flying");  
    }  
}
```

Simple implementations: the duck just prints out what it is doing.



Code

```
public interface Turkey {  
    public void gobble();  
    public void fly();  
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they
can only fly short distances.

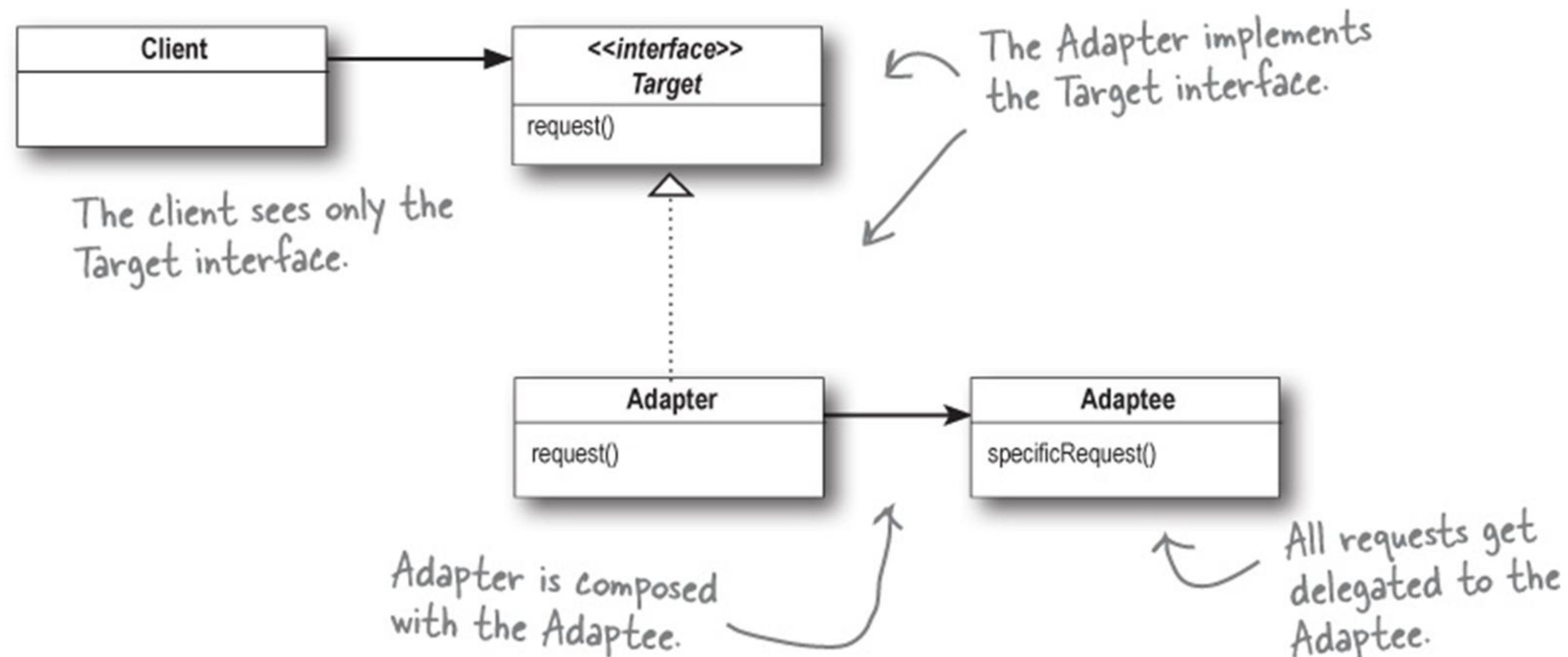
Code

```
public class WildTurkey implements Turkey {  
    public void gobble() {  
        System.out.println("Gobble gobble");  
    }  
  
    public void fly() {  
        System.out.println("I'm flying a short distance");  
    }  
}
```

Here's a concrete implementation
of Turkey; like Duck, it just
prints out its actions.



Code



Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Example (Turkey's adapted to Ducks)

- Target
 - Duck - interface that a WildTurkey uses to act like a duck
- Client
 - WildTurkey
 - incompatible with Duck
 - collaborates with TurkeyAdapter which conforms to Duck Interface
- Adaptee
 - Turkey - existing interface that needs adapting
- Adapter
 - TurkeyAdapter- adapts the Turkey to a Duck

Adapter

- let's say you are short on Duck objects and you would like to use some Turkey objects in their place
 - you can not use the turkeys outright because they have a different interface
- let's write an Adapter

Adapter Code

```
public class TurkeyAdapter implements Duck {  
    Turkey turkey;  
  
    public TurkeyAdapter(Turkey turkey) { ←  
        this.turkey = turkey;  
    }  
  
    public void quack() { ←  
        turkey.gobble();  
    }  
  
    public void fly() { ←  
        for(int i=0; i < 5; i++) {  
            turkey.fly();  
        }  
    }  
}
```

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Test Code

```
public class DuckTestDrive {  
    public static void main(String[] args) {  
        MallardDuck duck = new MallardDuck(); // Let's create a Duck...  
  
        WildTurkey turkey = new WildTurkey(); // ...and a Turkey.  
        Duck turkeyAdapter = new TurkeyAdapter(turkey); // And then wrap the turkey  
                                                        // in a TurkeyAdapter, which  
                                                        // makes it look like a Duck.  
  
        System.out.println("The Turkey says...");  
        turkey.gobble();  
        turkey.fly(); // Then, let's test the Turkey:  
                      // make it gobble, make it fly.  
  
        System.out.println("\nThe Duck says...");  
        testDuck(duck); // Now let's test the duck  
                        // by calling the testDuck()  
                        // method, which expects a  
                        // Duck object.  
  
        System.out.println("\nThe TurkeyAdapter says...");  
        testDuck(turkeyAdapter);  
    }  
  
    static void testDuck(Duck duck) {  
        duck.quack(); // Now the big test: we try to pass  
                      // off the turkey as a duck...  
        duck.fly();  
    }  
}
```

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

Output

Test run ↗

```
File Edit Window Help Don'tForgetToDuck
%java DuckTestDrive
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
```

↖ The Turkey gobbles and flies a short distance.

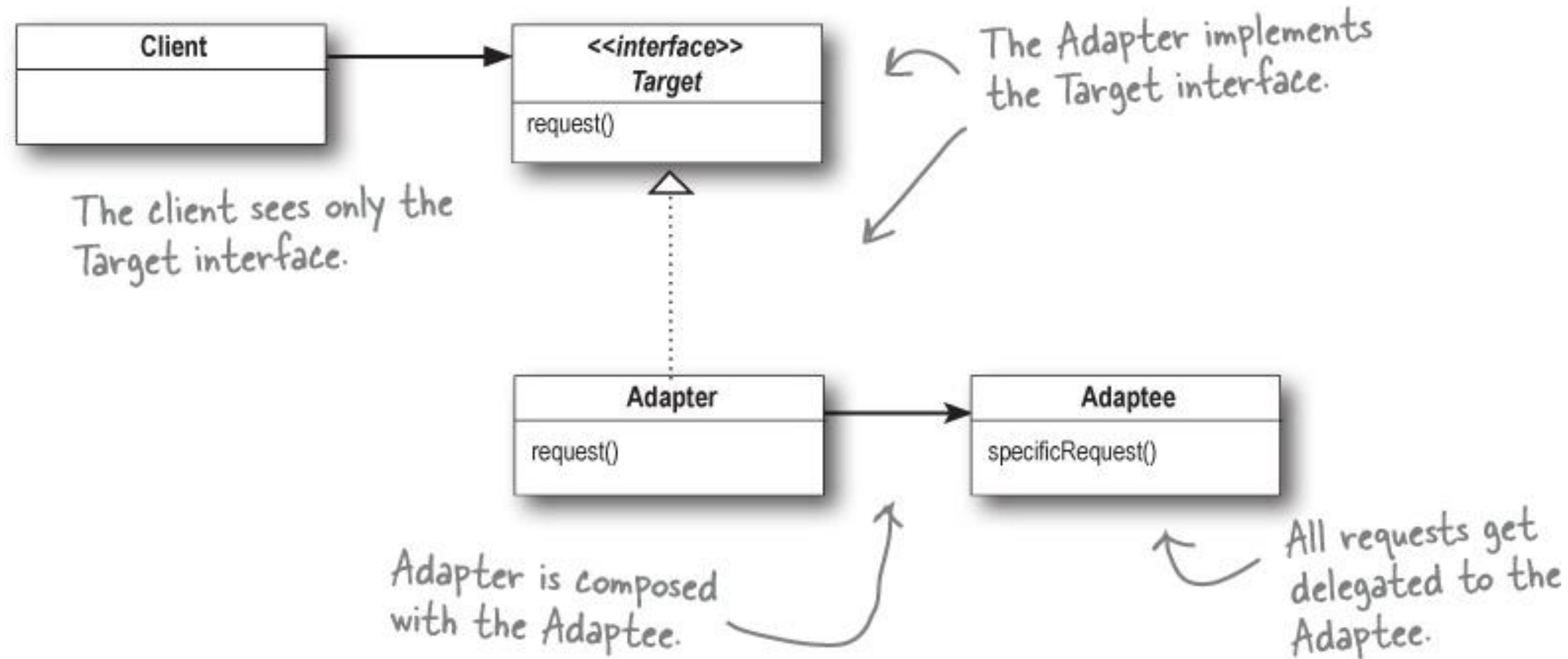
↖ The Duck quacks and flies just like you'd expect.

↖ And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

Object Adapter Example

Jason Fedin

Class Diagram



Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Overview

- I am going to demonstrate an application that uses an adapter to calculate the area of different shapes
 - rectangle and triangle
- in our current implementation, we can calculate the area of a rectangle easily
 - use a Calculator class and its getArea() method (takes a Rectangle Object as input)
- we cannot calculate the area of a triangle
 - the calculator interface is incompatible with a triangle
 - getArea() method does not take a triangle object as input
- we will create a Calculator/Triangle Adapter
 - will adapt the Calculator by having the adapter take a triangle as input
 - pass a triangle object to the adapter's constructor
 - translate the triangle member variable to a rectangle in the getArea method
 - call the getArea() of Calculator with null as input to get the area of the triangle
- from the user's point of view
 - seems to the user that he/she is passing a triangle to get the area of that triangle

Participants

- Target
 - CalculatorInterface – interface that a triangle uses to get its area
- Client
 - Triangle
 - incompatible with Calculator
 - collaborates with CalculatorAdapter which conforms to CalculatorInterface
- Adaptee
 - Calculator - existing interface that needs adapting
- Adapter
 - CalculatorAdapter – adapts the calculator to the CalculatorInterface

Code (Rectangle and Triangle)

```
class Rect {  
    public double l;  
    public double w;  
}
```

```
class Triangle {  
    public double b;//base  
    public double h;//height  
    public Triangle(int b, int h)  
    {  
        this.b = b;  
        this.h = h;  
    }  
}
```

Target – interface that client uses

```
interface CalculatorInterface  
{  
    // target interface  
    public double getArea(Rectangle r);  
}
```

Adaptee - existing interface that needs adapting

```
/*
Calculator can calculate the area of a rectangle.
To calculate the area we need a Rectangle input.
*/
```

```
class Calculator implements CalculatorInterface
{
    Rect rectangle;
    public double getArea(Rect r) {
        rectangle=r;
        return rectangle.l * rectangle.w;
    }
}
```

Code – CalculatorAdapter

```
/*Calculate the area of a Triangle using CalculatorAdapter. Please note here: this time our input is a Triangle.*/
class CalculatorAdapter implements CalculatorInterface {
    Calculator calculator;
    Triangle t;

    public CalculatorAdapter(Triangle myTriangle) {
        t = myTriangle;
    }

    public double getArea(Rectangle r) {
        calculator = new Calculator();
        Rect r = new Rect();
        //Area of Triangle=0.5*base*height
        r.l = t.b;
        r.w = 0.5*t.h;
        return calculator.getArea(r);
    }
}
```

Code - main

```
public class AdapterPattern
{
    public static void main(String[] args) {
        System.out.println("***Adapter Pattern Demo***");

        Triangle t = new Triangle(20,10);

        CalculatorInterface cal = new CalculatorAdapter(t);
        System.out.println("\nAdapter Pattern Example\n");

        System.out.println("Area of Triangle is :" + cal.getArea(null));
    }
}
```

Output



The screenshot shows a Java IDE's Console tab with the following output:

```
<terminated> AdapterPattern [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (May 20, 2015, 8:14:10 PM)
***Adapter Pattern Demo***

Adapter Pattern Example

Area of Triangle is :100.0
```

Class Adapter Implementation

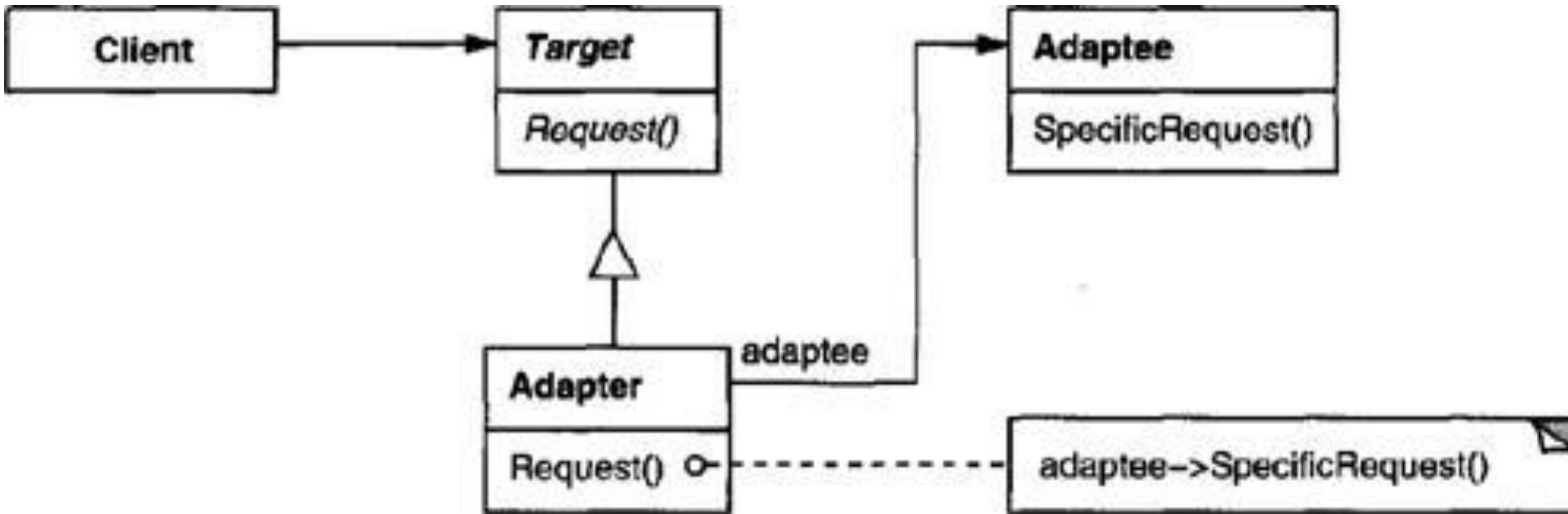
Jason Fedin

Overview

- we have defined the adapter pattern, however, there are actually two kinds of adapters
 - *object* adapters
 - what we have been studying up to this point in the class
 - relies on one object (the adapting object) containing another (the adapted object) (composition)
 - *class* adapters
 - another way to implement the adapter pattern (uses multiple inheritance)
- a class adapter is less of a focus in Java because you need multiple inheritance to implement it
 - not possible in Java
- that does not mean you might not encounter a need for class adapters down the road
 - can utilize interfaces as a workaround the lack of support for multiple inheritance in java

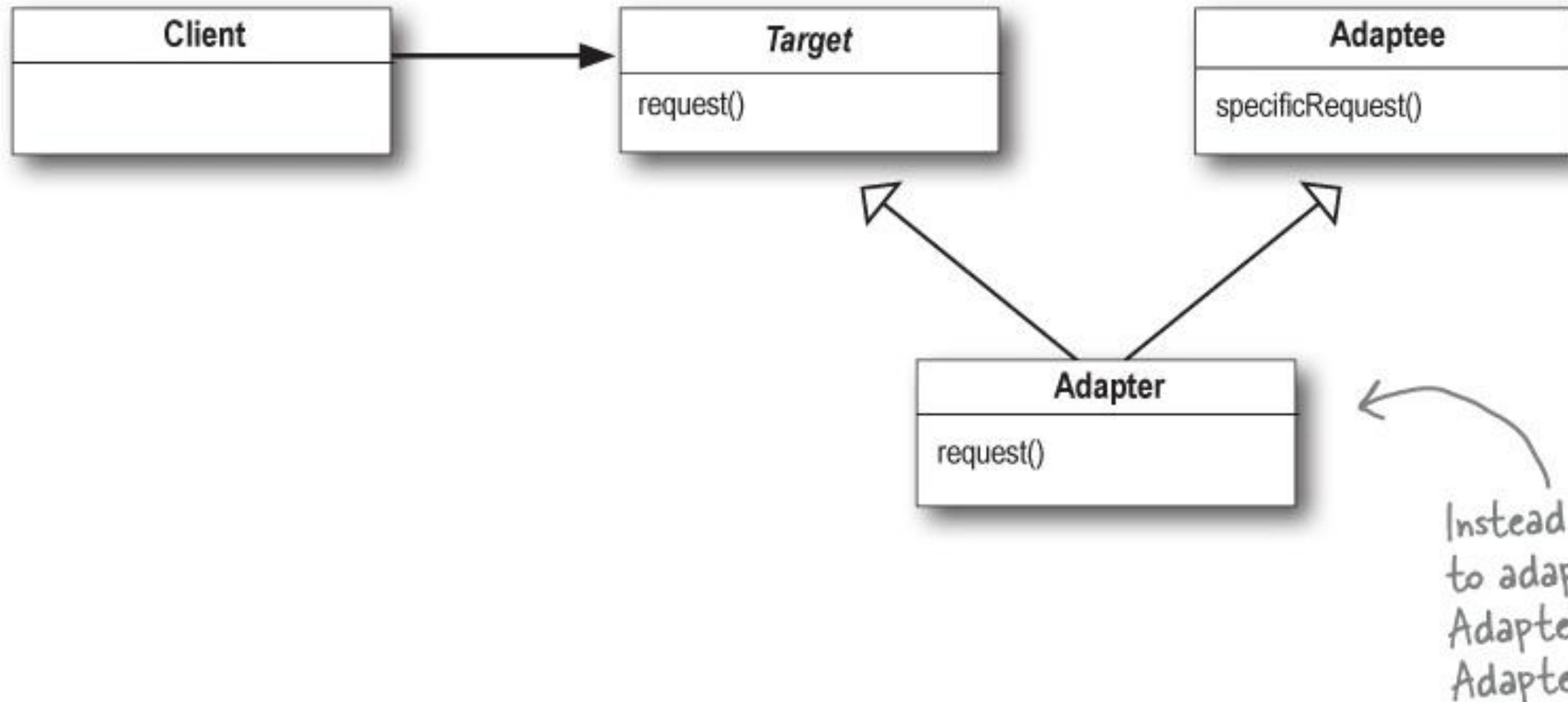
Object adapter review

- as a reminder, here is an object adapter
 - relies on object composition



Design Patterns: Elements of Reusable Object-Oriented Software By: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Class Adapter class diagram

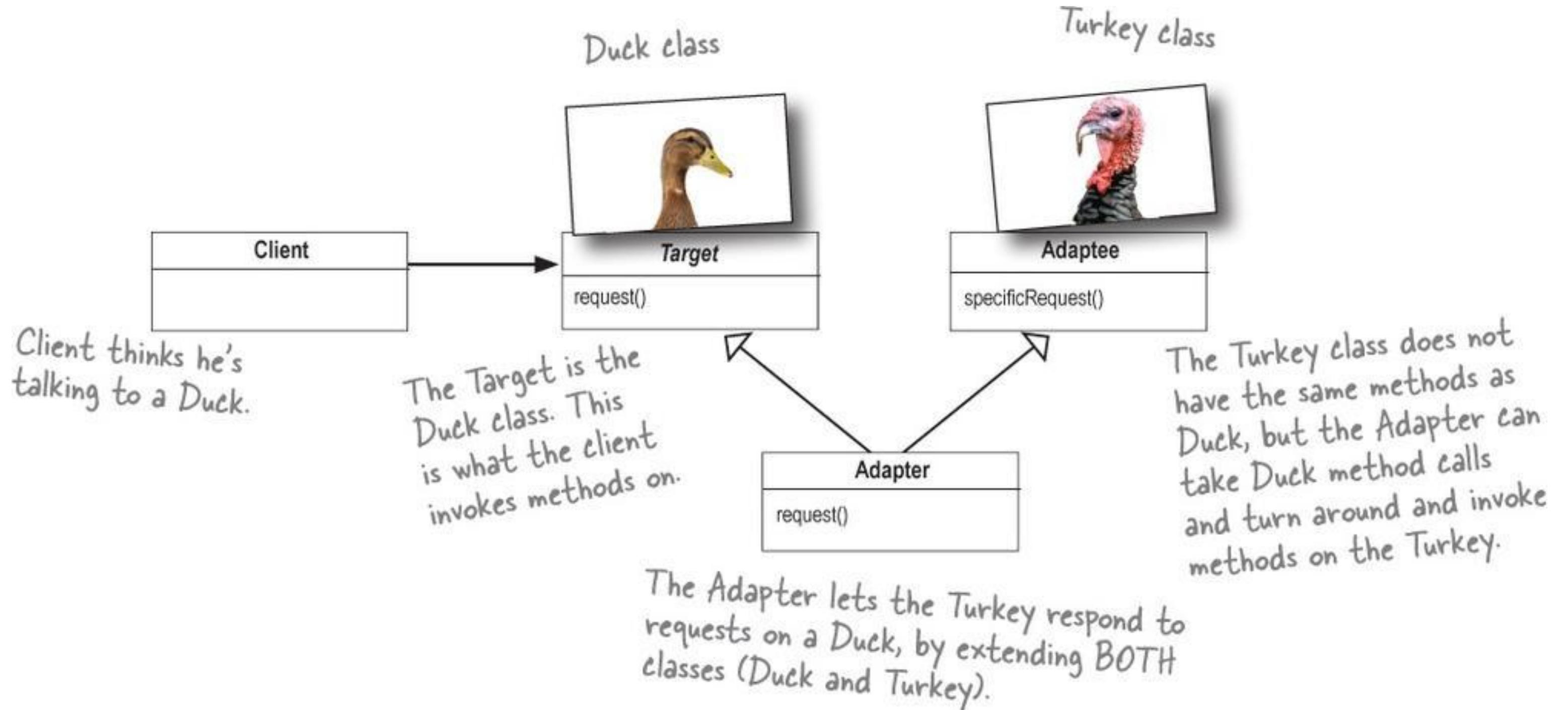


Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Class Adapters

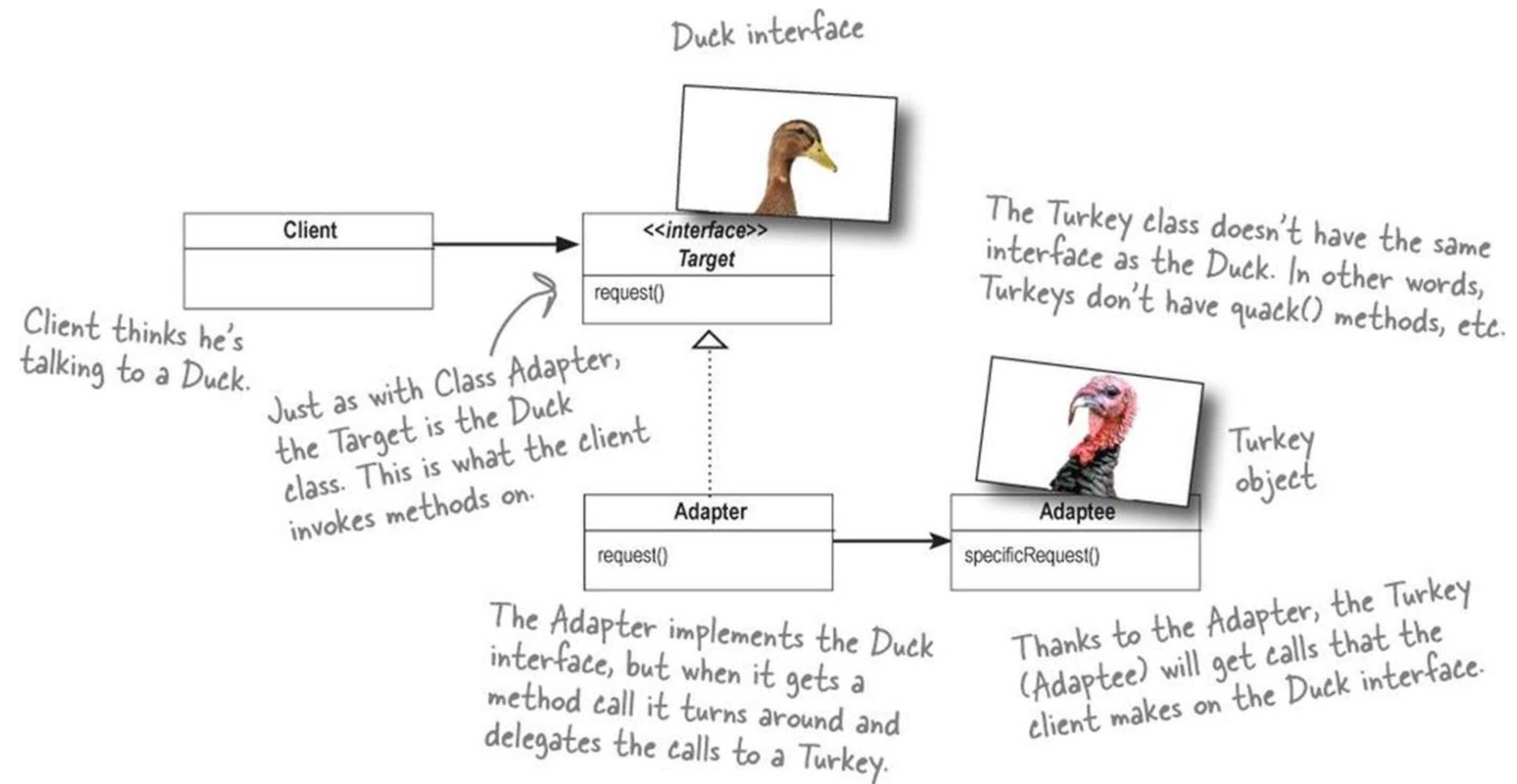
- a class adapter is very similar to an Object adapter
 - the class adapter will subclass the Target and the Adaptee
 - the object adapter will use composition to pass requests to an Adaptee
 - composition versus inheritance
- a Class Adapter works by creating a new class which subclasses publicly from an abstract class to define its interface
 - subclasses privately from our existing class to access its implementation
 - each wrapped method calls its associated, privately inherited method
- a class adapter adapts Adaptee to Target by committing to a concrete Adaptee class
 - will not work when we want to adapt a class and all its subclasses
- a class adapter lets the Adapter override some of the Adaptee's behavior since Adapter is a subclass of Adaptee

Class Adapter (would need to use interfaces in Java)



Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Object Adapter



Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

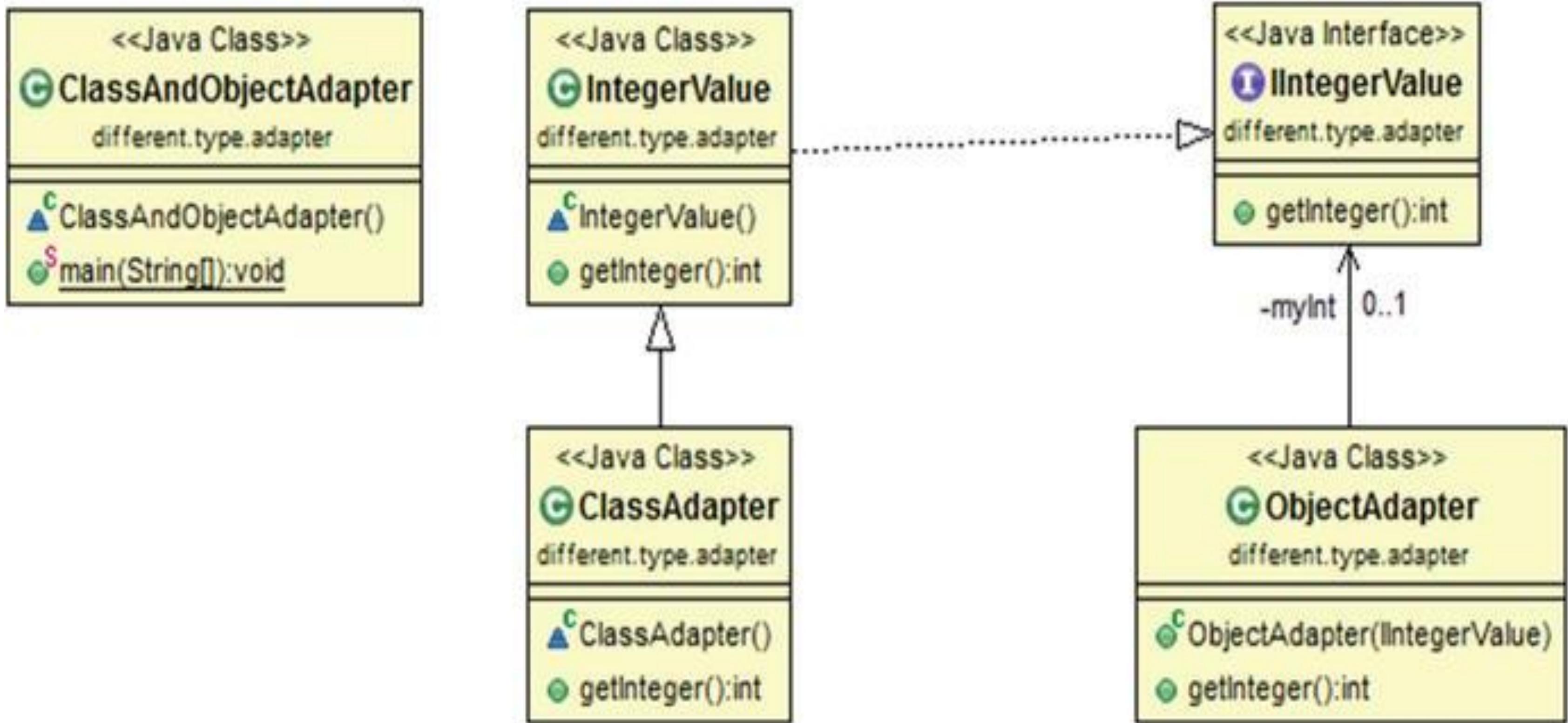
Summary

- an advantage of the object adapter is that it can adapt an adaptee class and any of its subclasses (because of composition)
 - a class adapter is committed to one specific adaptee class
- an advantage of a class adapter is that it does not need to re-implement its entire adaptee
 - can override the behavior of an adaptee if it needs to because it is just sub-classing
- the object adapter is more flexible
 - composition is also preferred over inheritance
 - write very little code to delegate to the adaptee
- a class adapter is more efficient
 - there is only one, not an adapter and an adaptee
- an object adapter can add new behavior and everything works with the adaptee class and all of its subclasses

Class Adapter Example

Jason Fedin

Example in intellij



Notes

- you can create a class adapter with an interface as long as you are only wrapping a single adaptee
- with multiple inheritance you could take two or more adaptees and wrap them into a single interface
- class Adapters are kind of possible in Java by using single inheritance

Code

```
interface IIntegerValue {  
    public int getInteger();  
}  
  
class IntegerValue implements IIntegerValue {  
    @Override  
    public int getInteger() {  
        return 5;  
    }  
}  
  
class ClassAdapter extends IntegerValue {  
    //Incrementing by 2  
    public int getInteger() {  
        return 2+super.getInteger();  
    }  
}
```

Code

```
class ObjectAdapter
{
    private IIntegerValue myInt;
    public ObjectAdapter(IIntegerValue myInt) {
        this.myInt=myInt;
    }

    //Incrementing by 2
    public int getInteger() {
        return 2+this.myInt.getInteger();
    }
}
```

Code

```
class ClassAndObjectAdapter
{
    public static void main(String args[])
    {
        System.out.println("***Class and Object Adapter Demo***");
        ClassAdapter ca1=new ClassAdapter();
        System.out.println("Class Adapter is returning :" +ca1.getInteger());

        ClassAdapter ca2=new ClassAdapter();
        ObjectAdapter oa=new ObjectAdapter(new IntegerValue());
        System.out.println("Object Adapter is returning :" +oa.getInteger());
    }
}
```

Output

```
Console X
<terminated> ClassAndObjectAdapter [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 17, 2015, 8:08:43 PM)
***Class and Object Adapter Demo***
Class Adapter is returning :7
Object Adapter is returning :7
```

(Demonstration) Adapter

Jason Fedin

Code

Step 1 - create interfaces for Media Player and Advanced Media Player
(MediaPlayer.java, AdvancedMediaPlayer.java)

```
public interface MediaPlayer {  
    public void play(String audioType, String fileName);  
}
```

```
public interface AdvancedMediaPlayer {  
    public void loadFilename(String fileName);  
    public void listen();  
}
```

Code

Step 2 - create concrete classes implementing the AdvancedMediaPlayer interface (VlcPlayer.java, Mp4Player.java)

```
public class VlcPlayer implements AdvancedMediaPlayer{
    String myFile;

    @Override
    public void loadFilename(String filename) {
        this.myFile = filename;
    }

    @Override
    public void listen() {
        System.out.println("Playing vlc file. Name: "+ myFile);
    }
}
```

Code (Mp4Player.java)

```
public class Mp4Player implements AdvancedMediaPlayer{
    String myFile;

    @Override
    public void loadFilename(String filename) {
        this.myFile = filename;
    }

    @Override
    public void listen() {
        System.out.println("Playing mp4 file. Name: "+ myFile);
    }
}
```

Code

Step 4 - create concrete class implementing the MediaPlayer interface (AudioPlayer.java)

```
public class AudioPlayer implements MediaPlayer {  
  
    @Override  
    public void play(String audioType, String fileName) {  
        if(audioType.equalsIgnoreCase("mp3")){  
            System.out.println("Playing mp3 file. Name: " + fileName);  
        }  
        else{  
            System.out.println("Invalid media. " + audioType + " format not supported");  
        }  
    }  
}
```

Code

Step 3 - Create adapter class implementing the MediaPlayer interface (AdvancedMediaPlayerAdapter.java)

```
public class AdvancedMediaPlayerAdapter implements MediaPlayer {  
  
    AdvancedMediaPlayer advancedMusicPlayer;  
  
    public AdvancedMediaPlayerAdapter(AdvancedMediaPlayer myMediaPlayer){  
        advancedMusicPlayer = myMediaPlayer;  
    }  
  
    @Override  
    public void play(String audioType, String fileName) {  
        advancedMusicPlayer.loadFilename(fileName);  
        advancedMusicPlayer.listen();  
    }  
}
```

Code

Step 5 - Use the MediaPlayer to play different types of audio formats (AdapterPatternDemo.java)

```
public class AdapterPatternDemo {  
    public static void main(String[] args) {  
        MediaPlayer audioPlayer = new AudioPlayer();  
        audioPlayer.play("mp3", "beyond the horizon.mp3");  
  
        AdvancedMediaPlayer mp4Player = new Mp4Player();  
        MediaPlayer advancedMediaPlayerAdapter1 = new AdvancedMediaPlayerAdapter(mp4Player);  
        advancedMediaPlayerAdapter1.play("mp4", "alone.mp4");  
  
        AdvandedMediaPlayer vlcPlayer = new VlcPlayer();  
        MediaPlayer advancedMediaPlayerAdapter2 = new AdvancedMediaPlayerAdapter(vlcPlayer);  
        advancedMediaPlayerAdapter2.play("vlc", "far far away.vlc");  
  
        advancedMediaPlayerAdapter2.play("vlc", "far far away.vlc");    }  
    }
```

Output

Step 6 - Verify the output

Playing mp3 file. Name: beyond the horizon.mp3

Playing mp4 file. Name: alone.mp4

Playing vlc file. Name: far far away.vlc

Invalid media. avi format not supported

Bridge Design Pattern

Jason Fedin

Overview

- the bridge pattern will decouple an abstraction from its implementation so that the two can vary independently
 - decouples implementation class and abstract class by providing a bridge structure between them
- we already understand the benefits of decoupling and abstraction
 - decoupling means to have things behave independently from each other
 - abstraction is how different things are related to each other conceptually (hiding details)
- implementations here mean the objects that the abstract class and its derivations use to implement themselves
 - not the derivations of the abstract class (concrete classes)
- this pattern helps us to make concrete class functionalities independent from the interface implementer class
 - can alter these different kind of classes structurally without affecting each other

Examples

- a standard software company will typically consist of the following two teams
 - development team
 - technical support team
- a change in the operational strategy of one team should not have a direct impact on the other team
- the technical support team plays the role of a bridge between the clients and the development team that implements the product
- another example would be in a GUI framework
 - separate window abstraction from window implementation

When to use the Bridge Pattern

- when you want to avoid a permanent binding between an abstraction and its implementation
 - when the implementation must be selected or switched at run-time
- when both the abstractions and their implementations should be extensible by subclassing
 - lets you combine the different abstractions and implementations and extend them independently
- when changes in the implementation of an abstraction should have no impact on clients
 - clients code should not have to be recompiled
- when you want to hide the implementation of an abstraction completely from clients
- when you have a ton of implementation classes
 - a class hierarchy indicates the need for splitting an object into two parts

Advantages of the Bridge Pattern

- decouples an implementation so that it is not bound permanently to an interface
- abstraction and implementation can be extended independently
 - allows you to vary the implementation and the abstraction by placing the two in separate class hierarchies
- changes to the concrete abstraction classes do not affect the client
- adds one more method level redirection to achieve the objective
- one drawback is that it does slightly increase complexity

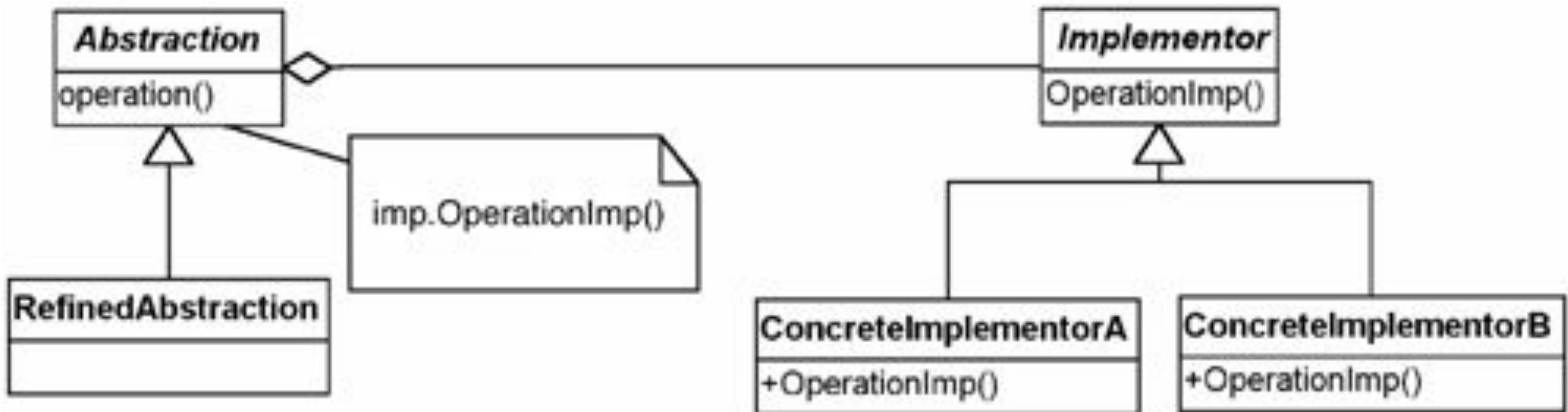
Compared to the Adapter

- the Adapter pattern is geared toward making unrelated classes work together
 - usually applied to systems after they have been designed
- in contrast, the Bridge is used up-front in a design
 - lets abstractions and implementations vary independently

Implementing the Bridge Pattern

Jason Fedin

Class Diagram



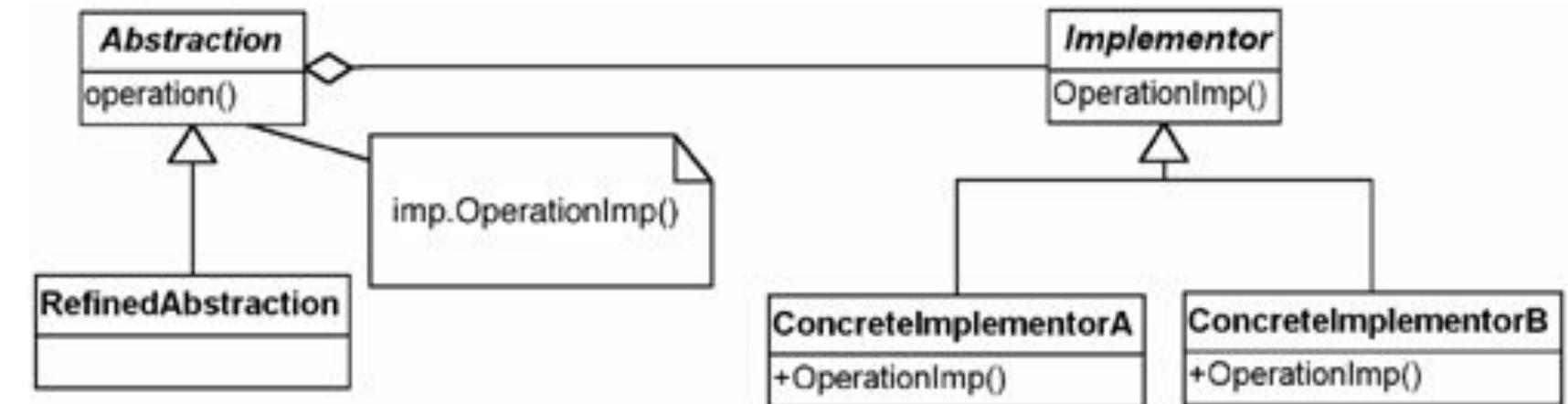
Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Overview

- there are 2 parts in the Bridge design pattern implementation
 - Abstraction - an interface or an abstract class
 - Implementation - an interface or abstract class
- allows the abstraction and the implementation to be developed independently
 - the client code can access only the abstraction part
 - client not concerned about the Implementation part
- the abstraction contains a reference to the implementer
- children of the abstraction are referred to as refined abstractions
- children of the implementer are concrete implementers
- since we can change the reference to the implementer in the abstraction, we are able to change the abstraction's implementer at run-time
 - changes to the implementer do not affect client code
 - increases the loose coupling between class abstraction and its implementation

Participants in detail

- Abstraction
 - core of the bridge design pattern and defines the crux.
 - defines the abstraction's interface
 - contains a reference to the implementer
- RefinedAbstraction
 - extends the abstraction takes the finer detail one level below
 - hides the finer elements from implementers
- Implementer
 - defines the interface for implementation classes
 - does not need to correspond directly to the abstraction interface and can be very different
 - Implementer provides only primitive operations
 - Abstraction defines higher-level operations based on these primitives
 - provides an implementation in terms of operations provided by Implementer interface
- Concrete Implementer
 - Implements the above implementer by providing concrete implementation



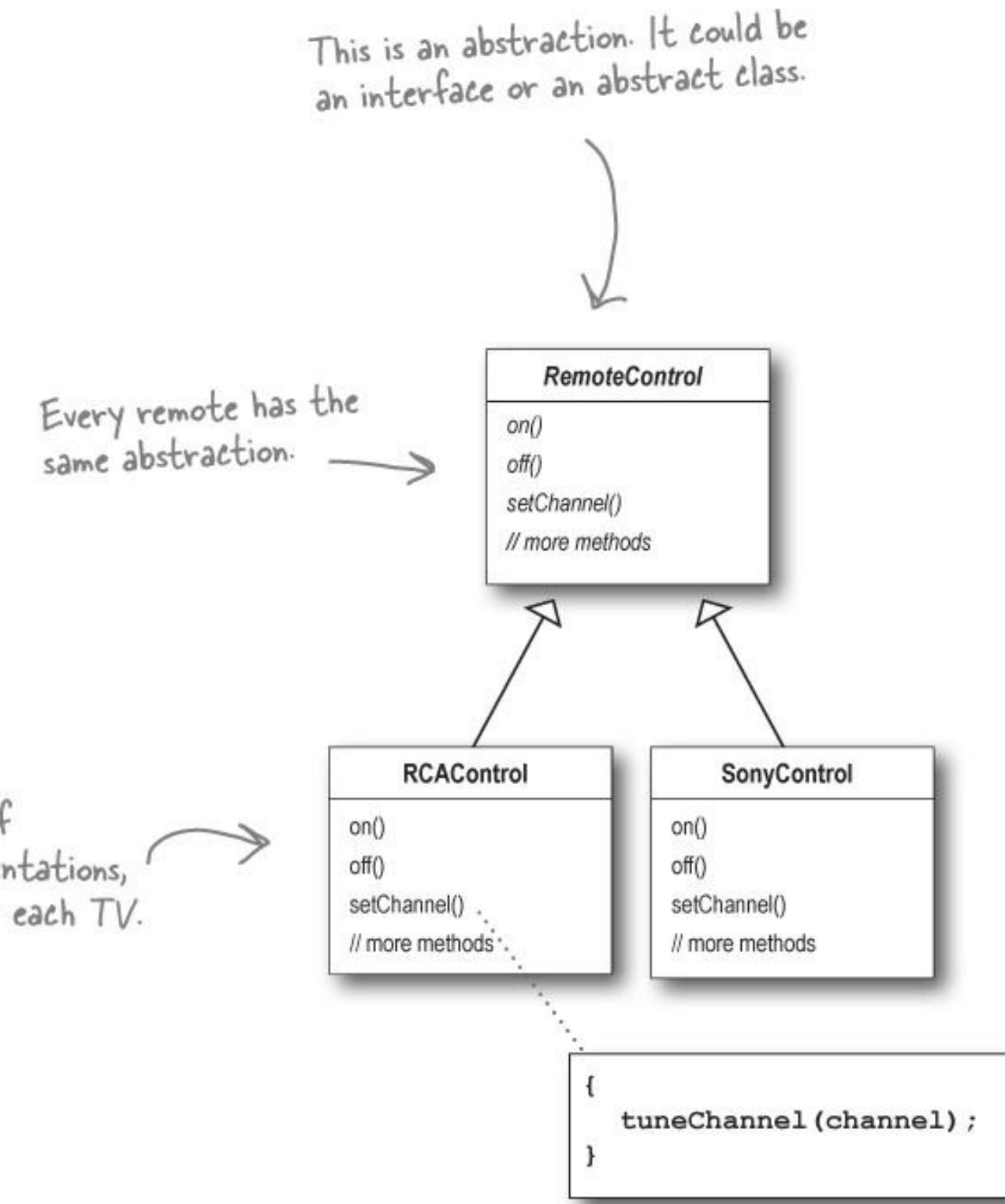
Composition/Aggregation over inheritance

- the implementation of bridge design pattern follows the notion to prefer Composition over inheritance
- when an abstraction can have one of several possible implementations
 - usually inheritance will be used to accommodate this
 - an abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways
- this approach is not always flexible enough
 - inheritance binds an implementation to the abstraction permanently
 - makes it difficult to modify, extend, and reuse abstractions and implementations independently
- the bridge pattern is an excellent example of following two of the mandates of the design pattern community
 - “Find what varies and encapsulate it”
 - “Favor aggregation over class inheritance”
- the Bridge pattern is one of the toughest patterns to understand in part because it is so powerful and applies to so many situations
 - also goes against a common tendency to handle special cases with inheritance

High Level Design Example

- imagine you're going to revolutionize "extreme lounging"
- you are writing the code for a new ergonomic and user-friendly remote control for TVs
- you already know that you have got to use good OO techniques
 - while the remote is based on the same abstraction, there will be lots of implementations
 - one for each model of TV

Class Diagram

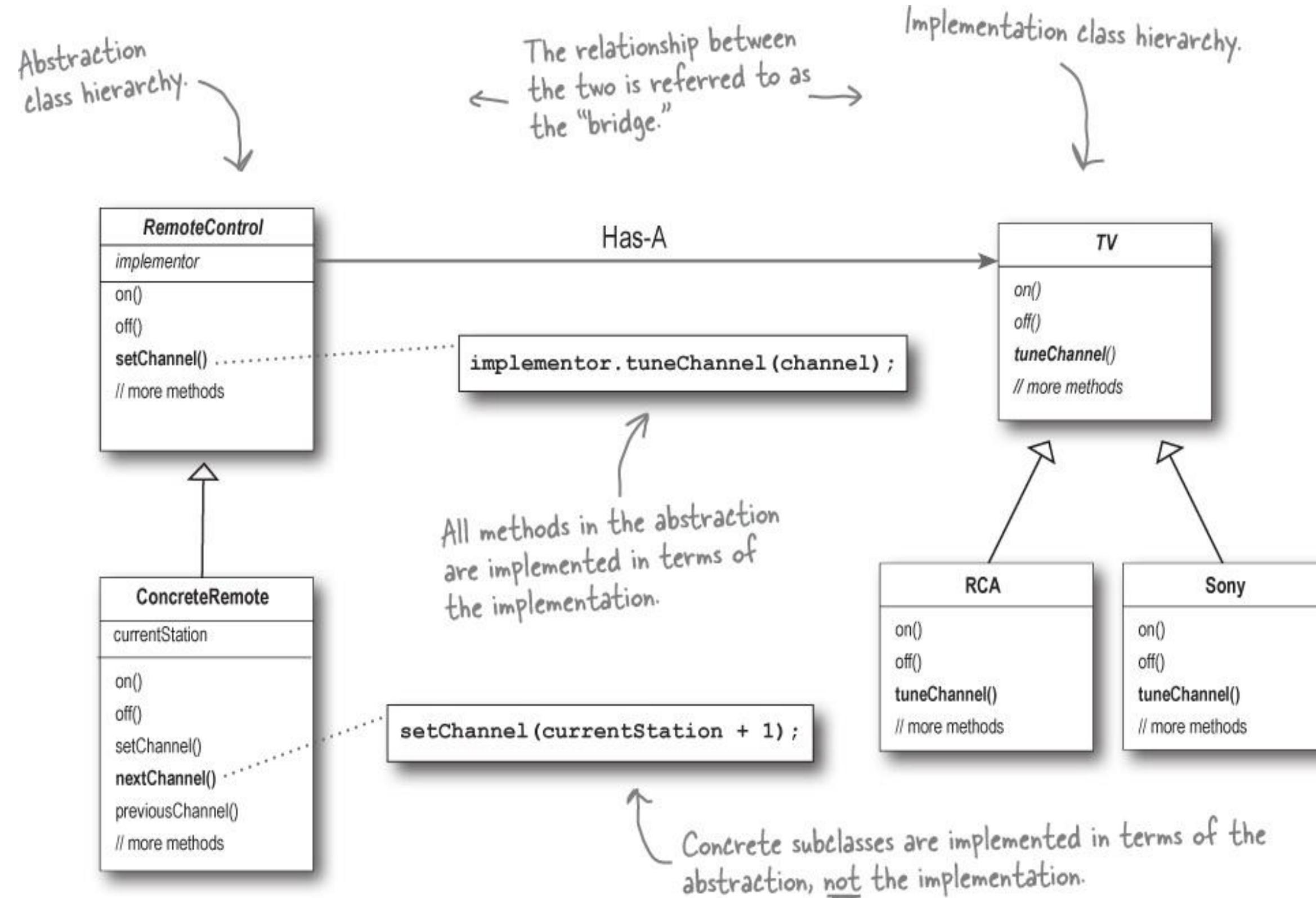


Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Problems with current design

- you know that the remote's user interface won't be right the first time
- you expect that the product will be refined many times as usability data is collected on the remote control
- your dilemma is that the remotes are going to change and the TVs are going to change
- you have already abstracted the user interface
 - you can vary the implementation over the many TVs your customers will own
- you are also going to need to vary the abstraction
 - it is going to change over time as the remote is improved based on the user feedback
- with the current design we can vary only the TV implementation, not the user interface
- the bridge pattern will allow you to vary the implementation and the abstraction

Bridge Pattern Design



Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Bridge Pattern Design

- now we have two hierarchies
 - one for the remotes
 - one for platform-specific TV implementations
- the bridge allows us to vary either side of the two hierarchies independently

Summary

- pattern is extremely helpful when our class and its associated functionalities may change in frequent intervals
- we remove the concrete binding between an abstraction and the corresponding implementation
 - both hierarchies (abstraction and its implementations) can extend through child classes
 - both hierarchies can grow independently
 - if we make any change in abstraction methods, they do not have an impact on the implementer method
- the abstraction and implementer do not need to be abstract classes
 - can be interfaces
- the abstraction contains the reference to its implementer
- you can change the implementers dynamically (at runtime) by changing the reference in the abstraction

Example in intelliJ

Jason Fedin

Steps

- in this example, we are producing and assembling the two different vehicles using Bridge design pattern
- We will first create bridge implementer interface. (Workshop.java)
- Followed by creating concrete bridge implementer classes implementing the WorkShop interface. (Produce, Assemble)
- We then create an abstract class Vehicle using the Workshop interface
- We follow that up with creating concrete classes implementing the Vehicle interface. (Car and Bike)
- Lastly, we use the Vehicle and Workshop classes to manufacture different vehicles

Code

- Step 1 - Create bridge implementer interface. (Workshop.java)

```
// Implementer for bridge pattern
interface Workshop
{
    abstract public void work();
}
```

Code

Step 2 - Create concrete bridge implementer classes implementing the WorkShop interface. (Produce.java, Assemble.java)

```
class Produce implements Workshop {  
    @Override  
    public void work() {  
        System.out.print("Produced");  
    }  
}
```

```
class Assemble implements Workshop {  
    @Override  
    public void work() {  
        System.out.print(" And");  
        System.out.println(" Assembled.");  
    }  
}
```

Code

- Step 3 - Create an abstract class Vehicle using the Workshop interface (Vehicle.java)

```
// abstraction in bridge pattern
abstract class Vehicle {
    protected Workshop workShop1;
    protected Workshop workShop2;

    protected Vehicle(Workshop workShop1, Workshop workShop2) {
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
    }

    abstract public void manufacture();
}
```

Code

- Step 4 - Create concrete class implementing the Vehicle interface. (Car.java)

```
// Refine abstraction 1 in bridge pattern
class Car extends Vehicle {
    public Car(Workshop workShop1, Workshop workShop2) {
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture() {
        System.out.print("Car ");
        workShop1.work();
        workShop2.work();
    }
}
```

Code

Step 4 - Create concrete class implementing the Vehicle interface. (Bike.java)

```
// Refine abstraction 2 in bridge pattern
class Bike extends Vehicle {
    public Bike(Workshop workShop1, Workshop workShop2) {
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture() {
        System.out.print("Bike ");
        workShop1.work();
        workShop2.work();
    }
}
```

Code

- Step 5 - Use the Vehicle and Workshop classes to manufacture different vehicles

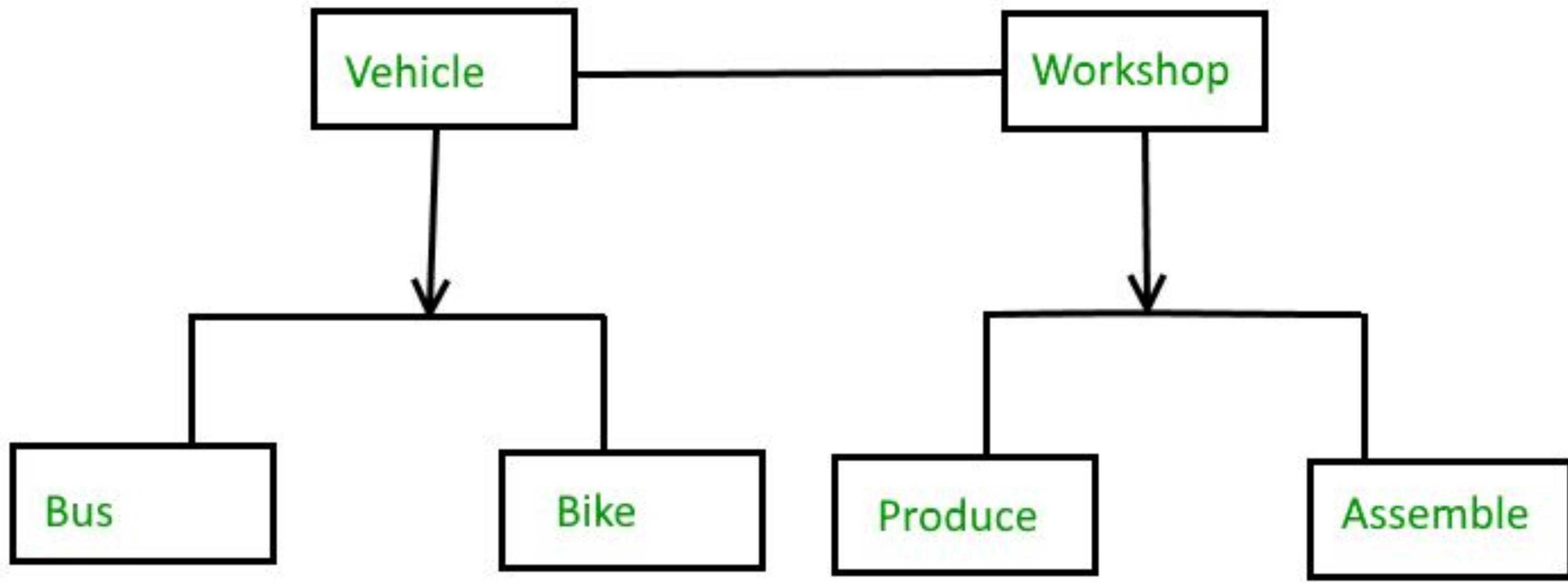
```
// Demonstration of bridge design pattern
class BridgePattern {
    public static void main(String[] args)
    {
        Vehicle vehicle1 = new Car(new Produce(), new Assemble());
        vehicle1.manufacture();
        Vehicle vehicle2 = new Bike(new Produce(), new Assemble());
        vehicle2.manufacture();
    }
}
```

Output

Car Produced And Assembled.

Bike Produced And Assembled.

Ending Design

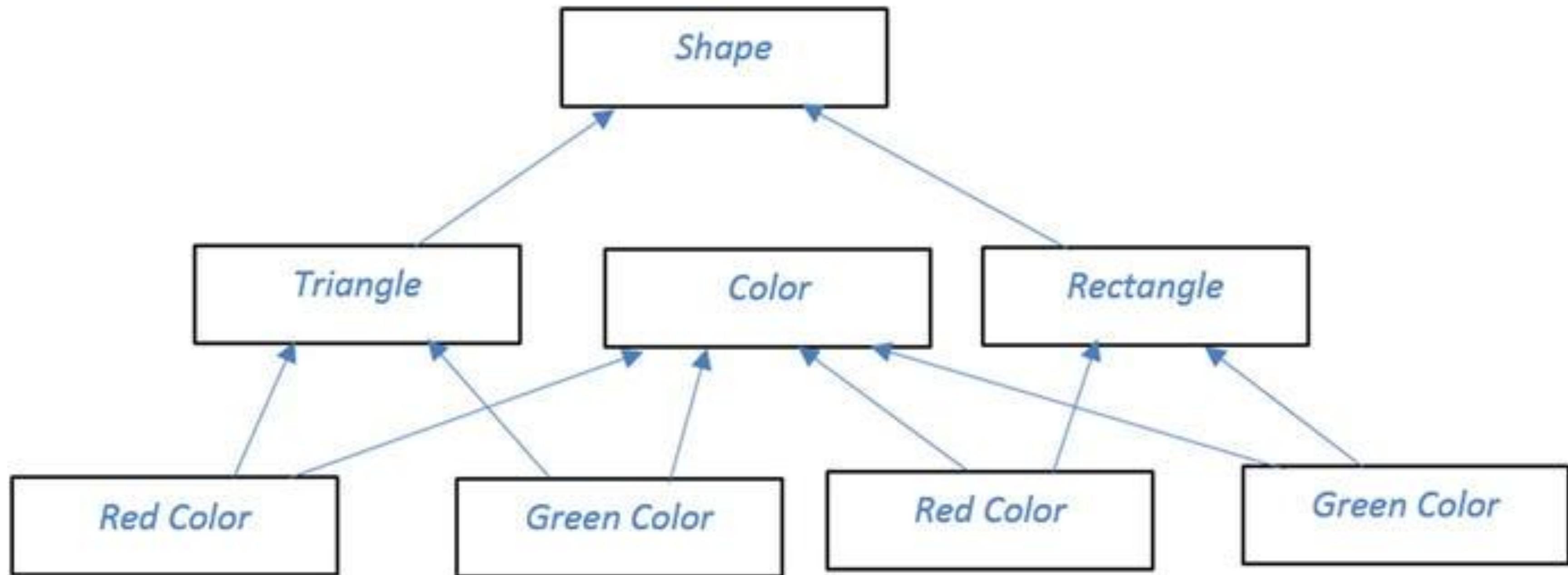


<https://www.geeksforgeeks.org/bridge-design-pattern/>

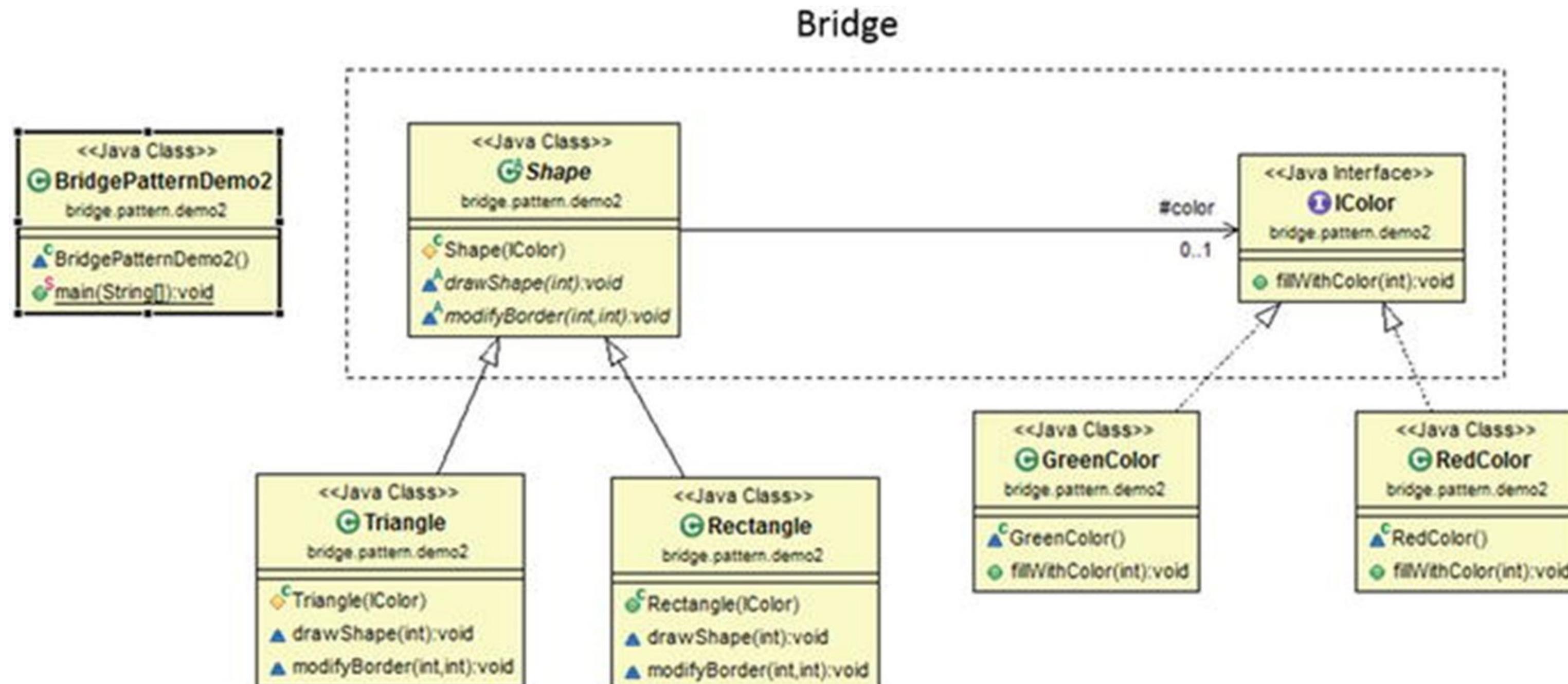
Demonstration

Jason Fedin

Started with this Design



Class Diagram



Overview

- the resulting implementation includes an abstraction-specific and an implementer-specific method to represent the power and usefulness of this pattern
- we can draw a triangle and a rectangle with a particular color with the implementer-specific method `drawShape()`
- we can change the thickness of the border by the abstraction-specific method `modifyBorder()`

Steps

- we will first create the bridge implementer interface (Icolor.java)
- the second step is to create concrete bridge implementer classes implementing the Icolor interface (GreenColor, RedColor)
- step 3 will be to create an abstract class Shape using the Icolor interface
- step 4 consists of creating concrete classes implementing the Shape interface (Triangle and Rectangle)
- the last step will be to use the Shape and Icolor classes to draw different colored shapes

First step - create the bridge implementer interface (IColor.java)

```
//Colors-The Implementer  
interface IColor  
{  
    void fillWithColor(int border);  
}
```

Step 2 - create concrete bridge implementer classes

```
class RedColor implements IColor {  
    @Override  
    public void fillWithColor(int border) {  
        System.out.print("Red color with " +border+" inch border");  
    }  
}  
  
class GreenColor implements IColor {  
    @Override  
    public void fillWithColor(int border) {  
        System.out.print("Green color with " +border+" inch border.");  
    }  
}
```

create an abstract class Shape using the IColor interface

```
//Shapes-The Abstraction
```

```
abstract class Shape
{
    //Composition
    protected IColor color;
    protected Shape(IColor c)
    {
        this.color = c;
    }
    abstract void drawShape(int border);
    abstract void modifyBorder(int border,int increment);
}
```

create concrete classes implementing the Shape interface

```
class Triangle extends Shape {  
    protected Triangle(IColor c){  
        super(c);  
    }  
  
    //Implementer-specific method  
    @Override  
    void drawShape(int border) {  
        System.out.print(" This Triangle is colored with: ");  
        color.fillWithColor(border);  
    }  
  
    //Abstraction-specific method  
    @Override  
    void modifyBorder(int border,int increment) {  
        System.out.println("\nNow we are changing the border length "+increment+ " times");  
        border=border*increment;  
        drawShape(border);  
    }  
}
```

create concrete classes implementing the Shape interface

```
class Rectangle extends Shape {  
    public Rectangle(IColor c) {  
        super(c);  
    }  
  
    //Implementer-specific method  
    @Override  
    void drawShape(int border) {  
        System.out.print(" This Rectangle is colored with: ");  
        color.fillWithColor(border);  
    }  
    //Abstraction-specific method  
    @Override  
    void modifyBorder(int border,int increment) {  
        System.out.println("\n Now we are changing the border length "+increment+ " times");  
        border=border*increment;  
        drawShape(border);  
    }  
}
```

use the Shape and Icolor classes

```
class BridgePatternEx {  
    public static void main(String[] args) {  
        System.out.println("*****BRIDGE PATTERN*****");  
        //Coloring Green to Triangle  
        System.out.println("\nColoring Triangle:");  
        IColor green = new GreenColor();  
        Shape triangleShape = new Triangle(green);  
        triangleShape.drawShape(20);  
        triangleShape.modifyBorder(20, 3);  
  
        //Coloring Red to Rectangle  
        System.out.println("\n\nColoring Rectangle :");  
        IColor red = new RedColor();  
        Shape rectangleShape = new Rectangle(red);  
        rectangleShape.drawShape(50);  
        //Modifying the border length twice  
        rectangleShape.modifyBorder(50,2);  
    }  
}
```

Output

```
Console X
<terminated> BridgePatternDemo2 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 19, 2015, 1:29:15 PM)
*****BRIDGE PATTERN*****

Coloring Triangle:
This Triangle colored with: Green color with 20 inch border.
Now we are changing the border length 3 times
This Triangle colored with: Green color with 60 inch border.

Coloring Rectangle :
This Rectangle colored with: Red color with 50 inch border
Now we are changing the border length 2 times
This Rectangle colored with: Red color with 100 inch border
```

Composite Design Pattern

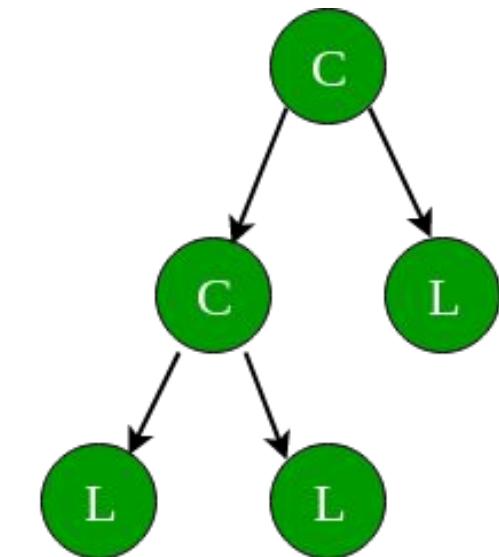
Jason Fedin

Overview

- the composite design pattern composes objects into tree structures to represent part-whole hierarchies
 - lets clients treat individual objects and compositions of objects uniformly
- a composite is an object designed as a composition of one-or-more similar objects that all exhibit similar functionality
 - i.e.
 - a group of objects that is treated the same way as a single instance of the same type of object
- when we have many objects with common functionalities we create a composite object
 - creates a class that contains a group of its own objects

Overview (cont'd)

- allows us to build structures of objects in the form of trees
 - contains both compositions of objects and individual objects as nodes/leaves
 - we ask each node in the tree structure to perform a task
- using a composite structure, we can apply the same operations over both composites and individual objects
 - we can ignore the differences between compositions of objects and individual objects
- programmers often have to understand the difference between a leaf-node and a branch when dealing with tree-structured data
 - makes code more complex, and therefore, more error prone
 - the composite pattern solves this by providing an interface that allows treating complex and primitive objects uniformly
- the main purpose of the composite is to allow you to manipulate a single instance of the object just as you would manipulate a group of them
 - the operations you can perform on all the composite objects often have a least common denominator relationship



Where, C = Composite & L = Leaf

Examples

- any organization that has many departments with each department having many employees to serve would utilize a composite pattern
 - all are employees of the organization
 - groupings of employees could create a department
 - those departments ultimately can be grouped together to build the whole organization (hierarchy)
- any tree structure in computer science follows the same concept as the composite
- graphics applications like drawing editors and schematic capture systems
 - let users build complex diagrams out of simple components
 - user can group components to form larger components

Examples

- nested groups of menus and menu items in a GUI
 - by putting menus and items in the same structure we create a part-whole hierarchy
 - a tree of objects that is made of parts (menus and menu items)
 - can be treated as a whole, like one big huge menu
- we can use this pattern to treat this “huge” menu as “individual objects and compositions uniformly”
- we have a tree structure of menus, submenus, and perhaps sub submenus along with menu items
 - any menu is a “composition” because it can contain both other menus and menu items
 - individual objects are just the menu items
- using a design that follows the Composite Pattern would allow you to write the same operation (like printing!) over the entire menu structure

When to use the Composite

- when clients need to ignore the difference between compositions of objects and individual objects
 - using multiple objects in the same way with nearly identical code to handle them
 - less complex in this situation to treat primitives and composites as the same
- when you are worried about memory usage
 - less number of objects reduces the memory usage
 - keep you away from errors related to memory like `java.lang.OutOfMemoryError`
- when efficiency is a concern
 - creating an object in Java is really fast, however, we can still reduce the execution time of our program by sharing objects
- when you are forced to maintain child ordering
 - parse trees as components
 - we need to take special care to maintain that order

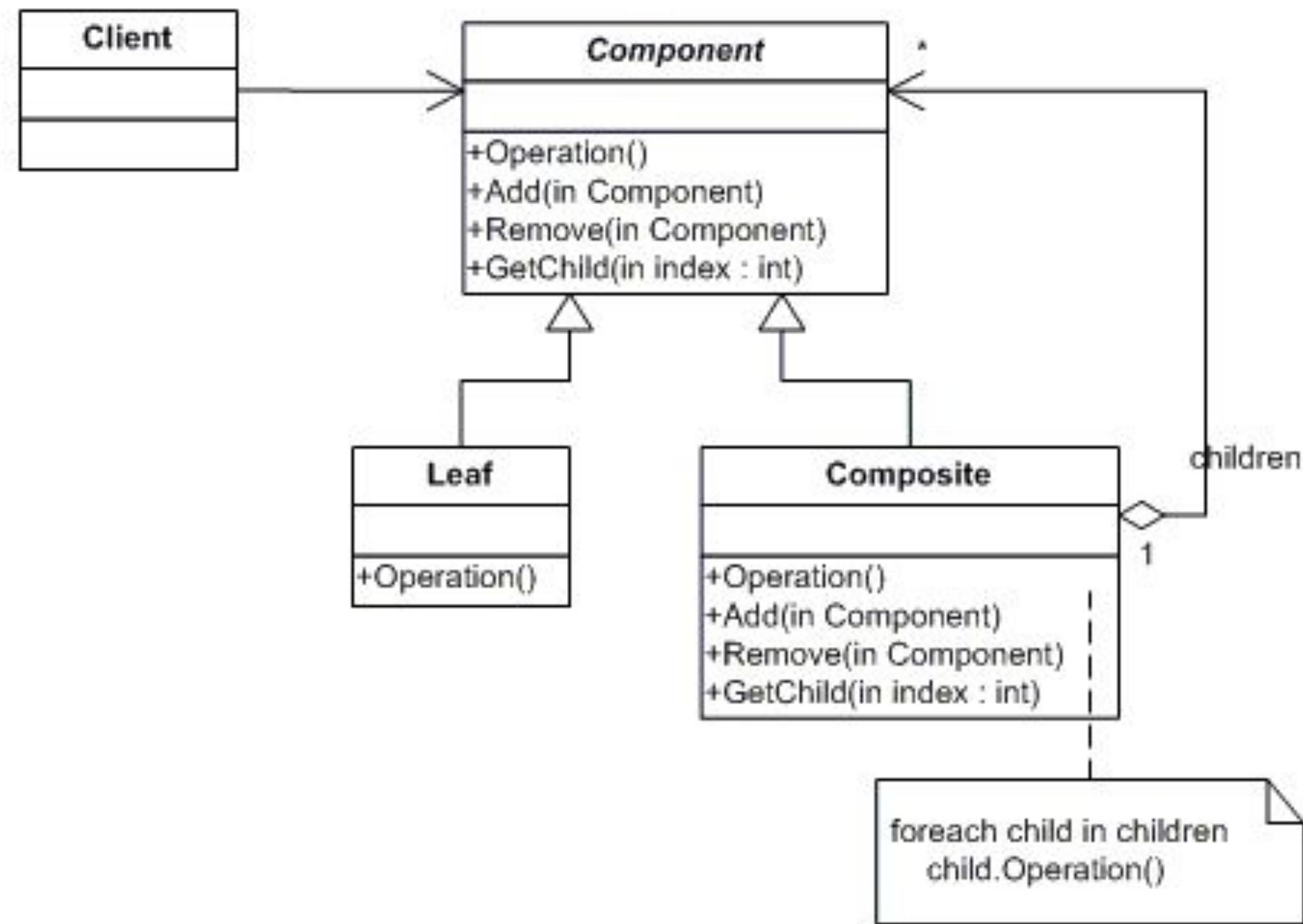
Benefits

- makes it easy to add new kinds of components
- makes clients simpler
 - they do not have to know if they are dealing with a leaf or a composite component
- one drawback is that it makes it harder to restrict the type of components of a composite
 - you cannot rely on the type system to enforce constraints for you
 - have to use run-time checks instead

Implementing the Composite

Jason Fedin

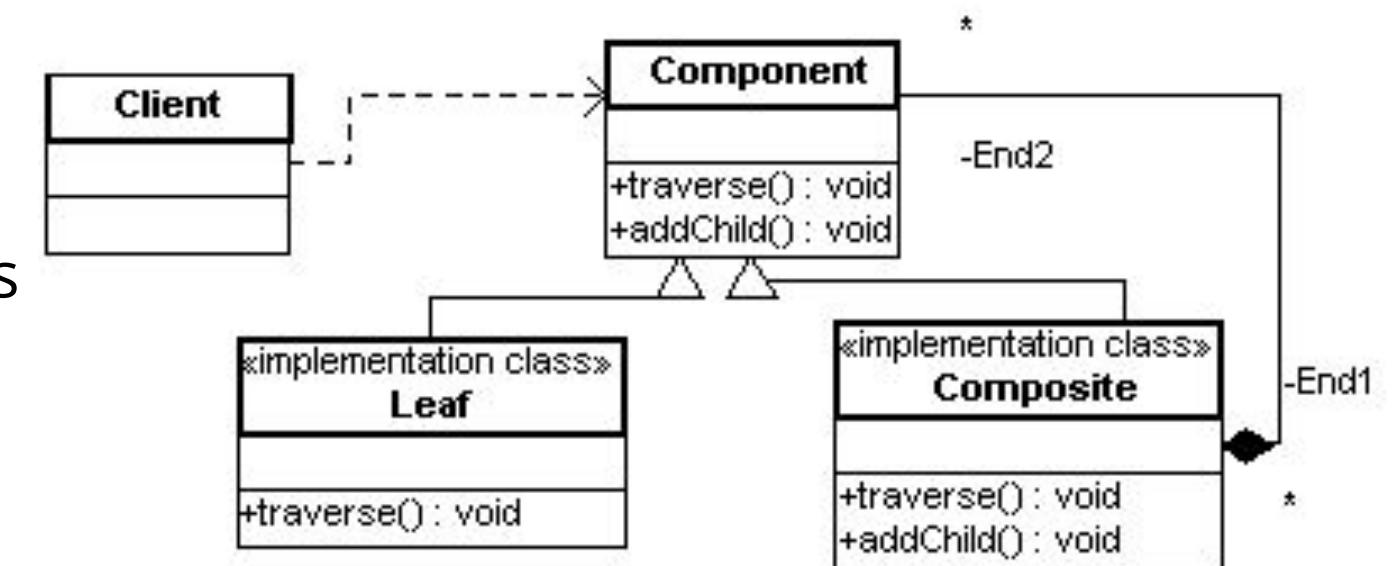
Class Diagram



<https://howtodoinjava.com/design-patterns/structural/composite-design-pattern/>

Participants

- Component
 - declares the interface for objects in the composition
 - implements default behavior for the interface common to all classes
 - declares an interface for accessing and managing its child components
- Leaf
 - represents leaf objects in the composition
 - a leaf has no children
 - defines behavior for primitive objects in the composition
- Composite
 - defines behavior for components having children (add, remove, etc)
 - stores child components (some data structure, list??)
 - implements child-related operations in the Component interface
- Client
 - manipulates objects in the composition through the Component interface (does the grouping)



Usage

- a client uses the component class interface to interact with objects in the composition structure
- if recipient is a leaf then request is handled directly
- If recipient is a composite, then it usually forwards request to its child components
 - also may perform additional operations before and after forwarding

Tradeoff

- this pattern focuses on transparency and does not strictly follow the Single Responsibility design principle
- whether an element is a composite or leaf node is transparent to the client
 - the Component interface is allowed to contain the child management operations and the leaf operations (add, remove, and shared operation)
 - allows a client to treat both composites and leaf nodes uniformly
 - violates the single responsibility principle
 - we have both types of operations in the Component class
 - we lose a bit of safety because a client might try to do something inappropriate or meaningless on an element (add or remove on a leaf object itself instead of the composite)

Tradeoff (cont'd)

- this is a design decision
- we could take the design in the other direction and separate out the responsibilities into interfaces
 - makes our design more safe
 - any inappropriate calls on elements would be caught at compile time or runtime
 - we would lose transparency and our code would have to use conditionals and the instanceof operator
- this is a classic case of a tradeoff
- we are guided by design principles, but we always need to observe the effect they have on our designs
- sometimes we purposely do things in a way that seems to violate a principle
 - however, this is a matter of perspective
 - it might seem incorrect to have child management operations in the leaf nodes (like add(), remove() and getChild())
 - then again you can always shift your perspective and see a leaf as a node with zero children

Summary

- defines class hierarchies consisting of primitive objects and composite objects
- primitive objects can be composed into more complex objects
 - wherever client code expects a primitive object, it can also take a composite object
- clients can treat composite structures and individual objects uniformly
 - clients do not know whether they are dealing with a leaf or a composite component
 - simplifies client code, because it avoids having to write case-statement-style functions over the classes that define the composition
- newly defined composite or leaf subclasses work automatically with existing structures and client code
 - clients do not have to be changed for new component classes

Composite example in intellij

Jason Fedin

Overview

- lets look at an example concerning a company and its employee organization hierarchy
- we have general managers and under general managers, there can be managers and under managers there can be developers
- you can set a tree structure and ask each node (composite and leaf) to perform a common operation like displaying all of its details (name, position, emplId)
- we can have our client group objects by directory (engineering directory, company directory, etc)

Create the component (Employee.java)

- declares the interface for objects in the composition
- implements default behavior for the interface common to all classes
- declares an interface for accessing and managing its child components (moved to composite class, since leafs do not need this (add, remove), could have included these here with no ops)

```
public interface Employee
{
    public void showEmployeeDetails();
}
```

Create each leaf node (Developer and Manager)

- represents leaf objects in the composition
 - a leaf has no children
- defines behavior for primitive objects in the composition

```
public class Developer implements Employee {  
    private String name;  
    private long empld;  
    private String position;  
  
    public Developer(long empld, String name, String position) {  
        this.empld = empld;  
        this.name = name;  
        this.position = position;  
    }  
  
    @Override  
    public void showEmployeeDetails() {  
        System.out.println(empld+" "+name+);  
    }  
  
    // a bunch of other developer only methods, getSoftwareSkillSet, etc  
}
```

Create Manager leaf node

```
public class Manager implements Employee {  
    private String name;  
    private long empId;  
    private String position;  
  
    public Manager(long empId, String name, String position) {  
        this.empId = empId;  
        this.name = name;  
        this.position = position;  
    }  
  
    @Override  
    public void showEmployeeDetails() {  
        System.out.println(empId+" " +name);  
    }  
  
    // a bunch of other Manager only methods (setReviewPeriod, checkVacationTimeLeftForEmployee, etc)  
}
```

Create the composite (Directory)

- defines behavior for components having children
- stores child components
- implements child-related operations in the Component interface (we are also only defining them here instead of Employee interface)

// Class used to get Employee List and do the operations like add or remove Employee

```
import java.util.ArrayList;  
import java.util.List;  
  
public class Directory implements Employee {  
    private List<Employee> employeeList = new ArrayList<Employee>();  
  
    @Override  
    public void showEmployeeDetails() {  
        for(Employee emp:employeeList) {  
            emp.showEmployeeDetails();  
        }  
    }  
}
```

Create the composite (Directory) (cont'd)

```
public void addEmployee(Employee emp)
{
    employeeList.add(emp);
}
```

```
public void removeEmployee(Employee emp)
{
    employeeList.remove(emp);
}
```

Create the client (Company)

- manipulates objects in the composition through the Component interface (organizes the grouping)

```
public class Company {  
    public static void main (String[] args) {  
        Employee dev1 = new Developer(100, "Lokesh Sharma", "Pro Developer");  
        Employee dev2 = new Developer(101, "Vinay Sharma", "Developer");  
        Directory engDirectory = new Directory();  
        engDirectory.addEmployee(dev1);  
        engDirectory.addEmployee(dev2);
```

Create the client (Company) (cont'd)

```
Employee man1 = new Manager(200, "Kushagra Garg", "SEO Manager");
Employee man2 = new Manager(201, "Vikram Sharma ", "Kushagra's Manager");
```

```
Directory accDirectory = new Directory();
accDirectory.addEmployee(man1);
accDirectory.addEmployee(man2);
```

```
Directory companyDirectory = new Directory();
companyDirectory.addEmployee(engDirectory);
companyDirectory.addEmployee(accDirectory);
companyDirectory.showEmployeeDetails();
```

```
}
```

```
}
```

Output

100 Lokesh Sharma Pro Developer

101 Vinay Sharma Developer

200 Kushagra Garg SEO Manager

201 Vikram Sharma Kushagra's Manager

Another example with a different grouping

- lets modify the previous example by creating a composite for grouping a hierarchy
- we will keep the component interface the same
- however, we will move the manager leaf class and make it a composite class

Our new composite class (Manager.java)

```
import java.util.ArrayList;  
import java.util.List;  
  
public class Manager implements Employee {  
    private List<Employee> employeeList = new ArrayList<Employee>();  
    private String name;  
    private double salary;  
  
    public Manager(String name,double salary){  
        this.name = name;  
        this.salary = salary;  
    }  
  
    public void addEmployee(Employee emp) {  
        employeeList.add(emp);  
    }  
}
```

Our new composite class (Manager.java)

```
public void removeEmployee(Employee emp) {  
    employeeList.remove(emp);  
}  
  
public Employee getChild(int i) {  
    return employeeList.get(i);  
}  
  
public String getName() {  
    return name;  
}  
  
public double getSalary() {  
    return salary;  
}
```

Our new composite class (Manager.java)

```
public void showEmployeeDetails() {  
    System.out.println("-----");  
    System.out.println("Name =" + getName());  
    System.out.println("Salary =" + getSalary());  
    System.out.println("-----");  
  
    Iterator<Employee> employeeIterator = employeeList.iterator();  
    while(employeeIterator.hasNext()){  
        Employee employee = employeeIterator.next();  
        employee.showEmployeeDetails();  
    }  
}  
  
public void remove(Employee employee) {  
    employeeList.remove(employee);  
}
```

Our new client (grouping by hierarchy)

```
public class Hierarchy{  
    public static void main (String[] args) {  
        Employee dev1 = new Developer(100, "Lokesh Sharma", "Pro Developer");  
        Employee dev2 = new Developer(101, "Vinay Sharma", "Developer");  
  
        Manager manager1=new Manager("Daniel",25000);  
        manager1.add(dev1 );  
        manager1.add(dev2 );  
  
        Employee dev3=new Developer(102, "Michael", "Developer");  
  
        Manager generalManager=new Manager("Mark", 50000);  
        generalManager.add(dev3);  
        generalManager.add(manager1);  
        generalManager.print();  
    }  
}
```

Output

Name =Mark

Salary =50000.0

102 Michael Developer

Name =Daniel

Salary =25000.0

100 Lokesh Sharma Pro Developer

101 Vinay Sharma Developer

Demonstrating Composite

Jason Fedin

Steps – Identify and create the component

```
import java.util.*;  
interface Faculty  
{  
    public String getDetails();  
}
```

Identify and create the leaf (Professor)

```
class Professor implements Faculty {  
    private String mName;  
    private String mPosition;  
    private int mOfficeNum;  
  
    Professor(String name, String position, int officeNum) {  
        mName = name;  
        mPosition = position;  
        mOfficeNum = officeNum;  
    }  
  
    @Override  
    public String getDetails() {  
        return (mName + " is the " + mPosition);  
    }  
}
```

Steps – Identify and create the composite (Supervisor)

```
class Supervisor implements Faculty {  
    private String name;  
    private String deptName;  
    private List<Faculty> myFacultyList;  
  
    Supervisor(String name, String deptName) {  
        this.name = name;  
        this.deptName = deptName;  
        myFacultyList= new ArrayList<Faculty>();  
    }  
}
```

Steps – Identify and create the composite (Supervisor)

```
public void add(Faculty professor) {  
    myFacultyList.add(professor);  
}  
  
public void remove(Faculty professor) {  
    myFacultyList.remove(professor);  
}  
  
public List<Faculty> getMyFaculty() {  
    return myFacultyList;  
}  
  
@Override  
public String getDetails() {  
    return (name + " is the " + deptName);  
}  
}
```

Steps – Create the client

```
class CompositePatternEx
{
    public static void main(String[] args)
    {
        Supervisor technologyDean = new Supervisor("Dr. S.Som", "Dean of Technology");

        Supervisor chairOfMathDepartment = new Supervisor("Mrs.S.Das", "Chair of Math Department");

        Supervisor chairOfComputerScienceDepartment = new Supervisor("Mr. V.Sarcar", "Chair of Computer Science
Department");

        Professor mathProf1 = new Professor("Math Professor1", "Adjunct", 302);
        Professor mathProf2 = new Professor("Math Professor2", "Associate", 303);

        Professor cseProf1 = new Professor ("CSE Professor1", "Adjunct", 507);
        Professor cseProf2 = new Professor ("CSE Professor2", "Professor", 508);
        Professor cseProf3 = new Professor ("CSE Professor3", "Professor", 509);
```

Steps – Create the client

```
technologyDean.add(chairOfMathDepartment );
technologyDean.add(chairOfComputerScienceDepartment );

/* Professors of Mathematics directly reports to chair of math*/
chairOfMathDepartment.add(mathProf1);
chairOfMathDepartment.add(mathProf2);

/*Professors of Computer Sc. directly reports to chair of computer science*/
chairOfComputerScienceDepartment.add(cseProf1);
chairOfComputerScienceDepartment.add(cseProf2);
chairOfComputerScienceDepartment.add(cseProf3);

//Printing the details
System.out.println("***COMPOSITE PATTERN DEMO ***");
System.out.println("\nThe college has the following structure\n");
System.out.println(technologyDean.getDetails());
List<Faculty> chairs=technologyDean.getMyFaculty();
```

Steps – Create the client

```
for(int i=0;i<chairs.size();i++)  
{  
    System.out.println("\t"+chairs.get(i).getDetails());  
}
```

```
List<Faculty> mathProfessors= chairOfMathDepartment.getMyFaculty();  
for(int i=0;i<mathProfessors.size();i++)  
{  
    System.out.println("\t\t"+ mathProfessors.get(i).getDetails());  
}
```

```
List<Faculty> cseProfessors= chairOfComputerScienceDepartment.getMyFaculty();  
for(int i=0;i<cseProfessors.size();i++)  
{  
    System.out.println("\t\t"+cseProfessors.get(i).getDetails());  
}
```

Steps – Create the client

```
//One computer professor is leaving  
chairOfComputerScienceDepartment.remove(cseProf2);  
System.out.println("\n After CSE Professor2 leaving the organization- CSE department has following faculty:");  
cseProfessors = chairOfComputerScienceDepartment.getMyFaculty();  
for(int i=0;i< cseProfessors.size();i++)  
{  
    System.out.println("\t\t"+ cseProfessors.get(i).getDetails());  
}  
}  
}
```

Output

***COMPOSITE PATTERN DEMO ***

The college has the following structure

Dr. S.Som is the Dean of Technology

Mrs.S.Das is the Chair of Math Department

Mr. V.Sarcar is the Chair of Computer Science Department

Math Professor1 is the Adjunct

Math Professor2 is the Associate

CSE Professor1 is the Adjunct

CSE Professor2 is the Professor

CSE Professor3 is the Professor

After CSE Professor2 leaving the organization- CSE department has following faculty:

CSE Professor1 is the Adjunct

CSE Professor3 is the Professor

Decorator Design Pattern

Jason Fedin

Overview

- the decorator pattern will allow you to attach additional responsibilities to an object dynamically
 - allows a user to add new functionality to an existing object without altering its structure
- decorators provide a flexible alternative to sub-classing for extending functionality
- the main principle of this pattern says that we cannot modify existing functionalities but we can extend them
 - open for extension but closed for modification
- the core concept applies when we want to add some specific functionalities to some specific object instead of to the whole class
- decorator is used to modify the functionality of an object at runtime
 - other instances of the same class will not be affected by this, so individual object gets the modified behavior

Why not use inheritance?

- with the decorator, we want to add responsibilities to individual objects, not to an entire class
- one way to add responsibilities is with inheritance
 - we need to create a new class for new responsibilities
 - there will be many classes inside the system, which in turn can make the system complex
- inheriting a border from another class puts a border around every subclass instance
 - inflexible because the choice of border is made statically
 - a client can not control how and when to decorate the component with a border
- a more flexible approach is to enclose the component in another object that adds the border
 - the enclosing object is called the decorator
 - decorator conforms to the interface of the component it decorates
- we can add or remove responsibilities by simply attaching or detaching decorators

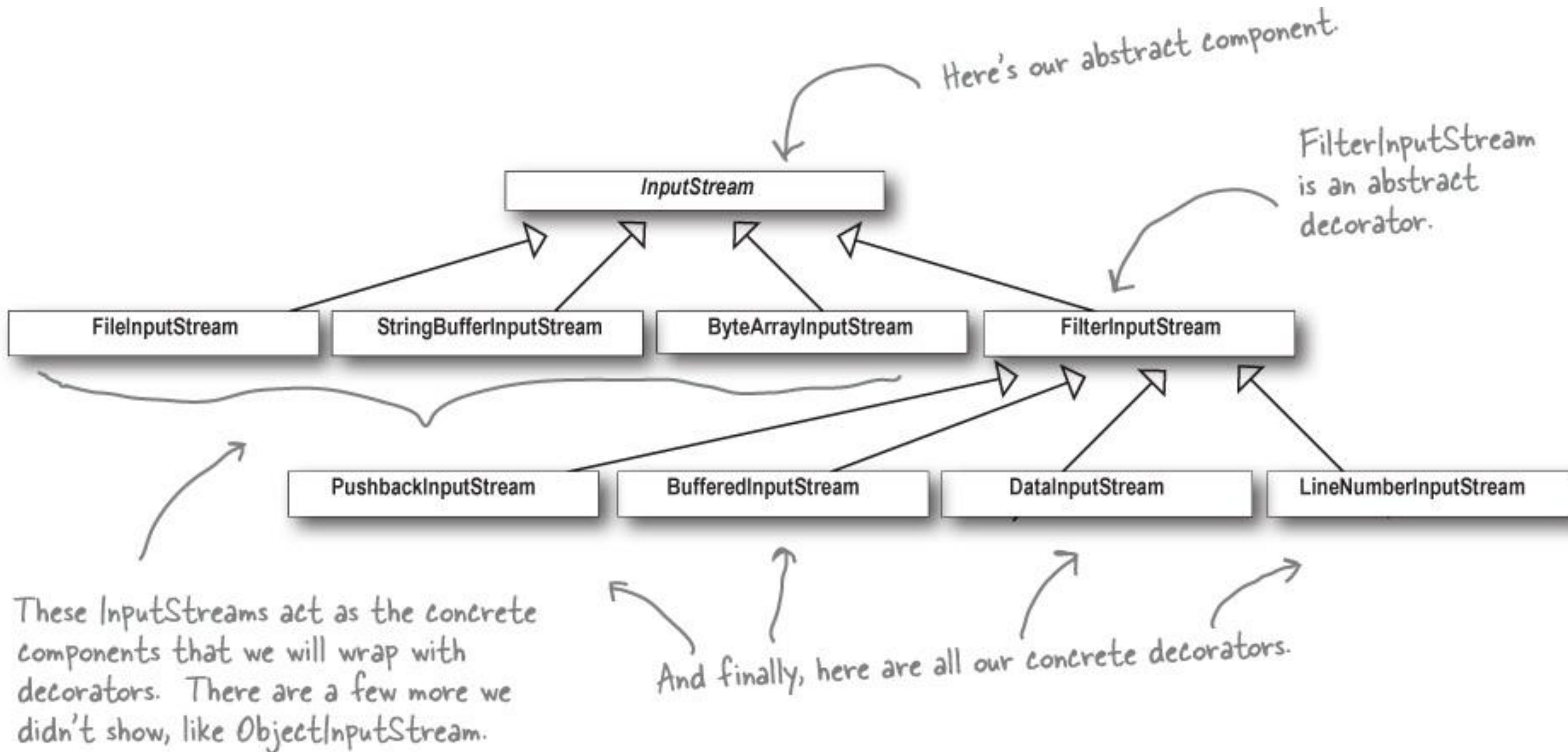
Examples

- suppose you already own a house
 - you want to add an additional floor
 - you do not want to change the architecture of ground floor (or existing floors)
- you want to change the design of the architecture for the newly added floor without affecting the existing architecture for existing floor(s)
 - use a decorator
- many object-oriented user interface toolkits use decorators to add graphical additions to widgets
 - you want to add properties like borders or behaviors like scrolling to any user interface component
 - do not want to change the core functionality of the graphical user interface
- suppose we have a TextView object that displays text in a window
 - textView has no scroll bars by default because we might not always need them
 - when we need scroll bars, we can use a ScrollDecorator to add them

Java i.o. packages example

- streams are a fundamental abstraction in most I/O facilities
 - can provide an interface for converting objects into a sequence of bytes or characters
 - lets us transcribe an object to a file or to a string in memory for retrieval later
- the large number of classes in the java.io package is largely based on using a decorator
 - use decorators to add functionality for reading/writing data from/to a file
- for example, BufferedInputStream and LineNumberInputStream both extend FilterInputStream
 - FilterInputStream acts as an abstract decorator class

Java i.o. packages example



Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

advantages and drawbacks

- the biggest advantage of using this pattern is that we can add new functionality to a particular object without disturbing existing objects in the system
- more flexibility than static inheritance
 - responsibilities can be added and removed at run-time simply by attaching and detaching them
- decorators also make it easy to add a property twice
 - to give a TextView a double border, simply attach two BorderDecorators
- we can code incrementally
 - we make a simple class first and then one by one we can add decorator objects to them as needed
 - we do not need to take care of each and every possible scenario in the beginning
- one drawback may be that designs using this pattern often result in a large number of small classes that can be overwhelming
 - lots of little objects can be hard to learn and debug

Decorator as a structural pattern??

- many developers believe that the decorator pattern should be classified as a behavioral pattern
- structural patterns describe how classes and objects are composed to create new structures or new functionality
- the Decorator pattern allows you to compose objects by wrapping one object with another to provide new functionality
- the focus is on how you compose the objects dynamically to gain functionality
- the focus is NOT on the communication and interconnection between objects (behavioral patterns)

When to use a decorator

- when you want to add responsibilities to individual objects dynamically and transparently without affecting other objects
- when extension by sub-classing is impractical
 - sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination
 - or a class definition may be hidden or otherwise unavailable for sub-classing

Summary

- inheritance is one form of extension, but not necessarily the best way to achieve flexibility in a design
- we should allow behavior to be extended without the need to modify existing code
- the Decorator pattern provides an alternative to sub-classing for extending behavior
- a way to add additional behavior to an existing class dynamically
- decorators can result in many small objects in a design, and overuse can make your design more complex

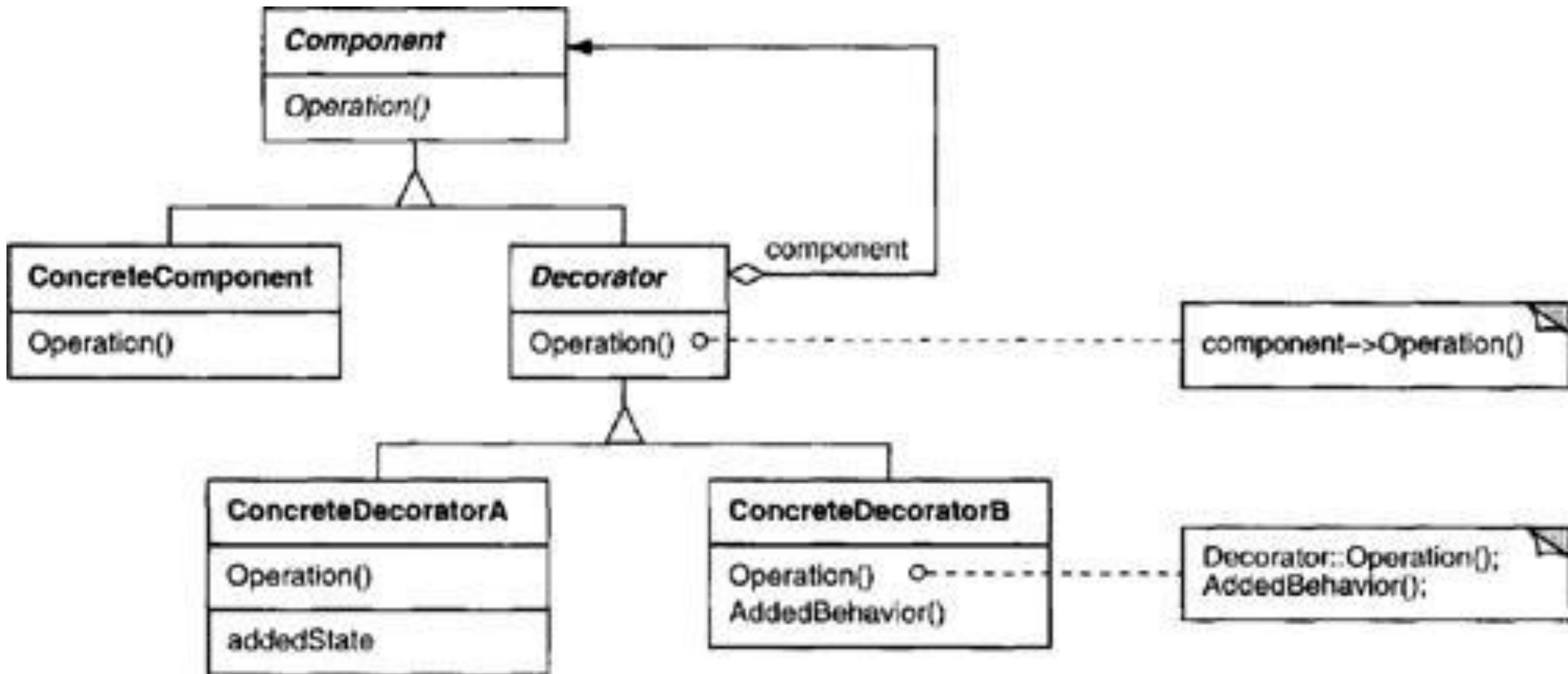
Implementing the Decorator

Jason Fedin

Overview

- the decorator pattern will create a set of decorator classes that are used to wrap concrete components
 - provides additional functionality keeping class methods signature intact
- decorator classes mirror the type of the components they decorate
 - they are the same type as the components they decorate
- decorators change the behavior of their components by adding new functionality before and/or after method calls to the component
- you can wrap a component with any number of decorators
- decorators are typically transparent to the client of the component
 - unless the client is relying on the component's concrete type

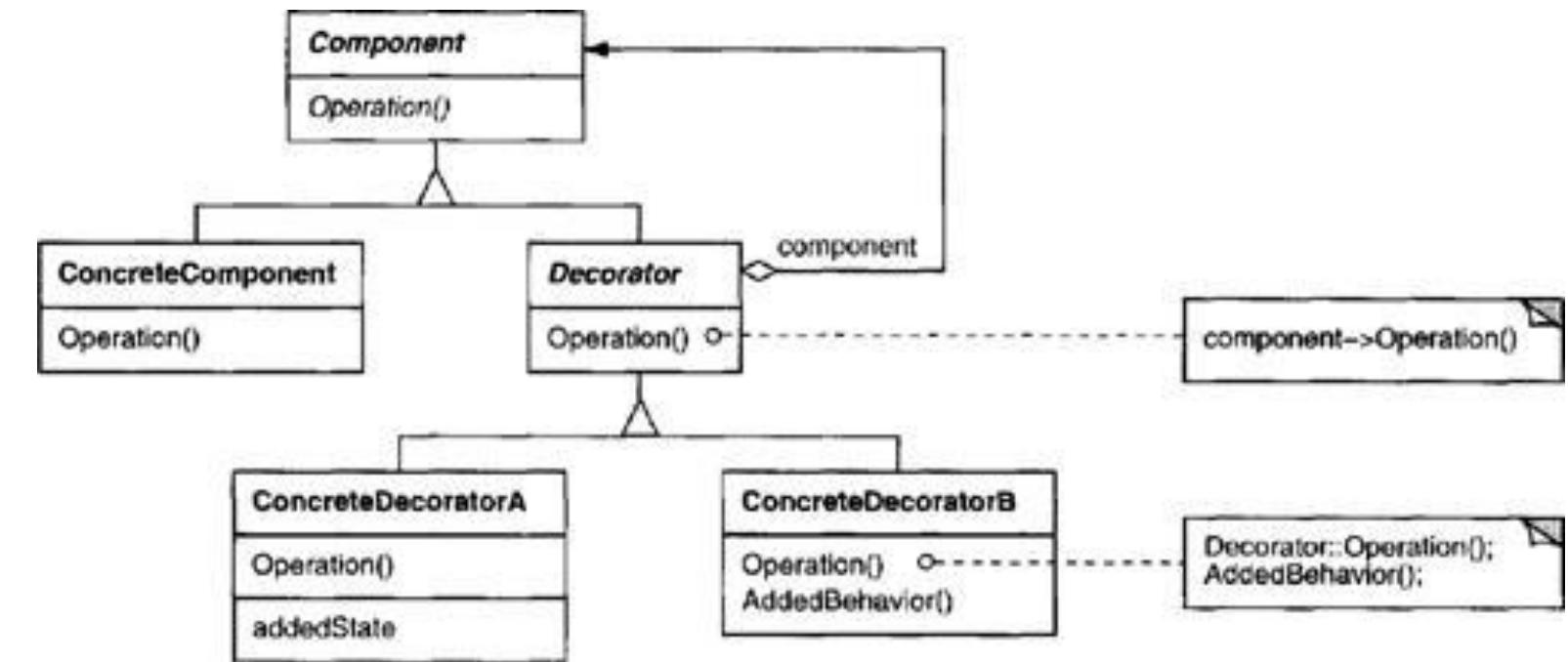
Class Diagram



Design Patterns: Elements of Reusable Object-Oriented Software By: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Participants

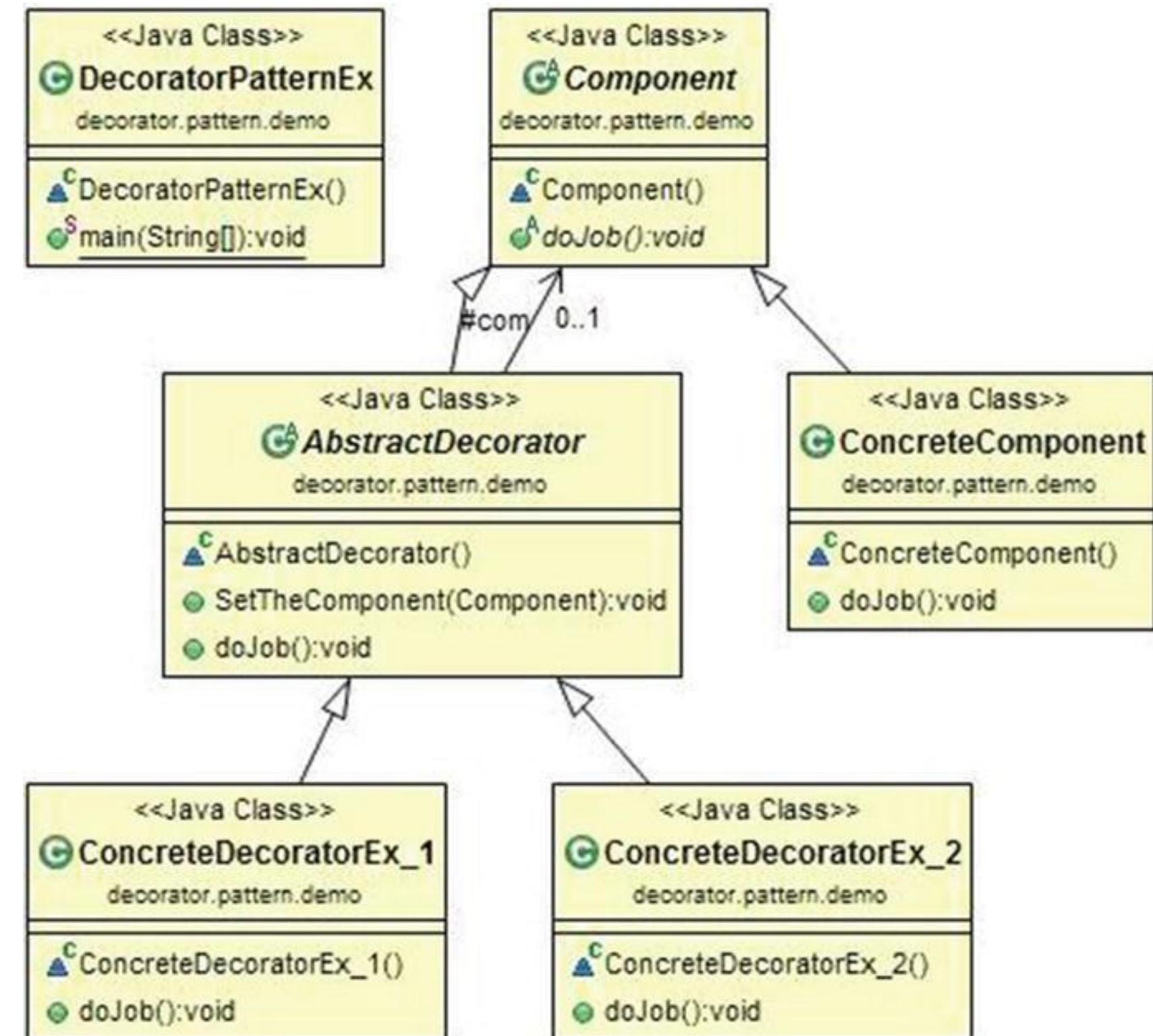
- Component
 - defines the interface for objects
 - can have responsibilities added to them dynamically
- ConcreteComponent
 - defines an object to which additional responsibilities can be attached
- Decorator
 - maintains a reference to a Component object
 - defines an interface that conforms to Component's interface
- ConcreteDecorator
 - adds responsibilities to the component
- the decorator forwards requests to its Component object
 - may optionally perform additional operations before and after forwarding the request



Example in intellij

- lets go through an example of modifying a simple method
- will create two decorators to enhance functionality without modifying the original method
 - ConcreteDecoratorEx_1 and ConcreteDecoratorEx_2

Class Diagram



Java Design Patterns: A tour of 23 gang of four design patterns in Java by: Vaskaran Sarcar

Code – create the component

- defines the interface for objects
- can have responsibilities added to them dynamically

```
abstract class Component {  
    public abstract void doJob();  
}
```

Create the concrete component

- defines an object to which additional responsibilities can be attached
 - this is the code you do not want to modify

```
class ConcreteComponent extends Component {  
  
    @Override  
    public void doJob() {  
        System.out.println(" I am from Concrete Component-I am closed for modification.");  
    }  
}
```

Create the abstract decorator

- maintains a reference to a Component object
- defines an interface that conforms to Component's interface

```
abstract class AbstractDecorator extends Component {  
    protected Component com ;  
    public void SetTheComponent(Component c) {  
        com = c;  
    }  
  
    public void doJob() {  
        if (com != null) {  
            com.doJob();  
        }  
    }  
}
```

Create the concrete decorators

- adds responsibilities to the component

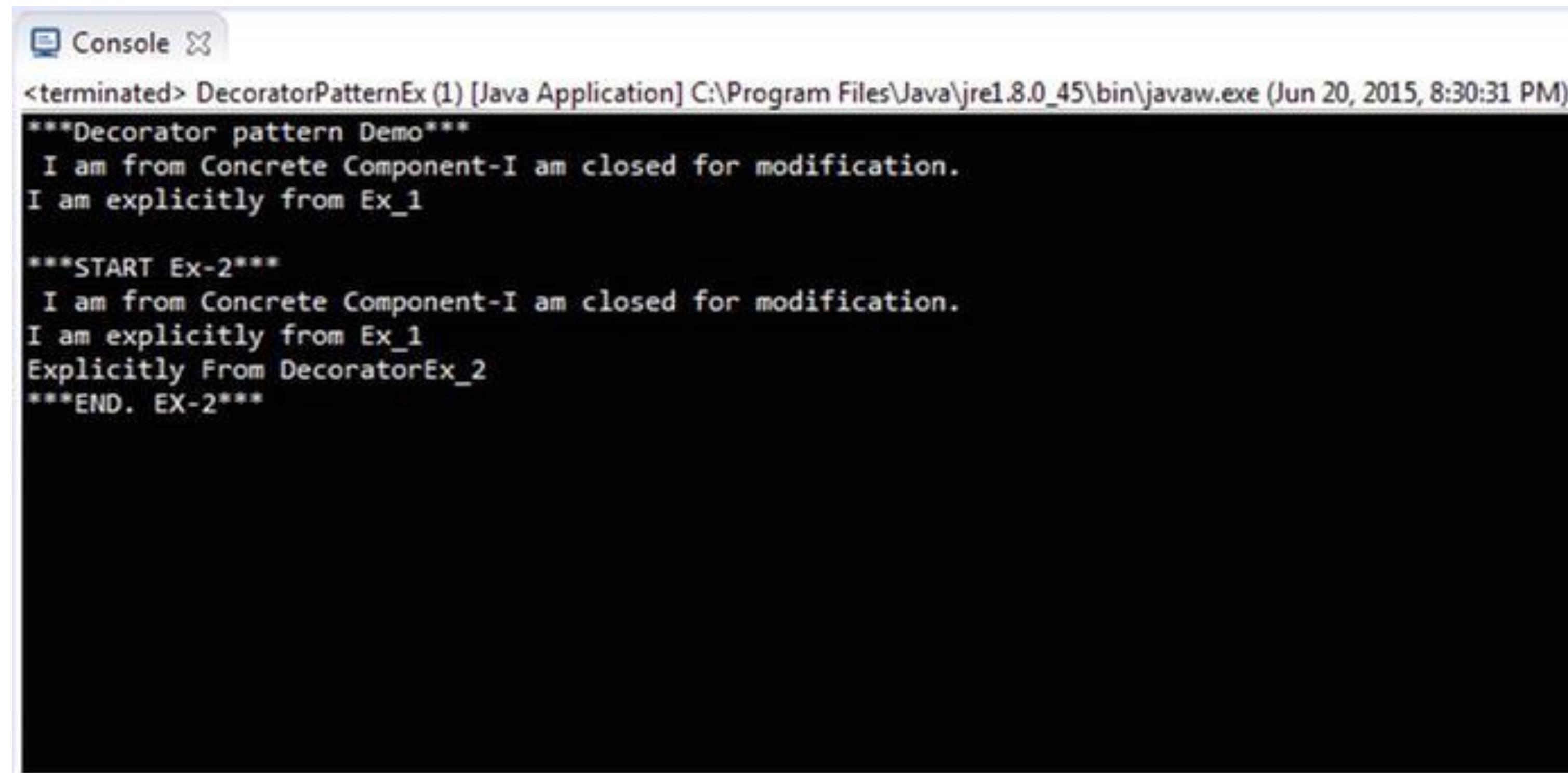
```
class ConcreteDecoratorEx_1 extends AbstractDecorator {  
    public void doJob() {  
        super.doJob();  
        //Add additional thing if necessary  
        System.out.println("I am explicitly from Ex_1");  
    }  
}
```

```
class ConcreteDecoratorEx_2 extends AbstractDecorator {  
    public void doJob() {  
        System.out.println("");  
        System.out.println("***START Ex-2***");  
        super.doJob();  
        //Add additional thing if necessary  
        System.out.println("Explicitly From DecoratorEx_2");  
        System.out.println("***END. EX-2***");  
    }  
}
```

Create the client

```
class DecoratorPatternEx {  
    public static void main(String[] args) {  
        System.out.println("***Decorator pattern Demo***");  
        ConcreteComponent cc = new ConcreteComponent();  
  
        ConcreteDecoratorEx_1 cd_1 = new ConcreteDecoratorEx_1();  
        // Decorating ConcreteComponent Object cc with ConcreteDecoratorEx_1  
        cd_1.SetTheComponent(cc);  
        cd_1.doJob();  
  
        ConcreteDecoratorEx_2 cd_2= new ConcreteDecoratorEx_2();  
        // Decorating ConcreteComponent Object cc with ConcreteDecoratorEx_1 & ConcreteDecoratorEX_2  
        cd_2.SetTheComponent(cd_1);//Adding results from cd_1 now  
        cd_2.doJob();  
    }  
}
```

Output



The screenshot shows a Java console window titled "Console". The output text is as follows:

```
<terminated> DecoratorPatternEx (1) [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Jun 20, 2015, 8:30:31 PM)
***Decorator pattern Demo***
I am from Concrete Component-I am closed for modification.
I am explicitly from Ex_1

***START Ex-2***
I am from Concrete Component-I am closed for modification.
I am explicitly from Ex_1
Explicitly From DecoratorEx_2
***END. EX-2***
```

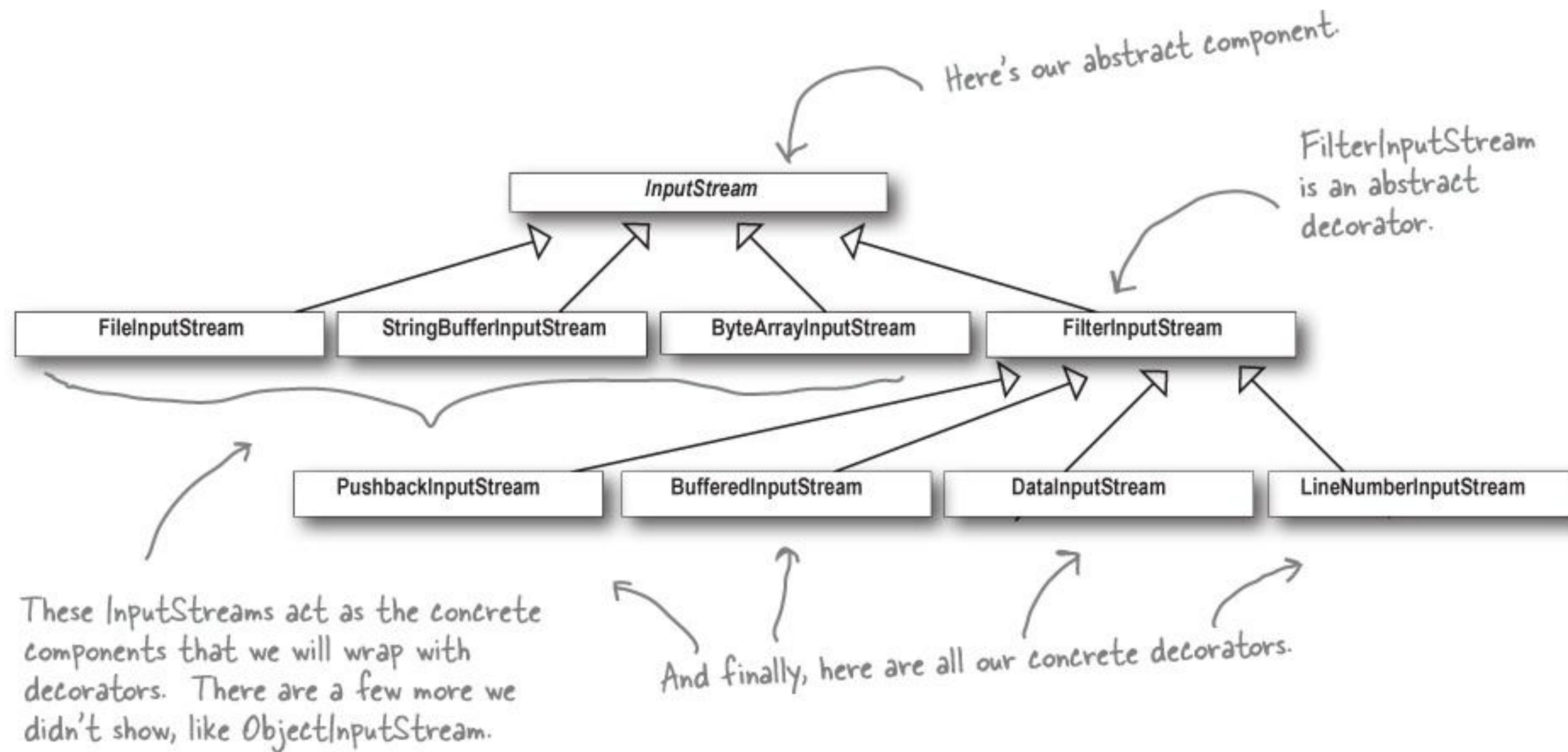
File I/O Decorator

Jason Fedin

Another Example (in intellij)

- lets write our own Java I/O Decorator
- we will write a decorator that converts all uppercase characters to lowercase in the input stream
- if we read in “I know the Decorator Pattern therefore I RULE!” then your decorator converts this to “i know the decorator pattern therefore i rule!”
- the component, concrete components, and abstract decorators have already been created for us
 - abstract component is the InputStream class
 - concrete components include FileInputStream, ByteArrayInputStream, etc.
 - the abstract decorator for all input streams is the FilterInputStream class

Java i.o. packages example



Head First Design Patterns By: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Step one, create the decorator

- we will first need to extend the FilterInputStream class and override the read() methods
 - we need to implement two read methods
 - take a byte (or an array of bytes)
 - convert each byte (that represents a character) to lowercase if it's an uppercase character

```
import java.io.*;

public class LowerCaseInputStream extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = in.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = in.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

Step 2 – use the decorator

- set up the FileInputStream and decorate it, first with a BufferedInputStream and then our brand new LowerCaseInputStream filter
- will create a new data file (test.txt)

```
import java.io.*;  
  
public class InputTest {  
    public static void main(String[] args) throws IOException {  
        int c;  
  
        try {  
            InputStream in = new LowerCaseInputStream(new BufferedInputStream(new FileInputStream("test.txt")));  
            while((c = in.read()) >= 0) {  
                System.out.print((char)c);  
            }  
  
            in.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output

```
File Edit Window Help DecoratorsRule
```

```
% java InputTest  
i know the decorator pattern therefore i rule!  
%
```

Code

Don't forget to import
java.io... (not shown).

First, extend the FilterInputStream, the
abstract decorator for all InputStreams.

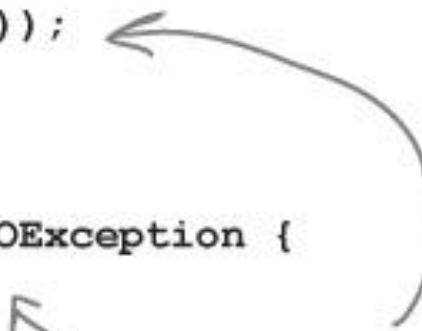


```
public class LowerCaseInputStream extends FilterInputStream {

    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = in.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = in.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```



Now we need to implement two
read methods. They take a
byte (or an array of bytes)
and convert each byte (that
represents a character) to
lowercase if it's an uppercase
character.

Code

```
public class InputTest {  
    public static void main(String[] args) throws IOException {  
        int c;  
  
        try {  
            InputStream in =  
                new LowerCaseInputStream(  
                    new BufferedInputStream(  
                        new FileInputStream("test.txt")));  
  
            while((c = in.read()) >= 0) {  
                System.out.print((char)c);  
            }  
            in.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Set up the FileInputStream and decorate it, first with a BufferedInputStream and then our brand new LowerCaseInputStream filter.

I know the Decorator Pattern therefore I RULE!

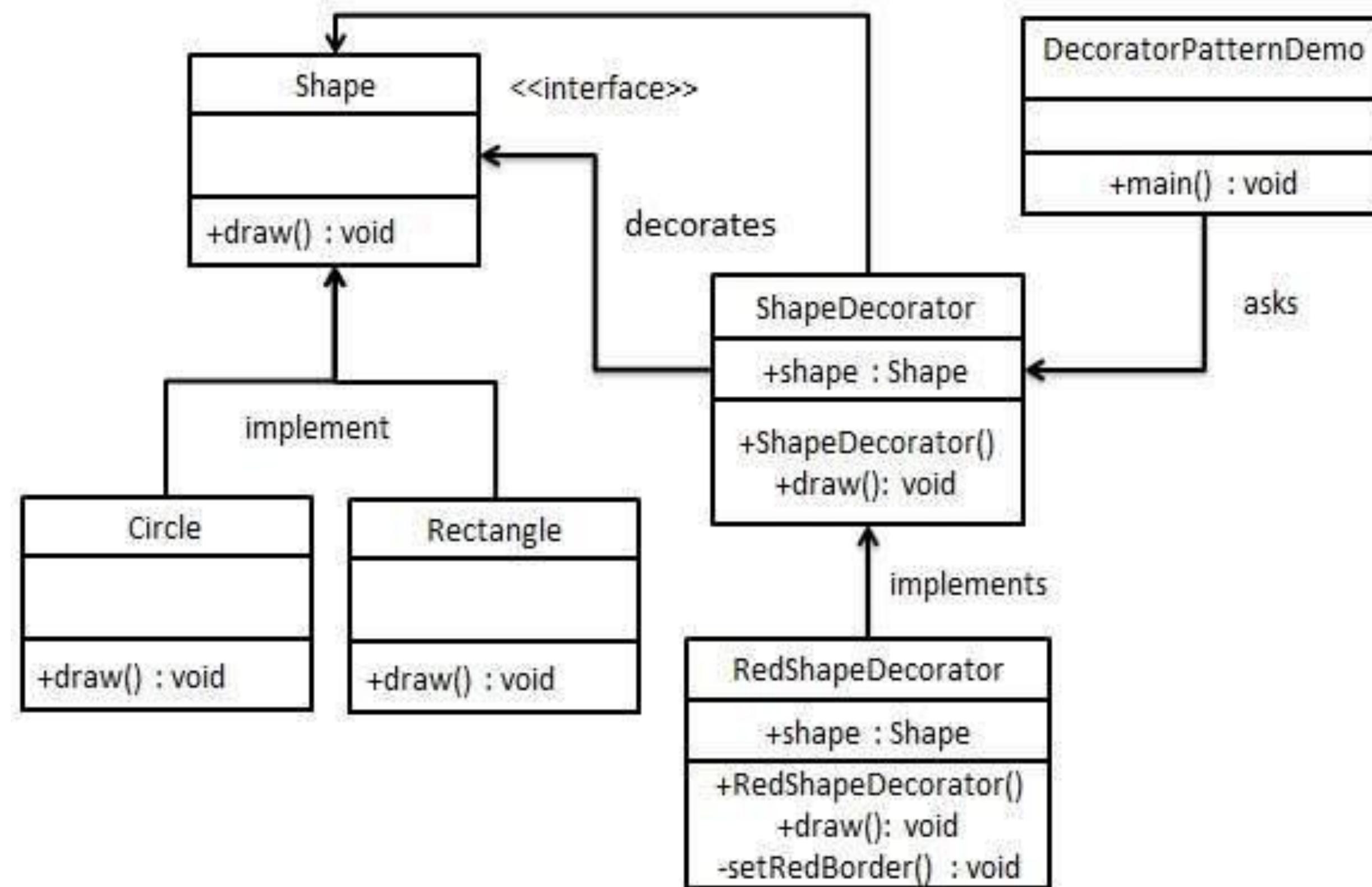
Just use the stream to read characters until the end of file and print as we go.

test.txt file
You need to make this file.

Demonstration (in intellij)

Jason Fedin

Class Diagram



Step 1 - Create an interface (Shape.java) (Component)

```
public interface Shape {  
    void draw();  
}
```

Create concrete components

- Step 2 - Create concrete classes implementing the component interface (Rectangle and Circle)

```
public class Rectangle implements Shape {
```

```
    @Override  
    public void draw() {  
        System.out.println("Shape: Rectangle");  
    }  
}
```

Create concrete components (Circle)

```
public class Circle implements Shape {
```

```
    @Override  
    public void draw() {  
        System.out.println("Shape: Circle");  
    }  
}
```

Create abstract decorator

- Step 3 - Create abstract decorator class implementing the component interface (Shape) (ShapeDecorator.java)

```
public abstract class ShapeDecorator implements Shape {  
    protected Shape decoratedShape;  
  
    public ShapeDecorator(Shape decoratedShape){  
        this.decoratedShape = decoratedShape;  
    }  
  
    public void draw(){  
        decoratedShape.draw();  
    }  
}
```

Create the concrete decorator

- Step 4 - Create concrete decorator class extending the ShapeDecorator class (RedShapeDecorator.java)

```
public class RedShapeDecorator extends ShapeDecorator {  
  
    public RedShapeDecorator(Shape decoratedShape) {  
        super(decoratedShape);  
    }  
  
    @Override  
    public void draw() {  
        decoratedShape.draw();  
        setRedBorder(decoratedShape);  
    }  
  
    private void setRedBorder(Shape decoratedShape){  
        System.out.println("Border Color: Red");  
    }  
}
```

Create the client to use the decorator

- Step 5 - Use the RedShapeDecorator to decorate Shape objects (DecoratorPatternDemo.java)

```
public class DecoratorPatternDemo {  
    public static void main(String[] args) {  
  
        Shape circle = new Circle();  
  
        Shape redCircle = new RedShapeDecorator(new Circle());  
  
        Shape redRectangle = new RedShapeDecorator(new Rectangle());  
        System.out.println("Circle with normal border");  
        circle.draw();  
  
        System.out.println("\nCircle of red border");  
        redCircle.draw();  
  
        System.out.println("\nRectangle of red border");  
        redRectangle.draw();  
    }  
}
```

Output

Circle with normal border

Shape: Circle

Circle of red border

Shape: Circle

Border Color: Red

Rectangle of red border

Shape: Rectangle

Border Color: Red

The Façade design pattern

Jason Fedin

Overview

- the façade design pattern provides a unified interface to a set of interfaces in a system
 - defines a higher-level interface that makes the subsystem easier to use
 - hide the complexities of the subsystem interfaces
 - does not add any functionality
- as the name suggests, it means the face of the building
 - people walking past the road can only see this glass face of the building
 - they do not know anything about it, the wiring, the pipes and other complexities
 - hides all the complexities of the building and displays a friendly face
- the pattern is basically saying that we need to interact with a system that is easier than the current method, or we need to use the system in a particular way
 - such as using a 3D drawing program in a 2D way

Examples

- suppose you are going to organize a birthday party and you have invited 100 people
 - you can go to any party organizer and let him/her know the minimum information
 - party type, date and time of the party, number of attendees, etc.
 - the organizer will do the rest for you
 - you do not even think about the other tasks such as
 - decorating the party room
 - whether people will take food from self-help counter or will be served by a caterer, and so on
- another good example can be the startup of a computer
 - when a computer starts up, it involves the work of the cpu, memory, hard drive, etc.
 - the facade will wrap the complexity of each task
- the JDBC interface in Java can be called a façade
 - clients create connections using the “java.sql.Connection” interface
 - the implementation of which we are not concerned about
 - implementation is left to the vendor of driver

Overview (cont'd)

- facade design pattern is more like a helper for client applications
 - does not hide subsystem interfaces from the client
- whether to use Facade or not is completely dependent on client code
- can be applied at any point of development, usually when the number of interfaces grow and system gets complex
- subsystem interfaces are not aware of Facade and they should not have any reference to the Facade interface

Why the façade?

- subsystems are groups of classes, or groups of classes and other subsystems
- structuring a system into subsystems helps reduce complexity
- the interface exposed by the classes in a subsystem or set of subsystems can become quite complex
- one way to reduce this complexity is to introduce a facade object
 - provides a single, simplified interface to the more general facilities of a subsystem
- if you need the power of the complex subsystem, it is still there for you to use

Advantages

- shields clients from subsystem components
 - reduces the number of objects that clients deal with
 - makes the subsystem easier to use
- the pattern supports loose coupling
 - we emphasize the abstraction and hide the complex details by exposing a simple interface
 - decouples a client from a subsystem of components
- facades help layer a system and the dependencies between objects
 - can eliminate complex or circular dependencies
- reduces compilation dependencies in large software systems
- simplifies porting systems to other platforms
- does not prevent applications from using subsystem classes if they need to
 - can choose between ease of use and generality

When to use the facade

- when you want to provide a simple interface to a complex subsystem
- when there are many dependencies between clients and the implementation classes of an abstraction
 - introduce a facade to decouple the subsystem from clients and other subsystems
 - promotes subsystem independence and portability
- when you want to layer your subsystems
 - use a facade to define an entry point to each subsystem level

Façade vs. adapter

- when you need to use an existing class and its interface is not the one you need, use an adapter
- when you need to simplify and unify a large interface or complex set of interfaces, use a facade
- an adapter changes an interface into one a client expects
- a facade decouples a client from a complex subsystem
- an adapter wraps an object to change its interface
- a decorator wraps an object to add new behaviors and responsibilities
- a facade “wraps” a set of objects to simplify

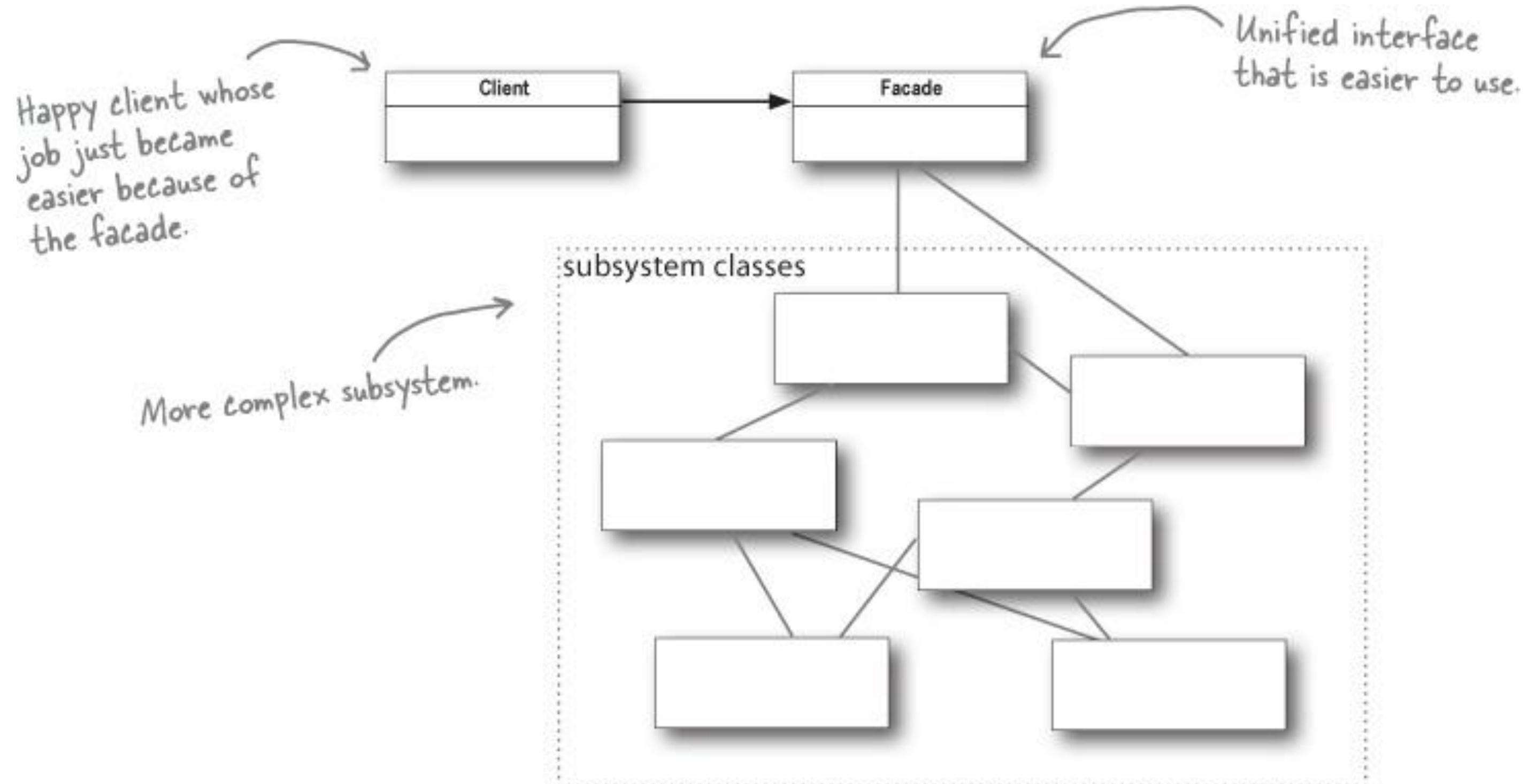
Implementing the facade pattern

Jason Fedin

Overview

- implementing the façade is fairly straightforward
 - one of the easiest patterns to implement
- there are no mind-bending abstractions to get your head around
- involves creating a single class which provides simplified methods required by the client
 - delegates calls to methods of existing system classes

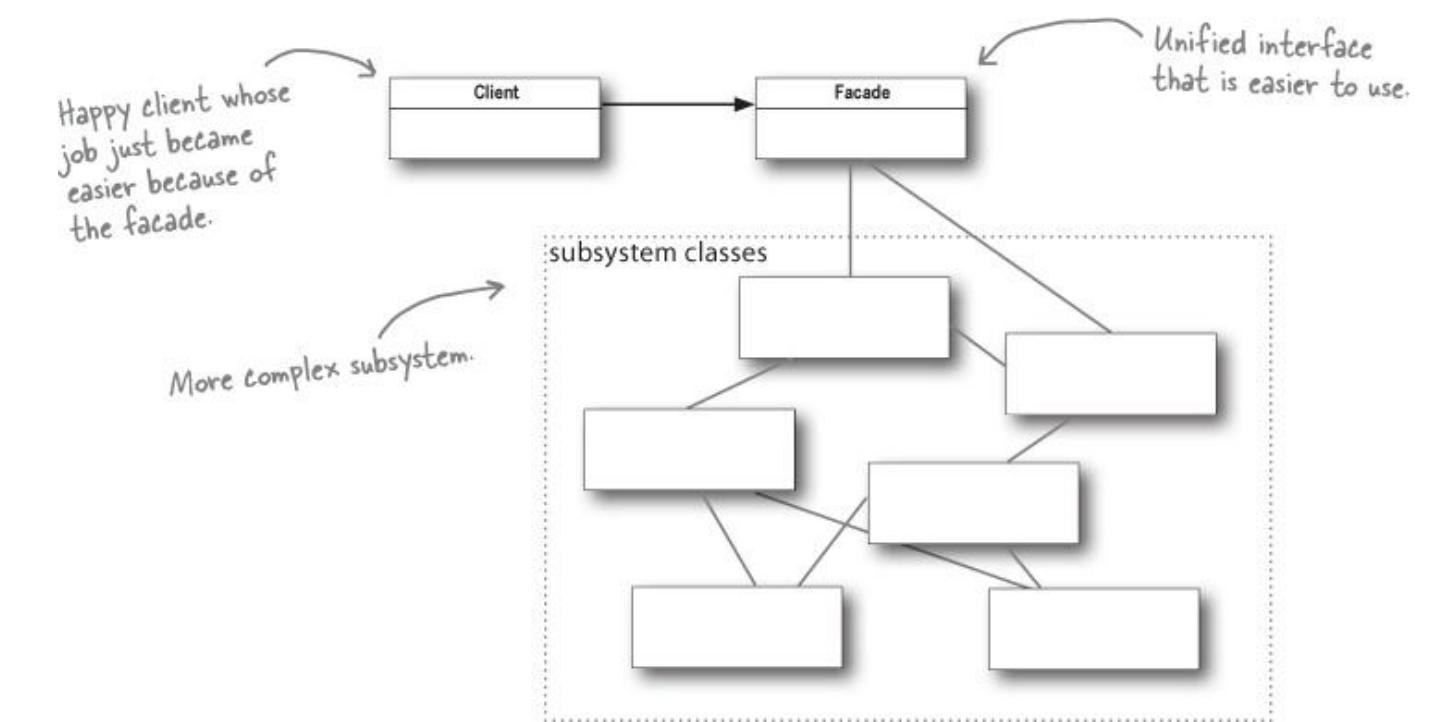
Class Diagram



Head First Design Patterns by: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Participants

- Facade
 - knows which subsystem classes are responsible for a request
 - delegates client requests to appropriate subsystem objects
- subsystem classes
 - implement subsystem functionality
 - handle work assigned by the Facade object
 - have no knowledge of the façade
 - they keep no references to it



Workflow

- clients communicate with the subsystem by sending requests to the facade
- the façade will forward the requests to the appropriate subsystem object(s)
- the subsystem objects perform the actual work
 - the facade may have to do work of its own to translate its interface to subsystem interfaces
- clients that use the facade do not have to access its subsystem objects directly

Implementation issues

- the coupling between clients and the subsystem can be reduced even further
 - make the Facade an abstract class with concrete subclasses for different implementations of a subsystem
 - clients can communicate with the subsystem through the interface of the abstract Facade class
 - keeps clients from knowing which implementation of a subsystem is used
- an alternative to sub-classing is to configure a Facade object with different subsystem objects
 - to customize the facade, simply replace one or more of its subsystem objects
- making subsystem classes private could also be useful

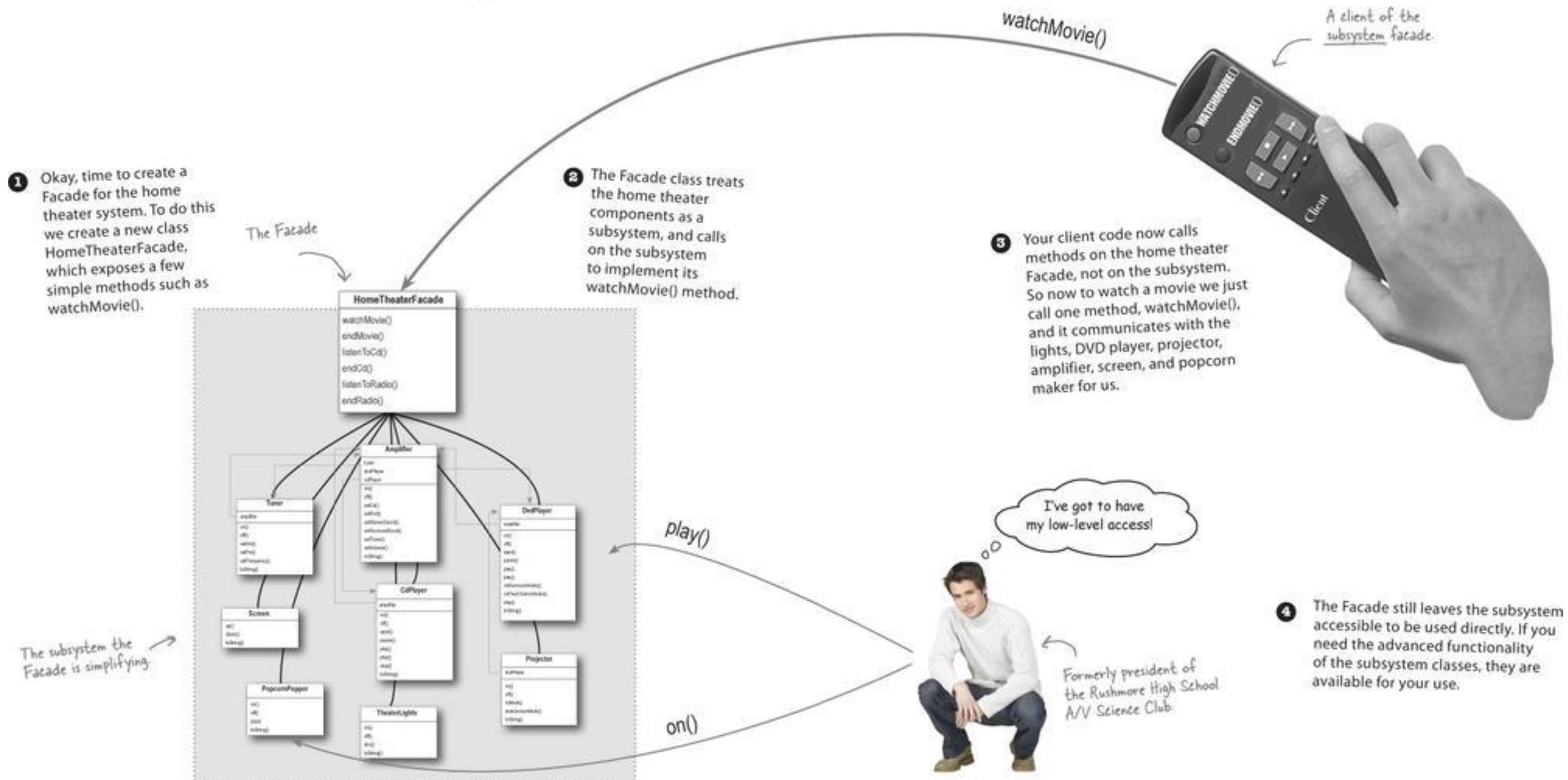
Implementation example in intelliJ

Jason Fedin

Go through source code in intellij

- lets go through an example of a home theater application
 - has a ton of subsystems that require a lot of objects to work with
 - overly complex
 - to watch a movie, we have to turn the popcorn maker on, dim the lights, pull the projector screen down, turn on the amplifier, set it to dvd, etc. and then finally play the movie
- all we want to do is watch some movies
 - lets create a façade that has a simpler interface to just watch movies
 - will call on the subsystem objects to implement the watching of a movie and ending of a movie, maybe some other things (listening to cds, etc)
- lets start by showing you the existing system and its subsystems
- go through the following java source files
 - Amplifier.java
 - CdPlayer.java
 - DvdPlayer.java
 - PopcornPopper.java
 - Projector.java
 - Screen.java
 - TheaterLights.java
 - Tuner.java

Overview



Head First Design Patterns by: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

Code – create the facade

- Go through HomeTheaterFacade.java

```
public class HomeTheaterFacade {  
    Amplifier amp;  
    Tuner tuner;  
    DvdPlayer dvd;  
    CdPlayer cd;  
    Projector projector;  
    TheaterLights lights;  
    Screen screen;  
    PopcornPopper popper;  
  
    public HomeTheaterFacade(Amplifier amp,  
        Tuner tuner,  
        DvdPlayer dvd,  
        CdPlayer cd,  
        Projector projector,  
        Screen screen,  
        TheaterLights lights,  
        PopcornPopper popper) {  
  
        this.amp = amp;  
        this.tuner = tuner;  
        this.dvd = dvd;  
        this.cd = cd;  
        this.projector = projector;  
        this.screen = screen;  
        this.lights = lights;  
        this.popper = popper;  
    }  
  
    // other methods here  
}
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

Code – create the façade (cont'd)

```
public void watchMovie(String movie) {  
    System.out.println("Get ready to watch a movie...");  
    popper.on();  
    popper.pop();  
    lights.dim(10);  
    screen.down();  
    projector.on();  
    projector.wideScreenMode();  
    amp.on();  
    amp.setDvd(dvd);  
    amp.setSurroundSound();  
    amp.setVolume(5);  
    dvd.on();  
    dvd.play(movie);  
}  
  
public void endMovie() {  
    System.out.println("Shutting movie theater down...");  
    popper.off();  
    lights.on();  
    screen.up();  
    projector.off();  
    amp.off();  
    dvd.stop();  
    dvd.eject();  
    dvd.off();  
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.

Create the client to use the facade

- go through HomeTheaterTestDrive.java code

```
public class HomeTheaterTestDrive {  
    public static void main(String[] args) {  
        // instantiate components here  
    }
```

Here we're creating the components right in the test drive. Normally the client is given a facade; it doesn't have to construct one itself.

```
    HomeTheaterFacade homeTheater =  
        new HomeTheaterFacade(amp, tuner, dvd, cd,  
        projector, screen, lights, popper);
```

First you instantiate the Facade with all the components in the subsystem.

```
    homeTheater.watchMovie("Raiders of the Lost Ark");  
    homeTheater.endMovie();
```

Use the simplified interface to first start the movie up, and then shut it down.

```
}
```

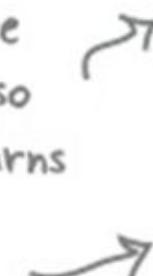
Show the Output

Here's the output.

Calling the Facade's
watchMovie() does all
this work for us...



...and here, we're done
watching the movie, so
calling endMovie() turns
everything off.



```
File Edit Window Help SnakesWhy'dItHaveToBeSnakes?  
%java HomeTheaterTestDrive  
Get ready to watch a movie...  
Popcorn Popper on  
Popcorn Popper popping popcorn!  
Theater Ceiling Lights dimming to 10%  
Theater Screen going down  
Top-O-Line Projector on  
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)  
Top-O-Line Amplifier on  
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player  
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)  
Top-O-Line Amplifier setting volume to 5  
Top-O-Line DVD Player on  
Top-O-Line DVD Player playing "Raiders of the Lost Ark"  
Shutting movie theater down...  
Popcorn Popper off  
Theater Ceiling Lights on  
Theater Screen going up  
Top-O-Line Projector off  
Top-O-Line Amplifier off  
Top-O-Line DVD Player stopped "Raiders of the Lost Ark"  
Top-O-Line DVD Player eject  
Top-O-Line DVD Player off  
%
```

Demonstration (Façade)

Jason Fedin

Code – the menu interface and classes (Menus.java)

```
public interface Menus  
{  
  
class NonVegMenu implements Menus{  
    public NonVegMenu() {  
        System.out.println("Created a NonVegMenu");  
    }  
}  
  
class VegMenu implements Menus{  
    public VegMenu() {  
        System.out.println("Created a VegMenu");  
    }  
}  
  
class Both implements Menus{  
    public Both() {  
        System.out.println("Created both menus");  
    }  
}
```

Code - (Hotel.java)

```
public interface Hotel  
{  
    public Menus getMenus();  
}
```

Code – NonVegRestaurant.java

```
public class NonVegRestaurant implements Hotel  
{  
    public Menus getMenus()  
    {  
        NonVegMenu nv = new NonVegMenu();  
        return nv;  
    }  
}
```

Code - VegRestaurant.java

```
public class VegRestaurant implements Hotel  
{  
    public Menus getMenus()  
    {  
        VegMenu v = new VegMenu();  
        return v;  
    }  
}
```

Code - VegNonBothRestaurant.java

```
public class VegNonBothRestaurant implements Hotel
{
    public Menus getMenus()
    {
        Both b = new Both();
        return b;
    }
}
```

Code – the façade (HotelKeeper.java)

```
public class HotelKeeper {  
    public VegMenu getVegMenu() {  
        VegRestaurant v = new VegRestaurant();  
        VegMenu vegMenu = (VegMenu)v.getMenus();  
        return vegMenu;  
    }  
  
    public NonVegMenu getNonVegMenu() {  
        NonVegRestaurant v = new NonVegRestaurant();  
        NonVegMenu NonvegMenu = (NonVegMenu)v.getMenus();  
        return NonvegMenu;  
    }  
  
    public Both getVegNonMenu() {  
        VegNonBothRestaurant v = new VegNonBothRestaurant();  
        Both bothMenu = (Both)v.getMenus();  
        return bothMenu;  
    }  
}
```

Code – Client.java

```
public class Client {  
    public static void main (String[] args) {  
        HotelKeeper keeper = new HotelKeeper();  
  
        VegMenu v = keeper.getVegMenu();  
        NonVegMenu nv = keeper.getNonVegMenu();  
        Both b = keeper.getVegNonMenu();  
    }  
}
```

- in this way the implementation is sent to the façade
 - the client is given just one interface and can access only that
 - hides all the complexities

The Flyweight design pattern

Jason Fedin

Overview

- the flyweight pattern uses sharing to support a large number of fine-grained objects efficiently
- the pattern is primarily used to reduce the number of objects created
 - less number of objects reduces the memory usage
 - memory usage is also minimized by sharing data as much as possible
 - crucial for low memory devices, such as mobile devices or embedded systems
 - performance is also increased
- tries to reuse already existing similar kind objects by storing them
 - one instance of a class can be used to provide many “virtual instances”
 - creates a new object when no matching object is found
- flyweight objects are shared and are immutable
 - cannot be modified once they have been constructed
- flyweight objects are used in multiple contexts simultaneously and act as an independent object in each context
 - indistinguishable from an instance of the object that is not shared

Examples

- suppose two people were each searching for an apartment so that they could stay nearby their office
 - neither of them was satisfied with the available options
 - one day, they found a place with all kind of facilities that they both desired
 - there were two constraints
 - there is only one apartment
 - the rent is high
 - so, they decided to stay together and share the rent
- the graphical representation of characters in word processors is a common example of this pattern
- a computer game where we have a large number of participants
 - looks are the same but differ from each other in their performances (or color, dresses, weapons, etc.)
- all the wrapper classes `valueOf()` method uses cached objects
 - Java String class String Pool implementation

intrinsic vs. extrinsic state

- two common terms are used when learning about the flyweight pattern
 - intrinsic state/properties - can be stored in the flyweight object and is shareable
 - extrinsic state/properties - depends on the flyweight's context and is not shareable
 - client objects define state and pass the extrinsic state to the flyweight
- lets look at an example that demonstrates the differences between these two terms
- a text editor application where we enter characters
 - an object of Character class is created
 - the attributes of the Character class are name, font, and size
 - we do not need to create an object every time a client enters a character since letter 'B' is no different from another 'B'

intrinsic vs. extrinsic state (cont'd)

- If a client again types a 'B' we simply return the object which we have already created before
 - all of these are intrinsic states (name, font, size)
 - they can be shared among the different objects as they are similar to each other
- if we add more attributes to the Character class
 - row and column
 - specify the position of a character in the document
 - these attributes will not be similar even for the same characters
 - no two characters will have the same position in a document
 - these states are termed as extrinsic states and cannot be shared amongst objects

advantages and drawbacks

- reduces the number of object instances at runtime
 - saves memory
- centralizes state for many “virtual” objects into a single location
- can control many instances of a class in the same way
- one drawback is that single, logical instances of the class will not be able to behave independently from the other instances

when to use the flyweight

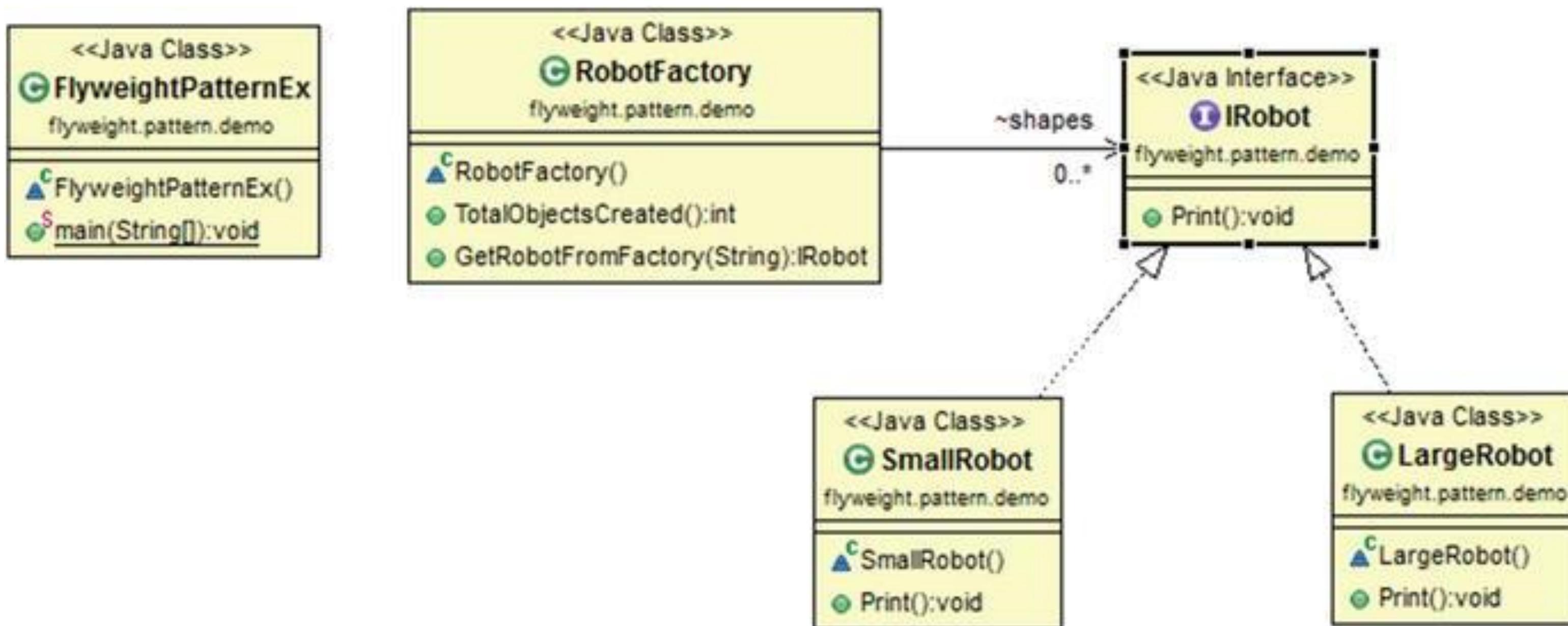
- when an application uses a large number of objects
- when storage costs are high because of the sheer quantity of objects
- when many groups of objects may be replaced by relatively few shared objects (once extrinsic state is removed)
- when the application does not depend on object identity
 - since flyweight objects may be shared, identity tests will return true for conceptually distinct objects

Example in intelliJ

Jason Fedin

Overview

- lets look at an example that uses robot objects (small and large), will use a flyweight pattern to share data (ONLY intrinsic data)



Code – Flyweight

- declares an interface through which flyweights can receive and act on extrinsic state

```
import java.util.HashMap;  
import java.util.Map;
```

```
interface IRobot  
{  
    void Print();  
}
```

Code – Concrete Flyweight (Shared)

- implements the Flyweight interface and adds storage (if any)
- must be sharable
- any state it stores must be intrinsic
 - must be independent of the ConcreteFlyweight object's context

```
class SmallRobot implements IRobot
{
    @Override
    public void Print()
    {
        System.out.println(" This is a Small Robot");
    }
}
```

Code – Concrete Flyweight (Shared)

- implements the Flyweight interface and adds storage (if any)
- must be sharable
- any state it stores must be intrinsic
 - must be independent of the ConcreteFlyweight object's context

```
class LargeRobot implements IRobot
{
    @Override
    public void Print()
    {
        System.out.println(" I am a Large Robot");
    }
}
```

Code – Flyweight factory

- creates and manages flyweight objects
- ensures that flyweights are shared properly
- when a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists

```
class RobotFactory {  
    Map<String, IRobot> shapes = new HashMap<String, IRobot>();  
  
    public int TotalObjectsCreated() {  
        return shapes.size();  
    }  
  
    public IRobot GetRobotFromFactory(String RobotCategory) throws Exception {  
        IRobot robotCategory = null;  
        if (shapes.containsKey(RobotCategory)) {  
            robotCategory = shapes.get(RobotCategory);  
        }  
    }  
}
```

Code – Flyweight factory (cont'd)

```
else {  
    switch (RobotCategory)  
    {  
        case "small":  
            System.out.println("We do not have Small Robot. So we are creating a Small Robot now.");  
            robotCategory = new SmallRobot();  
            shapes.put("small", robotCategory);  
            break;  
        case "large":  
            System.out.println("We do not have Large Robot. So we are creating a Large Robot now .");  
            robotCategory = new LargeRobot();  
            shapes.put("large", robotCategory);  
            break;  
        default:  
            throw new Exception(" Robot Factory can create only small and large shapes");  
    }  
}  
return robotCategory;  
}
```

Code - Client

- maintains a reference to flyweight(s)
- computes or stores the extrinsic state of flyweight(s)

```
class FlyweightPatternEx {  
    public static void main(String[] args) throws Exception  
{  
    RobotFactory myfactory = new RobotFactory();  
    System.out.println("\n***Flyweight Pattern Example***\n");  
  
    IRobot shape = myfactory.GetRobotFromFactory("small");  
    shape.Print();
```

Code – Client (cont'd)

```
/*Here we are trying to get the objects additional 2 times. Note that from now onward we do not need to create additional small robots as we have already created this category*/
for (int i = 0; i < 2; i++) {
    shape = myfactory.GetRobotFromFactory("small");
    shape.Print();
}

int NumOfDistinctRobots = myfactory.TotalObjectsCreated();
System.out.println("\nDistinct Robot objects created till now= " + NumOfDistinctRobots);

/*Here we are trying to get the objects 5 times.
Note that the second time onward we do not need to create additional large robots as we have already created this category in the first attempt(at i=0)*/
for (int i = 0; i < 5; i++) {
    shape = myfactory.GetRobotFromFactory("large");
    shape.Print();
}

NumOfDistinctRobots = myfactory.TotalObjectsCreated();
System.out.println("\n Finally no of Distinct Robot objects created: " + NumOfDistinctRobots);
}
```

Output

```
@ Javadoc Declaration Console ✎
<terminated> FlyweightPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Jun 8, 2015, 9:06:30 PM)

***Flyweight Pattern Example***

We do not have Small Robot.So we are creating a Small Robot now.
This is a Small Robot
This is a Small Robot
This is a Small Robot

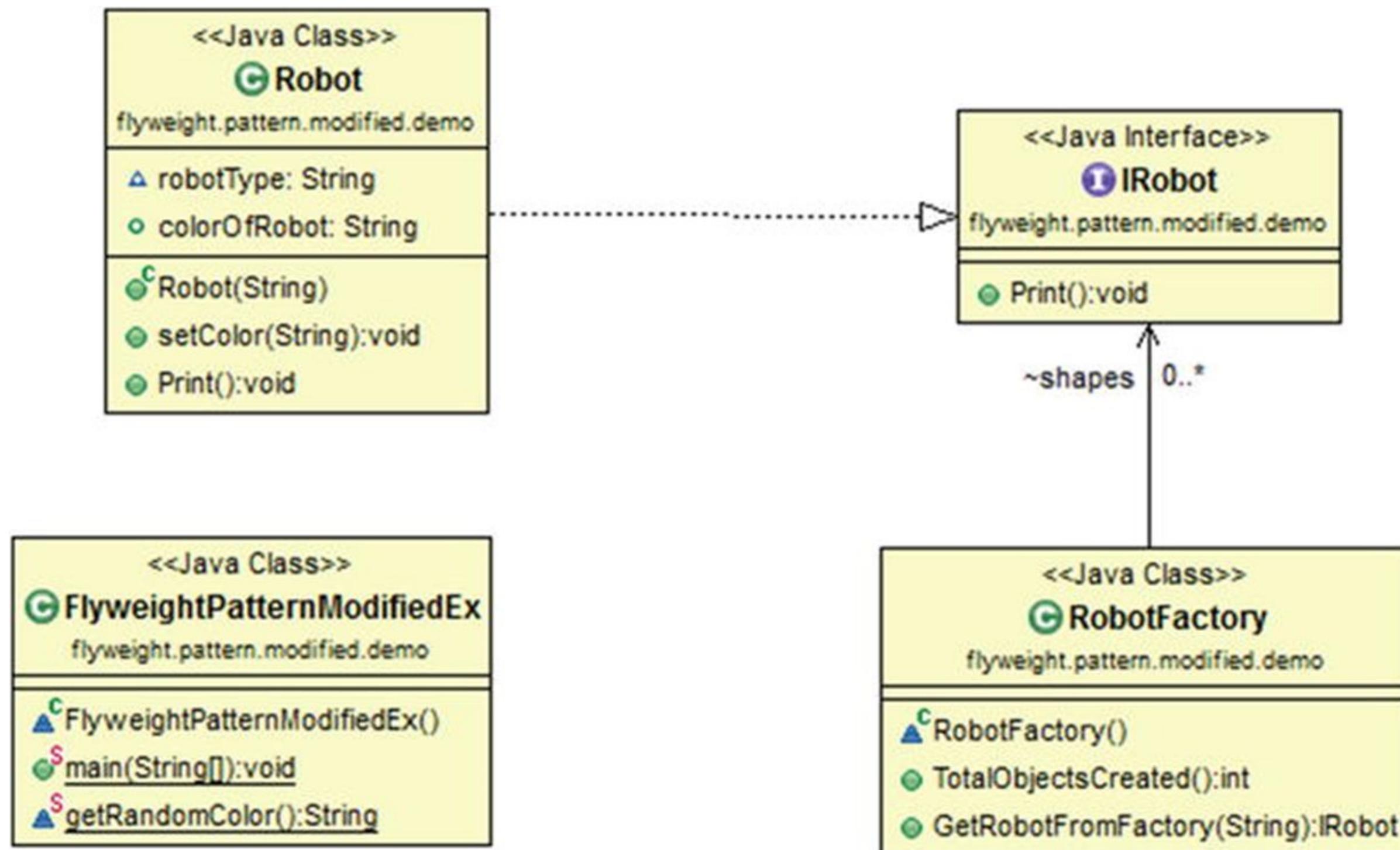
Distinct Robot objects created till now= 1
We do not have Large Robot.So we are creating a Large Robot now .
I am a Large Robot

Finally no of Distinct Robot objects created: 2
```

Improvement to the program

- it seems that this pattern is behaving similar to a singleton pattern
 - we cannot create two distinct types of small (or large) robots
 - however, there may be situations where we want different types of small (or large) robots
 - the basic structure should be the same but it will differ only with some special characteristics
- so, lets look at another example
 - will clear up any doubt and help you understand the distinction between the flyweight and the singleton patterns
- to make the program simple, we are dealing with robots which can be either king type or queen type
 - each of these types can be either green or red
- before making any robot, we will consult with our factory
 - if we already have king or queen types of robots, we will not create them again
 - we will collect the basic structure from our factory and after that we will color them
 - color is extrinsic data here, but the category of robot (king or queen) is intrinsic

Class Diagram (with extrinsic data, color)



Code – Flyweight

- declares an interface through which flyweights can receive and act on extrinsic state

```
import java.util.HashMap;
import java.util.Map;
import java.util.Random;

interface IRobot
{
    void Print();

    // extrinsic data is passed as arguments
    void setColor(String colorOfRobot);
}
```

Code – Concrete Flyweight (Shared)

- implements the Flyweight interface and adds storage (if any)
- must be sharable
- any state it stores must be intrinsic
 - must be independent of the ConcreteFlyweight object's context
- extrinsic data are passed as arguments (setColor)

```
class Robot implements IRobot {  
    String robotType;  
    public String colorOfRobot;  
    public Robot(String robotType){  
        this.robotType=robotType;  
    }  
    public void setColor(String colorOfRobot){  
        this.colorOfRobot=colorOfRobot;  
    }  
    @Override  
    public void Print(){  
        System.out.println(" This is a " +robotType+ " type robot with "+colorOfRobot+ "color");  
    }  
}
```

Code – Flyweight factory

- creates and manages flyweight objects
- ensures that flyweights are shared properly
- when a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists

```
class RobotFactory {  
    Map<String, IRobot> shapes = new HashMap<String, IRobot>();  
  
    public int TotalObjectsCreated() {  
        return shapes.size();  
    }  
  
    public IRobot GetRobotFromFactory(String robotType) throws Exception {  
        IRobot robotCategory= null;  
        if (shapes.containsKey(robotType)) {  
            robotCategory = shapes.get(robotType);  
        }  
    }  
}
```

Code – Flyweight factory (cont'd)

```
else {  
    switch (robotType) {  
        case "King":  
            System.out.println("We do not have King Robot. So we are creating a King Robot now.");  
            robotCategory = new Robot("King");  
            shapes.put("King",robotCategory);  
            break;  
        case "Queen":  
            System.out.println("We do not have Queen Robot. So we are creating a Queen Robot now.");  
            robotCategory = new Robot("Queen");  
            shapes.put("Queen",robotCategory);  
            break;  
        default:  
            throw new Exception(" Robot Factory can create only king and queen type robots");  
    }  
}  
return robotCategory;  
}
```

Code - Client

- maintains a reference to flyweight(s)
- computes or stores the extrinsic state of flyweight(s)

```
class FlyweightPatternModifiedEx {  
    public static void main(String[] args) throws Exception {  
        RobotFactory myfactory = new RobotFactory();  
        System.out.println("\n***Flyweight Pattern Example Modified***\n");  
        Robot shape;  
  
        /*Here we are trying to get 3 king type robots*/  
        for (int i = 0; i < 3; i++) {  
            shape =(Robot) myfactory.GetRobotFromFactory("King");  
            shape.setColor(getRandomColor());  
            shape.Print();  
        }  
    }  
}
```

Code – Client (cont'd)

```
/*Here we are trying to get 3 queen type robots*/
for (int i = 0; i < 3; i++) {
    shape =(Robot) myfactory.GetRobotFromFactory("Queen");
    shape.setColor(getRandomColor());
    shape.Print();
}
int NumOfDistinctRobots = myfactory.TotalObjectsCreated();
//System.out.println("\nDistinct Robot objects created till now = "+ NumOfDistinctRobots);
System.out.println("\n Finally no of Distinct Robot objects created: "+ NumOfDistinctRobots);
}
```

Code – Client (cont'd)

```
static String getRandomColor() {  
    Random r=new Random();  
    /*You can supply any number of your choice in nextInt argument.  
     * we are simply checking the random number generated is an even number  
     * or an odd number. And based on that we are choosing the color.  
     * For simplicity, we'll use only two colors—red and green  
     */  
    int random=r.nextInt(20);  
    if(random%2==0) {  
        return "red";  
    }  
    else {  
        return "green";  
    }  
}
```

Output

```
Console X
<terminated> FlyweightPatternModifiedEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 21, 2015, 4:31:43 PM)

***Flyweight Pattern Example Modified***

We do not have King Robot. So we are creating a King Robot now.
This is a King type robot with greencolor
This is a King type robot with greencolor
This is a King type robot with redcolor
We do not have Queen Robot. So we are creating a Queen Robot now.
This is a Queen type robot with greencolor
This is a Queen type robot with greencolor
This is a Queen type robot with redcolor

Finally no of Distinct Robot objects created: 2
```

Demonstration (Flyweight)

Jason Fedin

Step 1 - Create the Flyweight interface

```
import java.util.Random;
import java.util.HashMap;

// A common interface for all players
interface Player {
    // extrinsic data
    public void assignWeapon(String weapon);
    public void mission();
}
```

Create the Concrete Flyweights (Terrorist, shared)

```
// Terrorist must have weapon and mission
class Terrorist implements Player {
    // Intrinsic Attribute
    private final String TASK;

    // Extrinsic Attribute
    private String weapon;

    public Terrorist() {
        TASK = "PLANT A BOMB";
    }

    public void assignWeapon(String weapon) {
        // Assign a weapon
        this.weapon = weapon;
    }

    public void mission() {
        //Work on the Mission
        System.out.println("Terrorist with weapon " + weapon + " | " + " Task is " + TASK);
    }
}
```

Create the Concrete Flyweights (CT, shared)

```
// CounterTerrorist must have weapon and mission
class CounterTerrorist implements Player {
    // Intrinsic Attribute
    private final String TASK;
    // Extrinsic Attribute
    private String weapon;

    public CounterTerrorist() {
        TASK = "DIFFUSE BOMB";
    }
    public void assignWeapon(String weapon) {
        this.weapon = weapon;
    }
    public void mission() {
        System.out.println("Counter Terrorist with weapon " + weapon + " | " + " Task is " + TASK);
    }
}
```

Create the factory

```
class PlayerFactory
{
    /* HashMap stores the reference to the object of Terrorist(TS) or CounterTerrorist(CT). */
    private static HashMap <String, Player> hm = new HashMap<String, Player>();

    // Method to get a player
    public static Player getPlayer(String type) {
        Player p = null;

        /* If an object for TS or CT has already been created simply return its reference */
        if (hm.containsKey(type))
            p = hm.get(type);
    }
}
```

Create the factory

```
else {  
    /* create an object of TS/CT */  
    switch(type) {  
        case "Terrorist":  
            System.out.println("Terrorist Created");  
            p = new Terrorist();  
            break;  
        case "CounterTerrorist":  
            System.out.println("Counter Terrorist Created");  
            p = new CounterTerrorist();  
            break;  
        default :  
            System.out.println("Unreachable code!");  
    }  
  
    // Once created insert it into the HashMap  
    hm.put(type, p);  
}  
return p;  
}  
}
```

Create the Client

```
public class CounterStrike {  
    // All player types and weapons (used by getRandPlayerType()  
    // and getRandWeapon()  
  
    private static String[] playerType = {"Terrorist", "CounterTerrorist"};  
  
    private static String[] weapons = {"AK-47", "Maverick", "Gut Knife", "Desert Eagle"};
```

Client (cont'd)

```
public static void main(String args[]) {  
    /* Assume that we have a total of 10 players in the game. */  
    for (int i = 0; i < 10; i++) {  
  
        /* getPlayer() is called simply using the class name since the method is a static one */  
        Player p = PlayerFactory.getPlayer(getRandPlayerType());  
  
        /* Assign a weapon chosen randomly uniformly from the weapon array */  
        p.assignWeapon(getRandWeapon());  
  
        // Send this player on a mission  
        p.mission();  
    }  
}
```

Client (cont'd)

```
// Utility methods to get a random player type and weapon
public static String getRandPlayerType() {
    Random r = new Random();

    // Will return an integer between [0,2)
    int randInt = r.nextInt(playerType.length);

    // return the player stored at index 'randInt'
    return playerType[randInt];
}

public static String getRandWeapon() {
    Random r = new Random();

    // Will return an integer between [0,5)
    int randInt = r.nextInt(weapons.length);

    // Return the weapon stored at index 'randInt'
    return weapons[randInt];
}
```

Output

Counter Terrorist Created

Counter Terrorist with weapon Gut Knife | Task is DIFFUSE BOMB

Counter Terrorist with weapon Desert Eagle | Task is DIFFUSE BOMB

Terrorist Created

Terrorist with weapon AK-47 | Task is PLANT A BOMB

Terrorist with weapon Gut Knife | Task is PLANT A BOMB

Terrorist with weapon Gut Knife | Task is PLANT A BOMB

Terrorist with weapon Desert Eagle | Task is PLANT A BOMB

Terrorist with weapon AK-47 | Task is PLANT A BOMB

Counter Terrorist with weapon Desert Eagle | Task is DIFFUSE BOMB

Counter Terrorist with weapon Gut Knife | Task is DIFFUSE BOMB

Counter Terrorist with weapon Desert Eagle | Task is DIFFUSE BOMB

The proxy design pattern

Jason Fedin

Overview

- the proxy design pattern provides a surrogate or placeholder for another object to control access to it
 - used when we want to provide controlled access of a functionality
- the formal definition of a proxy is a person authorized to act for another person
 - an agent or substitute
 - the authority to act for another
- there are situations in which a client does not or can not reference an object directly, but wants to still interact with the object
 - introduces a level of indirection when accessing an object
- a proxy object can act as the intermediary between the client and the target object
- another common use case is to provide a wrapper implementation for better performance

examples

- a check or a credit card is a proxy for what is in a bank account
 - can be used in place of cash
 - provides a means of accessing that cash when required
 - exactly what a proxy does, it controls and manage access to the object they are protecting
- in a classroom, when one student is absent, during roll call, his best friend may try to mimic the student's voice to try to keep his friend from being marked as absent
- lets say we have a class that can run some command on a system
 - if we are using it, it works fine
 - if we want to give this program to a client application
 - can have severe issues because client program can issue commands to delete some system files or change some settings that you do not want to
 - a proxy class can be created to provide controlled access of the program
- a proxy is used in the java API remote method invocation package (java.rmi.*)

examples (cont'd)

- one reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it
- for example, a document editor that can embed graphical objects in a document
 - large raster images can be expensive to create
 - however, opening a document should be fast
 - we should avoid creating all the expensive objects at once when the document is opened
- these constraints would suggest creating each expensive object on demand
 - occurs when an image becomes visible
- the solution is to use another object
 - an image proxy that acts as a stand-in for the real image
 - the proxy acts just like the image and takes care of instantiating it when it is required

types of proxies

- remote proxy
 - manages interaction between a client and a remote object
 - provides a reference to an object located in a different address space on the same or different machine
- virtual proxy
 - controls access to an object that is expensive to instantiate
 - allows for the creation of a memory intensive object on demand
 - object will not be created until it is really needed
- Copy-On-Write proxy
 - defers copying (cloning) a target object until required by client actions
 - a form of a virtual proxy
- protection (Access) proxy
 - provides different clients with different levels of access to a target object

types of proxies (cont'd)

- cache proxy
 - provides temporary storage of the results of expensive target operations so that multiple clients can share the results
- firewall proxy
 - protects targets from bad clients (or vice versa)
- synchronization proxy
 - provides multiple accesses to a target object
- smart reference proxy
 - provides additional actions whenever a target object is referenced such as counting the number of references to the object

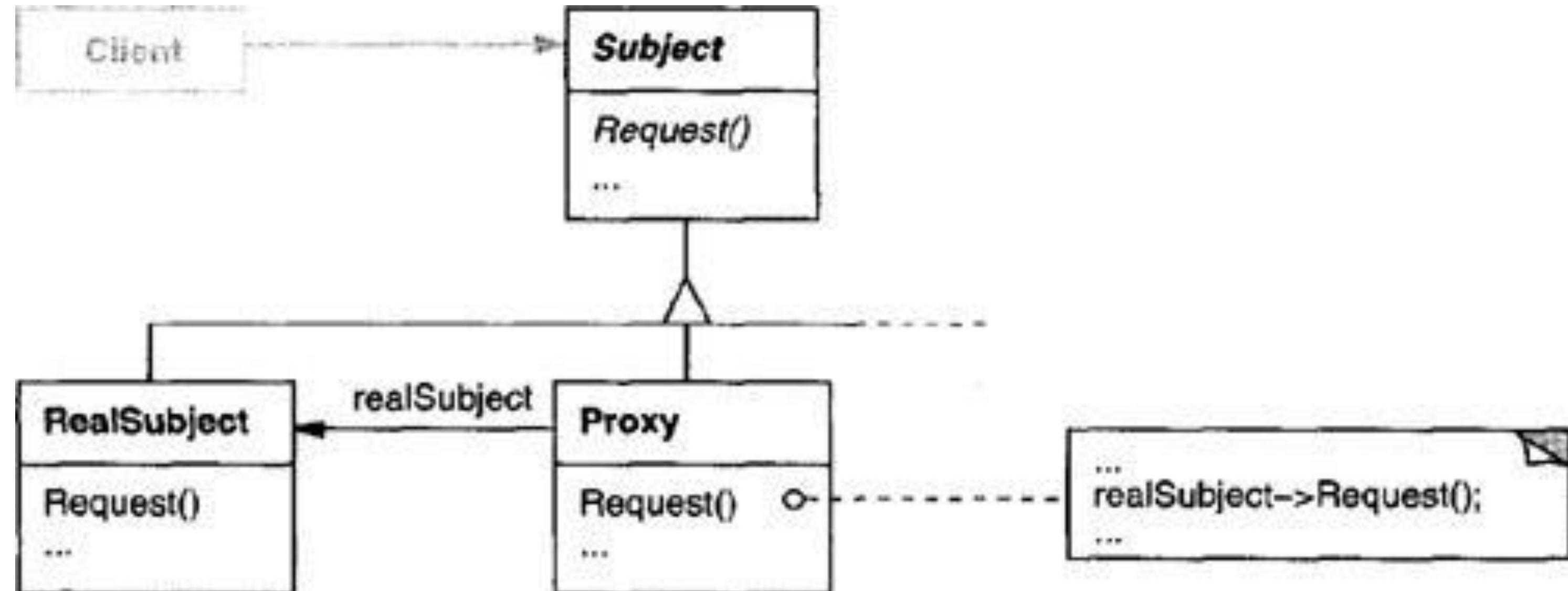
Summary

- the proxy pattern provides a representative for another object in order to control the client's access to it
- security is a big advantage
 - remote proxies ensures a more secure application by installing the local code proxy (stub) in the client machine and then accessing the server with help of the remote code
- avoids duplication of objects which might be huge in size and memory intensive
 - increases the performance of the application
- proxy is structurally similar to decorator, but the two differ in their purpose
 - decorator pattern adds behavior to an object
 - proxy controls access
- proxies will increase the number of classes and objects in your designs

Implementing the Proxy

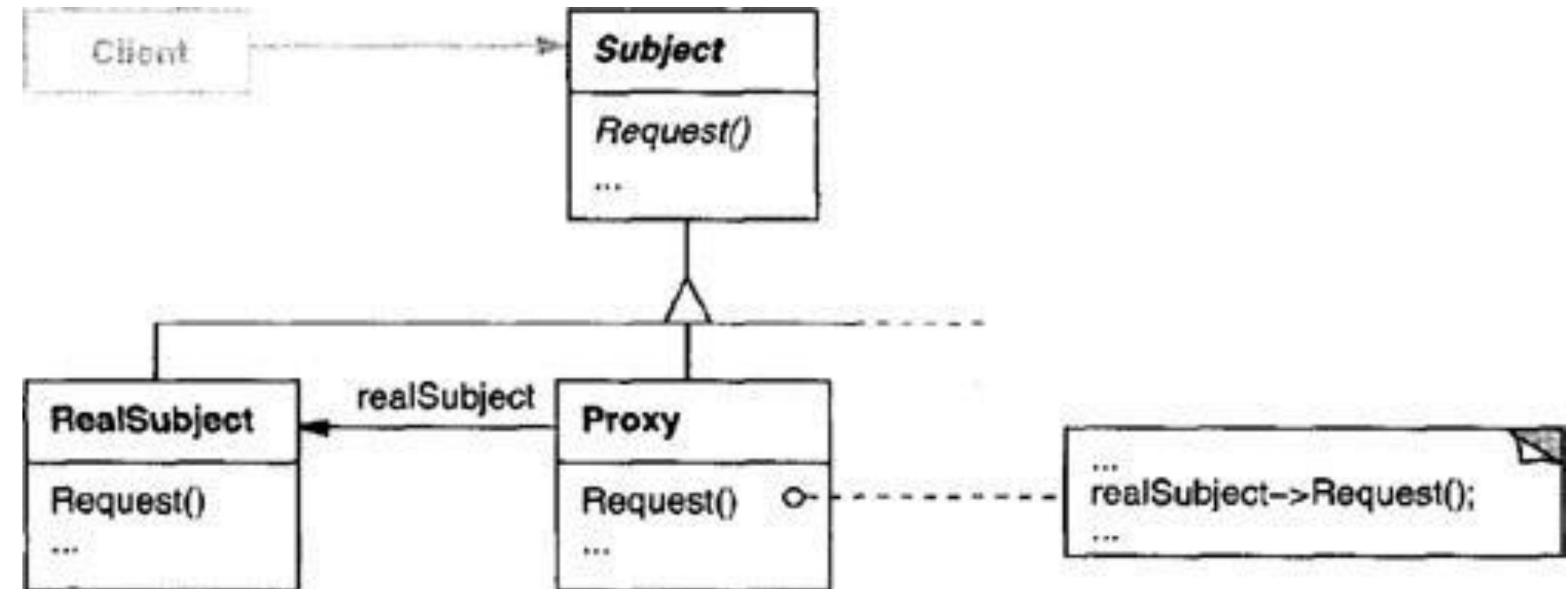
Jason Fedin

Class Diagram



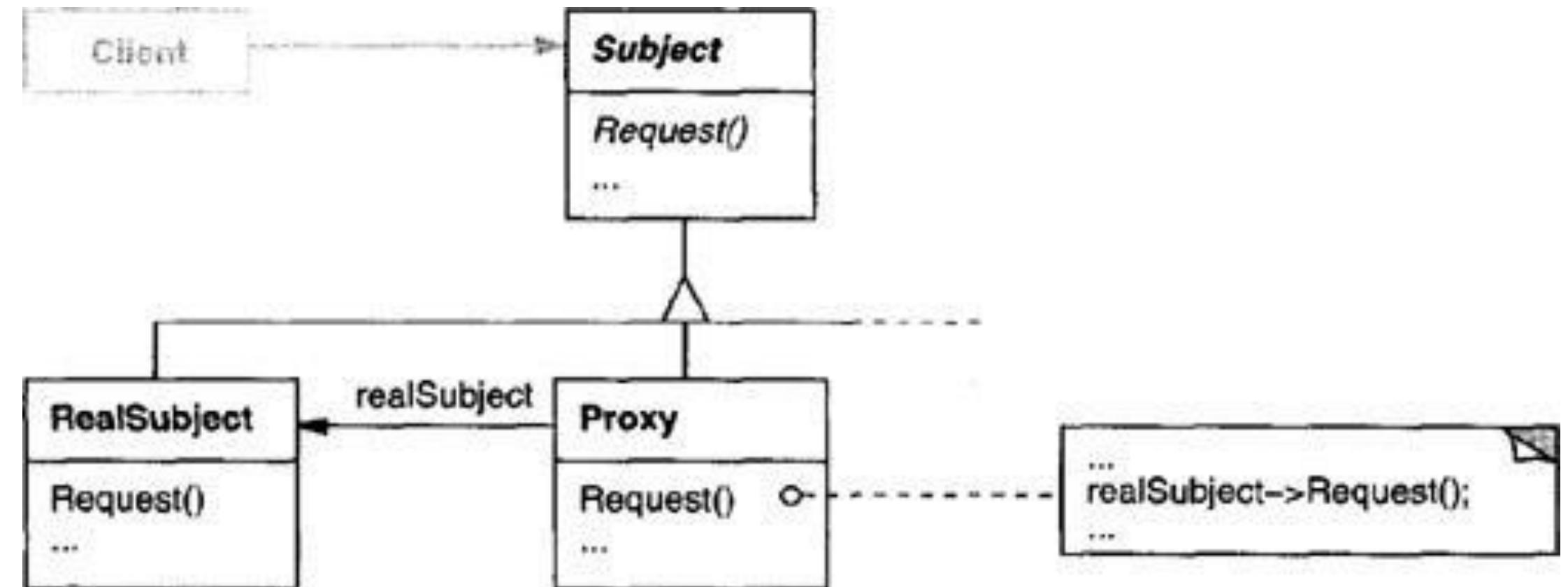
Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Participants



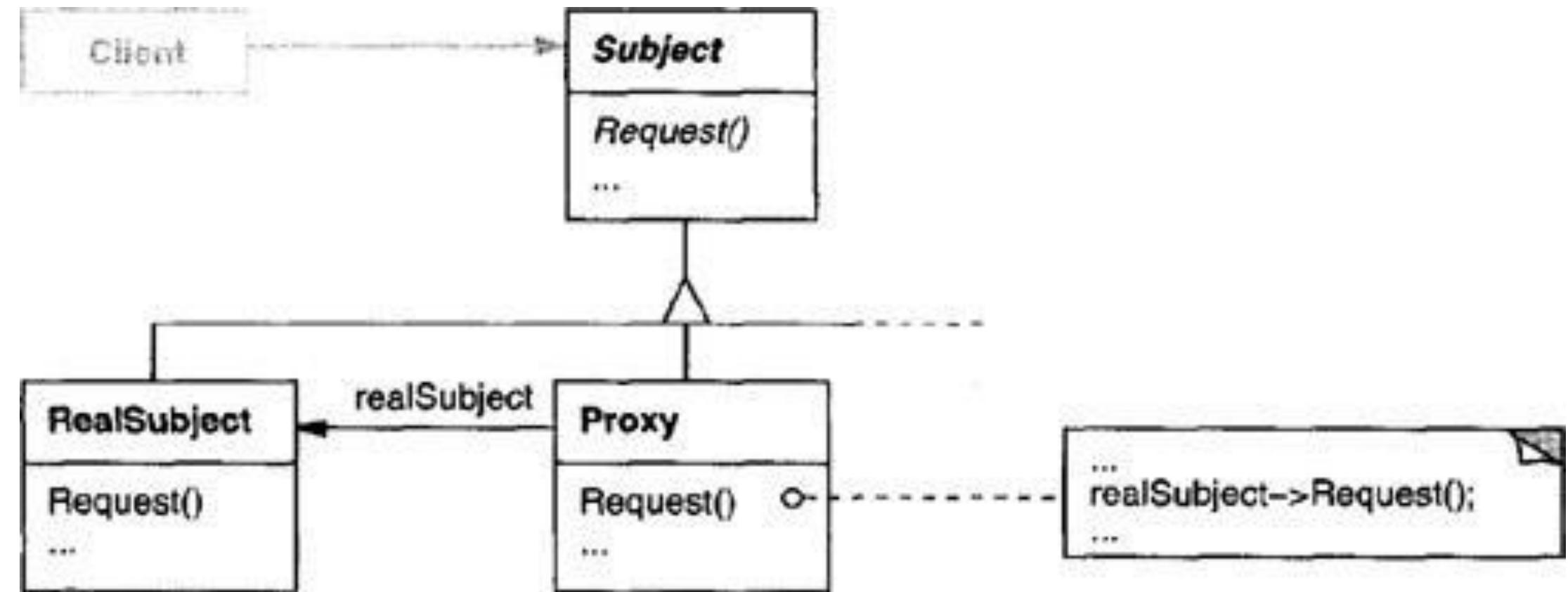
- **Subject**
 - defines the common interface for **RealSubject** and **Proxy**
 - a **Proxy** can be used anywhere a **RealSubject** is expected
- **RealSubject**
 - defines the real object that the proxy represents and controls access to
 - does the real work

Participants



- Proxy
 - clients interact with the RealSubject through the Proxy
 - maintains a reference that lets the proxy access the real subject
 - controls access to the real subject and may be responsible for creating and deleting it
 - may be needed if the Subject is running on a remote machine
 - may be needed if the Subject is expensive to create in some way or if access to the subject needs to be protected in some way
 - forwards requests to RealSubject when appropriate (delegation)
 - depending on type of proxy

Participants (proxy)

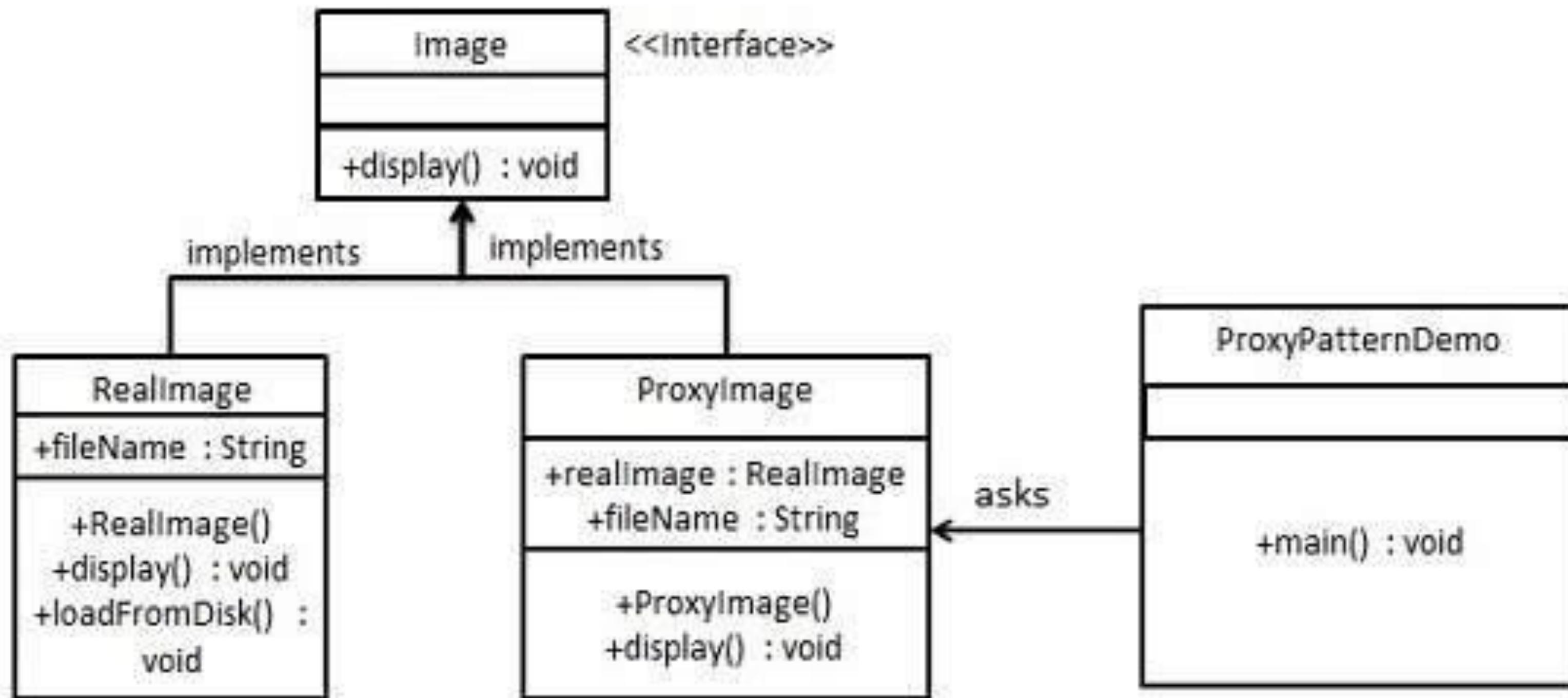


- Proxy (cont'd)
 - provides an interface identical to Subject's so that a proxy can be substituted for the real subject
 - may refer to a Subject if the RealSubject and Subject interfaces are the same
 - other responsibilities depend on the kind of proxy:
 - remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space
 - virtual proxies may cache additional information about the real subject so that they can postpone accessing it
 - protection proxies check that the caller has the access permissions required to perform a request
 - has the authority to act on behalf of the client to interact with the target object

Example in intellij

- lets create an image proxy
 - will help reduce the memory footprint of loading large images
- we are going to create an Image interface and concrete classes implementing the Image interface
- a proxylimage will be our proxy class

Class Diagram



Create the Subject (Image.java)

- defines the common interface for RealSubject and Proxy
 - a Proxy can be used anywhere a RealSubject is expected

```
public interface Image {  
    void display();  
}
```

Create the RealSubject (ReallImage.java)

- defines the real object that the proxy represents and controls access to
 - does the real work

```
public class ReallImage implements Image {  
  
    private String fileName;  
  
    public ReallImage(String fileName){  
        this.fileName = fileName;  
        loadFromDisk(fileName);  
    }  
  
    @Override  
    public void display() {  
        System.out.println("Displaying " + fileName);  
    }  
  
    private void loadFromDisk(String fileName){  
        System.out.println("Loading " + fileName);  
    }  
}
```

Create the proxy (ProxyImage.java)

- refer to slides 4 and 5 on what the proxy does

```
public class ProxyImage implements Image{  
  
    private ReallImage reallImage;  
    private String fileName;  
  
    public ProxyImage(String fileName){  
        this.fileName = fileName;  
    }  
  
    @Override  
    public void display() {  
        if(reallImage == null){  
            reallImage = new ReallImage(fileName);  
        }  
        reallImage.display();  
    }  
}
```

Create the Client (ProxyPatternDemo.java)

- use the ProxylImage to get object of ReallImage class when required

```
public class ProxyPatternDemo {  
  
    public static void main(String[] args) {  
        Image image = new ProxylImage("test_10mb.jpg");  
  
        //image will be loaded from disk  
        image.display();  
        System.out.println("");  
  
        //image will not be loaded from disk  
        image.display();  
    }  
}
```

Output

Loading test_10mb.jpg

Displaying test_10mb.jpg

Displaying test_10mb.jpg

(Demonstration) Proxy

Jason Fedin

Code – Create the Subject

```
public interface Internet  
{  
    public void connectTo(String serverhost) throws Exception;  
}
```

Code – Create the real subject

```
public class ReallInternet implements Internet
{
    @Override
    public void connectTo(String serverhost)
    {
        System.out.println("Connecting to "+ serverhost);
    }
}
```

Code – Create the proxy

```
import java.util.ArrayList;  
import java.util.List;  
  
public class ProxyInternet implements Internet {  
    private Internet internet = new ReallInternet();  
    private static List<String> bannedSites;  
  
    static {  
        bannedSites = new ArrayList<String>();  
        bannedSites.add("whatever.com");  
        bannedSites.add("nope.com");  
        bannedSites.add("yup.com");  
        bannedSites.add("hello.com");  
    }  
}
```

Code – Create the proxy (cont'd)

```
@Override  
public void connectTo(String serverhost) throws Exception  
{  
    if(bannedSites.contains(serverhost.toLowerCase()))  
    {  
        throw new Exception("Access Denied");  
    }  
  
    internet.connectTo(serverhost);  
}  
  
}
```

Code – create the client

```
public class Client
{
    public static void main (String[] args)
    {
        Internet internet = new ProxyInternet();
        try
        {
            internet.connectTo("jasonfedin.org");
            internet.connectTo("whatever.com");
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Summary

Jason Fedin

Adapter versus Bridge

- there are many similarities between the structural patterns, especially in their participants and collaborations
- Adapter versus Bridge
 - both promote flexibility by providing a level of indirection to another object
 - both involve forwarding requests to this object from an interface other than its own
- the key difference between these patterns lies in their intents
 - adapter focuses on resolving incompatibilities between two existing interfaces
 - the bridge bridges an abstraction and its (potentially numerous) implementations
 - provides a stable interface to clients even as it lets you vary the classes that implement it
- Adapter and Bridge are often used at different points in the software lifecycle
 - an adapter often becomes necessary when you discover that two incompatible classes should work together
 - to avoid replicating code
 - the user of a bridge understands up-front that an abstraction must have several implementations, and both may evolve independently
 - the Adapter pattern makes things work after they are designed and the Bridge makes them work before they are

Composite versus Decorator

- composite and decorator have similar structure diagrams
 - both rely on recursive composition to organize an open-ended number of objects
- decorator is designed to let you add responsibilities to objects without subclassing
 - avoids the explosion of subclasses that can arise from trying to cover every combination of responsibilities statically
- composite has a different intent than the decorator
 - focuses on structuring classes so that many related objects can be treated uniformly
 - multiple objects can be treated as one
 - focus is not on embellishment but on representation
- the Composite and Decorator patterns are often used together
 - both lead to the kind of design in which you can build applications just by plugging objects together without defining any new classes
 - consists of an abstract class with some subclasses that are composites, some that are decorators, and some that implement the fundamental building blocks of the system
 - both composites and decorators will have a common interface
 - a composite is a ConcreteComponent
 - a decorator is a Leaf

Decorator versus Proxy

- both patterns describe how to provide a level of indirection to an object
- both implementations keep a reference to another object to which they forward requests
- differ by their intent
- the proxy pattern composes an object and provides an identical interface to clients
 - not concerned with attaching or detaching properties dynamically
 - not designed for recursive composition
 - its intent is to provide a stand-in for a subject when it is inconvenient or undesirable to access the subject directly
 - it lives on a remote machine, has restricted access, or is persistent
- the decorator pattern is when the component provides only part of the functionality, and one or more decorators furnish the rest
 - addresses the situation where an object's total functionality cannot be determined at compile time
 - recursive composition an essential part
 - not the case in Proxy
 - focuses on one relationship between the proxy and its subject and that relationship can be expressed statically
- differences do not mean that these patterns can not be combined
 - might envision a proxy-decorator that adds functionality to a proxy, or a decorator-proxy that embellishes a remote object

Behavioral Design Patterns

Jason Fedin

Behavioral Patterns

- these design patterns are specifically concerned with communication between objects
 - characterize complex control flow that is difficult to follow at run-time
 - shift the focus away from flow of control to let you concentrate just on the way objects are interconnected
- these patterns increase flexibility in carrying out this communication
- provide solutions on how to segregate objects to be both dependent and independent
- concerned with algorithms and the assignment of responsibilities between objects

Class Patterns vs. Object Patterns (sub-category)

- behavioral class patterns use inheritance to describe algorithms and flow of control
 - the template method is an abstract definition of an algorithm
 - defines an algorithm step by step
 - a subclass fleshes out the algorithm by defining the abstract operations
- behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone
 - uses object composition rather than inheritance
 - the mediator pattern uses a mediator object for peer object communication
 - mediator provides the indirection needed for loose coupling
- there are eleven behavioral patterns that we will study

Behavioral Patterns that we will study

- Chain of Responsibility Pattern
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern

Behavioral Patterns that we will study (cont'd)

- Observer Pattern
- State Pattern
- Strategy Pattern
- Template Method Pattern
- Visitor Pattern

Chain of Responsibility Pattern

Jason Fedin

Overview

- the chain of responsibility pattern avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
 - chain the receiving objects and pass the request along the chain until an object handles it
- this pattern processes a series of objects one by one (in a sequential manner)
 - a source will initiate this processing
- lets you send requests to an object implicitly through a chain of candidate objects
 - after one's processing is done, if anything is still pending, it can be forwarded to the next object in the chain
 - each receiver contains reference to another receiver
 - we can add new objects anytime (run-time) at the end of a chain

Examples

- in an organization, there are some customer care executives who handle feedback/issues from customers
 - they will forward those customer issues/escalations to the appropriate department in the organization
 - not all departments will start fixing an issue
 - the department that seems to be responsible will take a look first, and if the department staff believe that the issue should be forwarded to another department, he/she will do that
- several class libraries use the Chain of Responsibility pattern to handle user events
 - when the user clicks the mouse or presses a key, an event gets generated and passed along the chain
- consider a context-sensitive help facility for a graphical user interface
 - user can obtain help information on any part of the interface just by clicking on it
 - the help that is provided depends on the part of the interface that is selected and its context
 - a button widget in a dialog box might have different help information than a similar button in the main window
- natural to organize help information according to its generality
 - from the most specific to the most general (a chain)

JDK Examples

- an example in the JDK would be the use of the try/catch blocks
 - every catch block is kind of a processor to process that particular exception
 - when any exception occurs in the try block
 - sends to the first catch block to process
 - If the catch block is not able to process it, it forwards the request to next object in chain (next catch block)
 - If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program
- other packages/methods that use this pattern in the JDK include
 - `java.util.logging.Logger#log()`
 - `javax.servlet.Filter#doFilter()`

when to use this pattern?

- when you want to decouple a request's sender and receiver
- when multiple objects, determined at runtime, are candidates to handle a request
- when you do not want to specify handlers explicitly in your code
- when you want to issue a request to one of several objects without specifying the receiver explicitly
 - we expect any of our receivers to handle that request
- when multiple objects can handle a request and the handler doesn't have to be a specific object

advantages and drawbacks

- decouples the sender of the request and its receivers
 - frees an object from knowing which other object handles a request
 - both the receiver and the sender have no explicit knowledge of each other
- simplifies your object
 - it does not have to know the chain's structure or keep direct references to its members
 - keeps a single reference to their successor
- gives you added flexibility in distributing responsibilities among objects
 - allows you to add or remove responsibilities dynamically by changing the members or order of the chain
- a drawback is that the execution of the request is not guaranteed
 - may fall off the end of the chain if no object handles it
- another drawback is that it can be hard to observe and debug at runtime

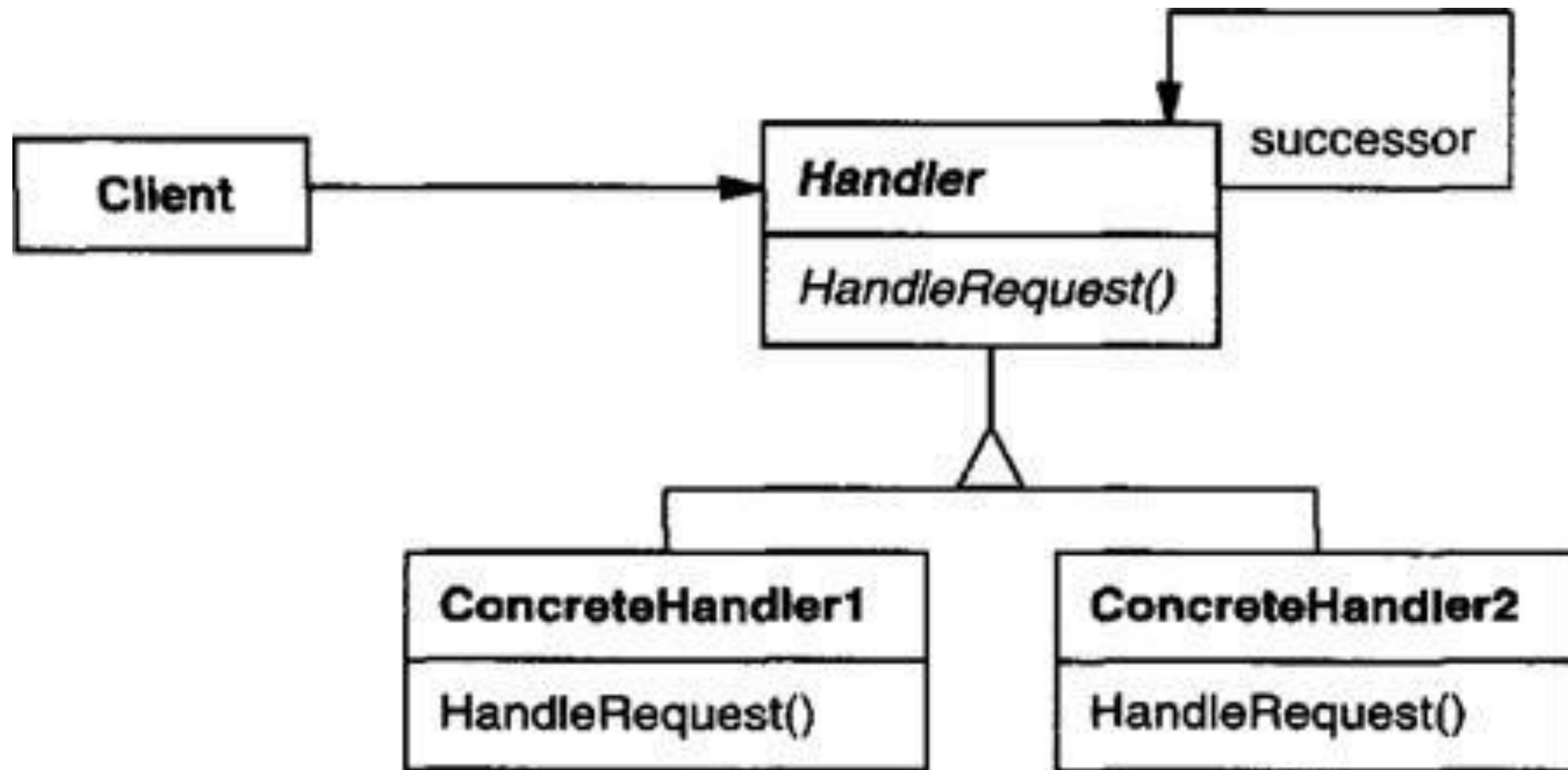
Summary

- with the Chain of Responsibility Pattern, you create a chain of objects to examine requests
- each object in turn examines a request and either handles it, or passes it on to the next object in the chain
- each object in the chain acts as a handler and has a successor object

Implementing the chain of responsibility

Jason Fedin

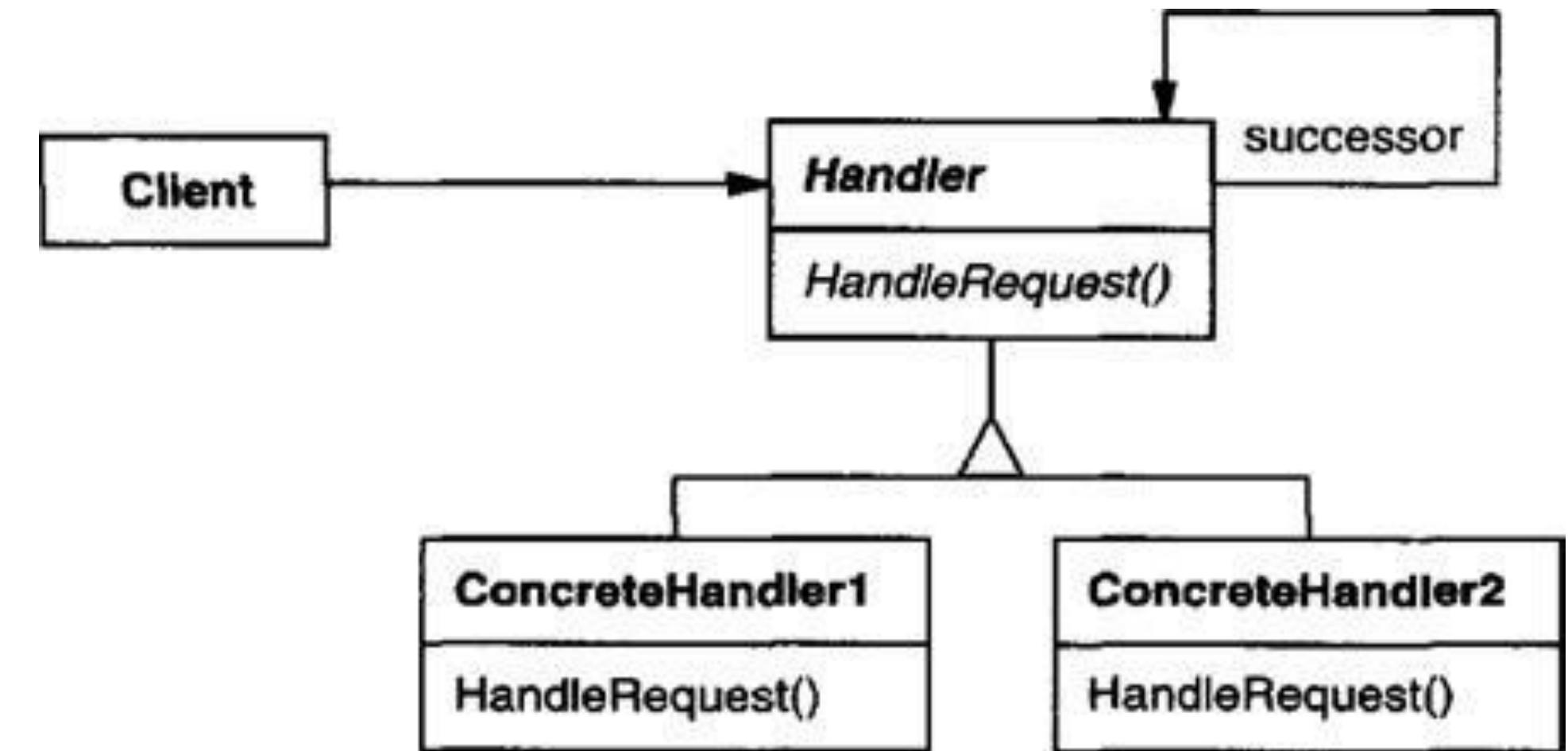
Class Diagram



Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Participants

- Handler
 - defines an interface for handling requests
 - (optional) implements the successor link
 - dispatches the request to chain of handlers
- ConcreteHandler
 - handles requests it is responsible for
 - can access its successor
 - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
- Client
 - initiates the request to a ConcreteHandler object on the chain
- when a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it



Important implementation considerations

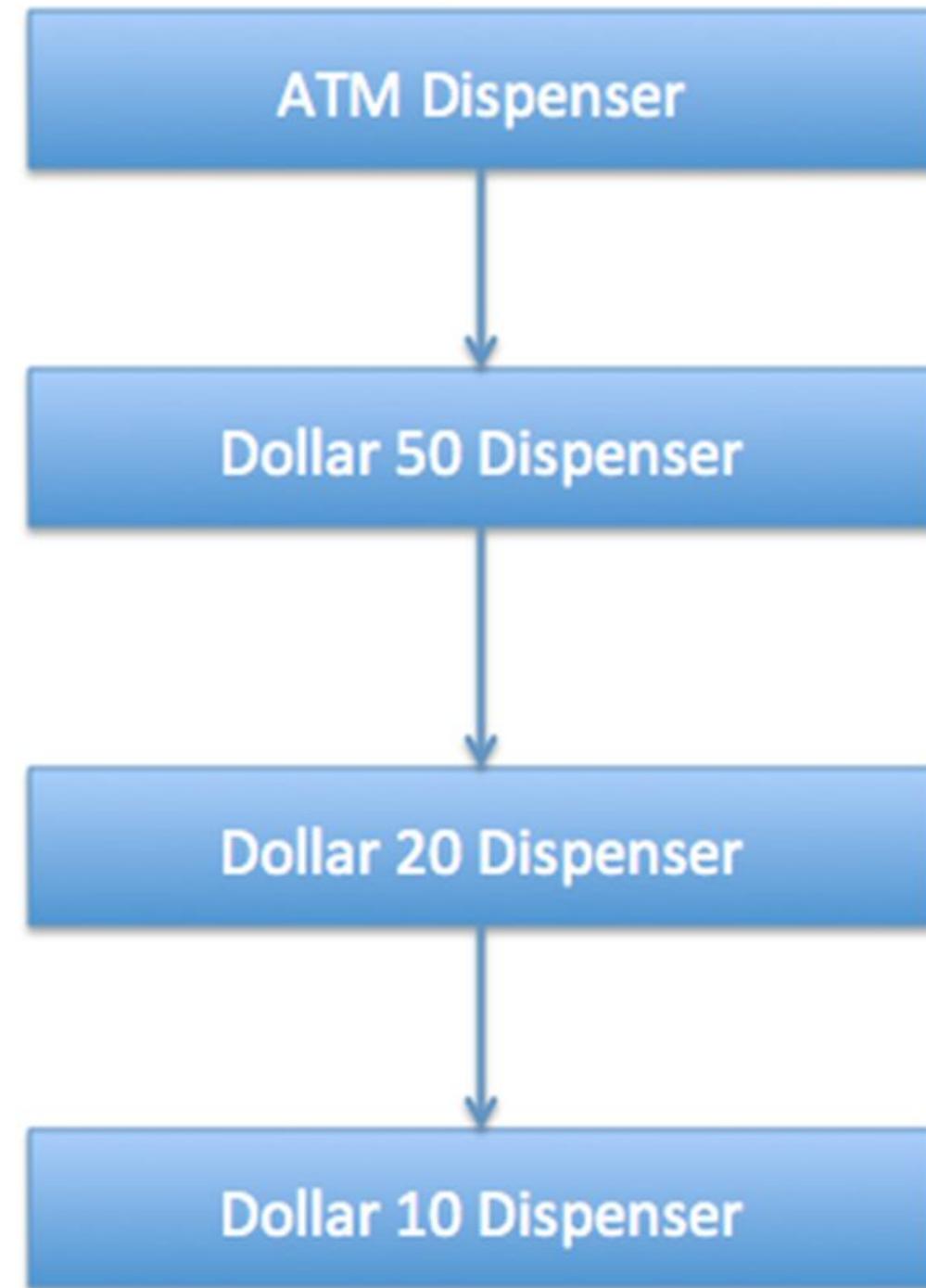
- client does not know which part of the chain will be processing the request
 - it will send the request to the first object in the chain
- each object in the chain will have its own implementation to process the request
 - either full or partial or to send it to the next object in the chain
- every object in the chain should have a reference to the next object in chain to forward the request to
 - this is achieved by composition
- creating the chain carefully is very important
 - there might be a case that the request will never be forwarded to a particular processor or there are no objects in the chain who are able to handle the request
 - add a check to make sure it gets processed fully by all the processors
 - or throw an exception if the request reaches the last object and there are no further objects in the chain to forward the request to

Example in intellij

- lets look at an example of an ATM machine
- user enters the amount to be dispensed and the machine dispense amount in terms of defined currency bills
 - if the user enters an amount that is not multiples of 10, it throws an error
- we could implement this solution easily in a single program itself but then the complexity will increase and the solution will be tightly coupled
- we will use the chain of responsibility to implement a solution for dispensing the correct denomination of bills
 - the chain will process the request in order of highest denomination (50->20->10)

Illustration of chain

Enter amount to dispense in multiples of 10



Create Helper classes (Currency.java)

- create a Currency class that will store the amount to dispense and used by the chain implementations

```
public class Currency {  
  
    private int amount;  
  
    public Currency(int amt) {  
        this.amount=amt;  
    }  
  
    public int getAmount(){  
        return this.amount;  
    }  
}
```

Create the handler interface

- defines an interface for handling requests
- has a method to define the next processor in the chain and the method that will process the request

```
public interface DispenseChain {  
  
    void setNextChain(DispenseChain nextChain);  
  
    void dispense(Currency cur);  
}
```

Create the Concrete Handlers

- we need to create different processor classes that will implement the DispenseChain interface
- will provide implementation of dispense methods
- since we are developing our system to work with three types of currency bills – 50\$, 20\$ and 10\$, we will create three concrete implementations
- handles requests it is responsible for
- can access its successor
- if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor

Create the Concrete Handlers (Dollar50Dispenser.java)

```
public class Dollar50Dispenser implements DispenseChain {  
    private DispenseChain chain;  
  
    @Override  
    public void setNextChain(DispenseChain nextChain) {  
        this.chain=nextChain;  
    }  
  
    @Override  
    public void dispense(Currency cur) {  
        if(cur.getAmount() >= 50){  
            int num = cur.getAmount()/50;  
            int remainder = cur.getAmount() % 50;  
            System.out.println("Dispensing "+num+" 50$ note");  
            if(remainder !=0) this.chain.dispense(new Currency(remainder));  
        } else{  
            this.chain.dispense(cur);  
        }  
    }  
}
```

Create the concrete handlers (Dollar20Dispenser.java)

```
public class Dollar20Dispenser implements DispenseChain {  
    private DispenseChain chain;  
  
    @Override  
    public void setNextChain(DispenseChain nextChain) {  
        this.chain=nextChain;  
    }  
  
    @Override  
    public void dispense(Currency cur) {  
        if(cur.getAmount() >= 20){  
            int num = cur.getAmount()/20;  
            int remainder = cur.getAmount() % 20;  
            System.out.println("Dispensing "+num+" 20$ note");  
            if(remainder !=0) this.chain.dispense(new Currency(remainder));  
            }else{  
                this.chain.dispense(cur);  
            }  
    }  
}
```

Create the concrete handlers (Dollar10Dispenser.java)

```
public class Dollar10Dispenser implements DispenseChain {  
    private DispenseChain chain;  
  
    @Override  
    public void setNextChain(DispenseChain nextChain) {  
        this.chain=nextChain;  
    }  
  
    @Override  
    public void dispense(Currency cur) {  
        if(cur.getAmount() >= 10){  
            int num = cur.getAmount()/10;  
            int remainder = cur.getAmount() % 10;  
            System.out.println("Dispensing "+num+" 10$ note");  
            if(remainder !=0) this.chain.dispense(new Currency(remainder));  
        } else{  
            this.chain.dispense(cur);  
        }  
    }  
}
```

Important points for concrete handlers

- here is the implementation of dispense method
- you will notice that every implementation is trying to process the request and based on the amount, it might process some or full part of it
- if one of the chain not able to process it fully, it sends the request to the next processor in chain to process the remaining request
- If the processor is not able to process anything, it just forwards the same request to the next chain

Create the client

- this is a very important step and we should create the chain carefully, otherwise a processor might not be getting any request at all
- for example, in our implementation if we keep the first processor chain as Dollar10Dispenser and then Dollar20Dispenser, then the request will never be forwarded to the second processor and the chain will become useless

Create the client (ATMDispenseChain.java)

```
import java.util.Scanner;

public class ATMDispenseChain {
    private DispenseChain c1;

    public ATMDispenseChain() {
        // initialize the chain
        this.c1 = new Dollar50Dispenser();
        DispenseChain c2 = new Dollar20Dispenser();
        DispenseChain c3 = new Dollar10Dispenser();

        // set the chain of responsibility
        c1.setNextChain(c2);
        c2.setNextChain(c3);
    }
}
```

Create the client (ATMDispenseChain.java)

```
public static void main(String[] args) {  
    ATMDispenseChain atmDispenser = new ATMDispenseChain();  
    while (true) {  
        int amount = 0;  
        System.out.println("Enter amount to dispense");  
        Scanner input = new Scanner(System.in);  
        amount = input.nextInt();  
        if (amount % 10 != 0) {  
            System.out.println("Amount should be in multiple of 10s.");  
            return;  
        }  
        // process the request  
        atmDispenser.c1.dispense(new Currency(amount));  
    }  
}
```

Output

Enter amount to dispense

530

Dispensing 10 50\$ note

Dispensing 1 20\$ note

Dispensing 1 10\$ note

Enter amount to dispense

100

Dispensing 2 50\$ note

Enter amount to dispense

120

Dispensing 2 50\$ note

Dispensing 1 20\$ note

Enter amount to dispense

15

Amount should be in multiple of 10s.

Demonstration

Jason Fedin

Code – Helper classes

```
enum MessagePriority {  
    Normal,  
    High  
}  
  
class Message {  
    public String Text;  
    public MessagePriority Priority;  
    public Message(String msg, MessagePriority p) {  
        Text = msg;  
        this.Priority = p;  
    }  
}
```

Create the Handler Interface

```
interface IReceiver  
{  
    boolean ProcessMessage(Message msg);  
    void setNextChain(IReceiver nextChain);  
}
```

Helper Class for first in chain (used by client)

```
class IssueRaiser {  
    public IReceiver setFirstReceiver;  
  
    public IssueRaiser(IReceiver firstReceiver) {  
        this.setFirstReceiver = firstReceiver;  
    }  
  
    public void RaiseMessage(Message msg) {  
        if (setFirstReceiver != null)  
            setFirstReceiver.ProcessMessage(msg);  
    }  
}
```

Create the concrete handler classes

```
class FaxErrorHandler implements IReceiver {  
    private IReceiver _nextReceiver;  
    public void setNextChain(IReceiver nextReceiver) {  
        _nextReceiver = nextReceiver;  
    }  
  
    public Boolean ProcessMessage(Message msg) {  
        if (msg.Text.contains("Fax")) {  
            System.out.println("FaxErrorHandler processed "+ msg.Priority+ "priority issue: "+ msg.Text);  
            return true;  
        }  
        else {  
            if (_nextReceiver != null)  
                _nextReceiver.ProcessMessage(msg);  
        }  
        return false;  
    }  
}
```

Create the concrete handler classes

```
class EmailErrorHandler implements IReceiver {  
    private IReceiver _nextReceiver;  
  
    public void setNextChain(IReceiver nextReceiver) {  
        _nextReceiver = nextReceiver;  
    }  
  
    public Boolean ProcessMessage(Message msg) {  
        if (msg.Text.contains("Email")) {  
            System.out.println("EmailErrorHandler processed "+ msg.Priority+ "priority issue: "+ msg.Text);  
            return true;  
        }  
        else {  
            if (_nextReceiver != null)  
                _nextReceiver.ProcessMessage(msg);  
        }  
        return false;  
    }  
}
```

Create the client

```
class ChainOfResponsibilityPatternEx {  
    public static void main(String[] args) {  
        System.out.println("/**Chain of Responsibility Pattern Demo**\\n");  
  
        //Making the chain first: IssueRaiser->FaxErrorHandler->EmailErrorHandler  
        IReceiver faxHandler, emailHandler;  
  
        //end of chain  
        emailHandler = new EmailErrorHandler();  
  
        //fax handler is before email  
        faxHandler = new FaxErrorHandler();  
        faxHandler.setNextChain(emailHandler);  
  
        //starting point: raiser will raise issues and set the first handler  
        IssueRaiser raiser = new IssueRaiser (faxHandler);
```

Create the client

```
Message m1 = new Message("Fax is reaching late to the destination", MessagePriority.Normal);
Message m2 = new Message("Email is not going", MessagePriority.High);
Message m3 = new Message("In Email, BCC field is disabled occasionally", MessagePriority.Normal);
Message m4 = new Message("Fax is not reaching destination", MessagePriority.High);

raiser.RaiseMessage(m1);
raiser.RaiseMessage(m2);
raiser.RaiseMessage(m3);
raiser.RaiseMessage(m4);

}
```

Output

```
Console ×
<terminated> ChainOfResponsibilityPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 18, 2015, 2:46:10 PM)
***Chain of Responsibility Pattern Demo***

FaxErrorHandler processed Normalpriority issue: Fax is reaching late to the destination
EmailErrorHandler processed Highpriority issue: Email is not going
EmailErrorHandler processed Normalpriority issue: In Email, BCC field is disabled occationally
FaxErrorHandler processed Highpriority issue: Fax is not reaching destination
```

The Command Design Pattern

Jason Fedin

Overview

- the command design pattern encapsulates a request as an object (command)
 - lets you parameterize clients with different requests
 - you can queue or log these requests as well as supporting “undoing” these requests
- the Command object encapsulates a request by binding together a set of actions on a specific receiver
 - does this by exposing just one method execute() that causes some actions to be invoked on the receiver
- the Command’s execute method can store state for reversing its effects of the Command itself thus supporting an “undo” operation
 - or you can even add an unExecute method that reverses the effects of a previous call to execute
- the pattern can also support logging changes so that they can be reapplied in case of a system crash

Overview (cont'd)

- this pattern is a data driven design pattern
- it is widely used for undo/redo operations
- a callback function can be designed with this pattern
- it is very useful when we model transactions (which can be responsible for changes in data)
- makes our code extensible as we can add new commands without changing existing code
- this pattern allows us to issue requests to objects without knowing anything about the operation being requested or the receiver of the request

Examples

- we could use the command pattern when placing orders to buy and sell stocks
 - we would create an interface Order which acts as a command (buying and selling as concrete classes)
 - a Stock class could act as the request
 - a class Broker could act as an invoker object
 - takes and places orders
- the broker object uses command pattern to identify which object will execute which command based on the type of command
- another example would be a user interface toolkit
 - includes objects like buttons and menus
 - carry out requests in response to user input
 - the toolkit cannot implement the request explicitly in the button or menu
 - only applications that use the toolkit know what should be done on which object
 - no way of knowing the receiver of the request or the operations that will carry it out
- the Command pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object
 - object can be stored and passed around like other objects

Examples (undo and redo)

- we can undo and redo many operations in our daily life
 - we can erase a pencil drawing with a rubber
 - we can re-architect our living places
 - we can forget bad memories and start a fresh journey
- undo/redo operations are part of our life
 - we are doing these operations through some commands
 - either externally or internally
- the above scenario also applies with the Microsoft paint application
 - we can do the undo/redo operations easily through some menu options or shortcut keys

When to use the command pattern?

- when you want to parameterize objects by an action to perform
- when you want to specify, queue, and execute requests at different times
 - a Command object can have a lifetime independent of the original request
- when you want to support undo
 - the Command's execute method can store state for reversing its effects in the command itself
- when you want to support logging changes so that they can be reapplied in case of a system crash
 - can augment the Command interface with load and store operations
 - keep a persistent log of changes
- when you want to implement a callback method

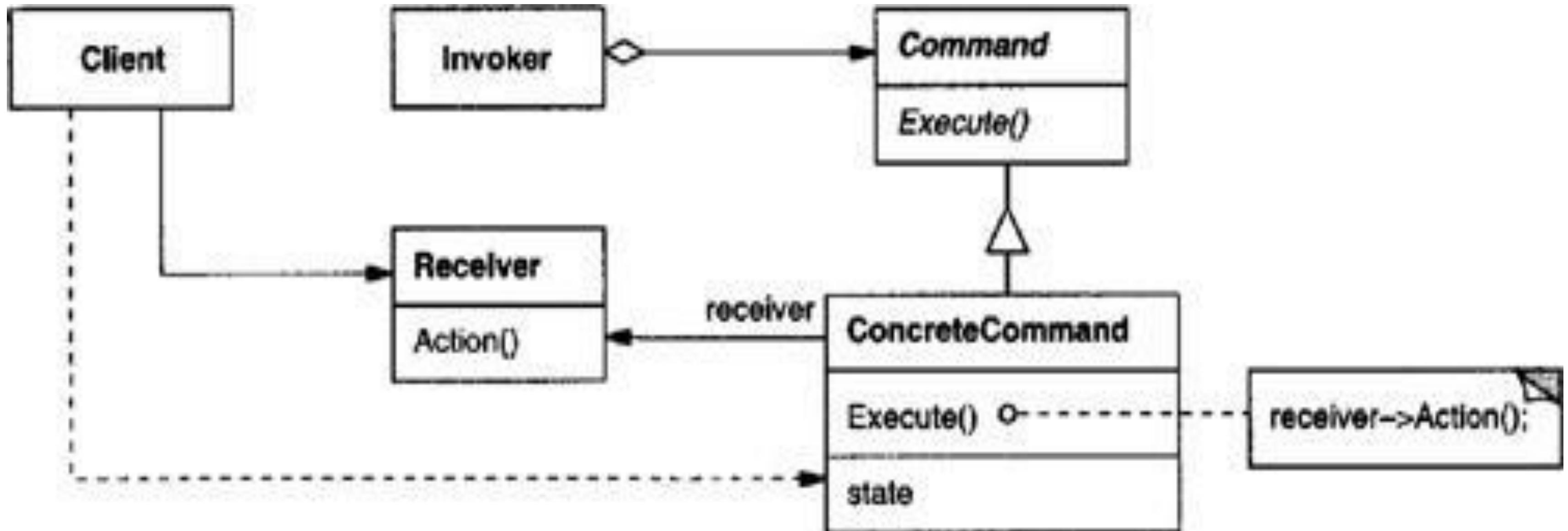
Summary

- the Command Pattern decouples an object making a request from the one that knows how to perform it
- a Command object is at the center of this decoupling and encapsulates a receiver with an action (or set of actions)
- commands may support undo by implementing an undo method that restores the object to its previous state
- commands may also be used to implement logging and transactional systems
- commands can be extended easily
 - while we use them, we do not need to change the classes in the system
- in the chain of responsibility pattern we forward a request along a chain of objects with the hope that any one of the objects along that chain will handle the request
- in the command pattern we will forward the request to a specific object

Implementing the Command Pattern

Jason Fedin

Class Diagram

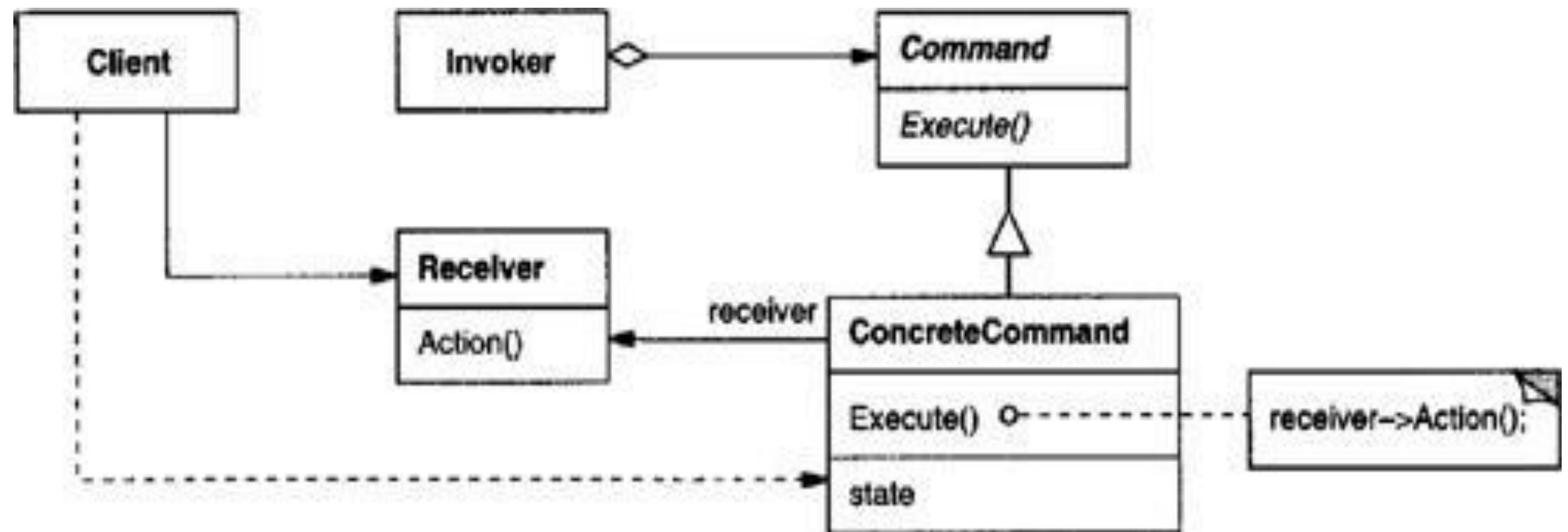


Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Overview

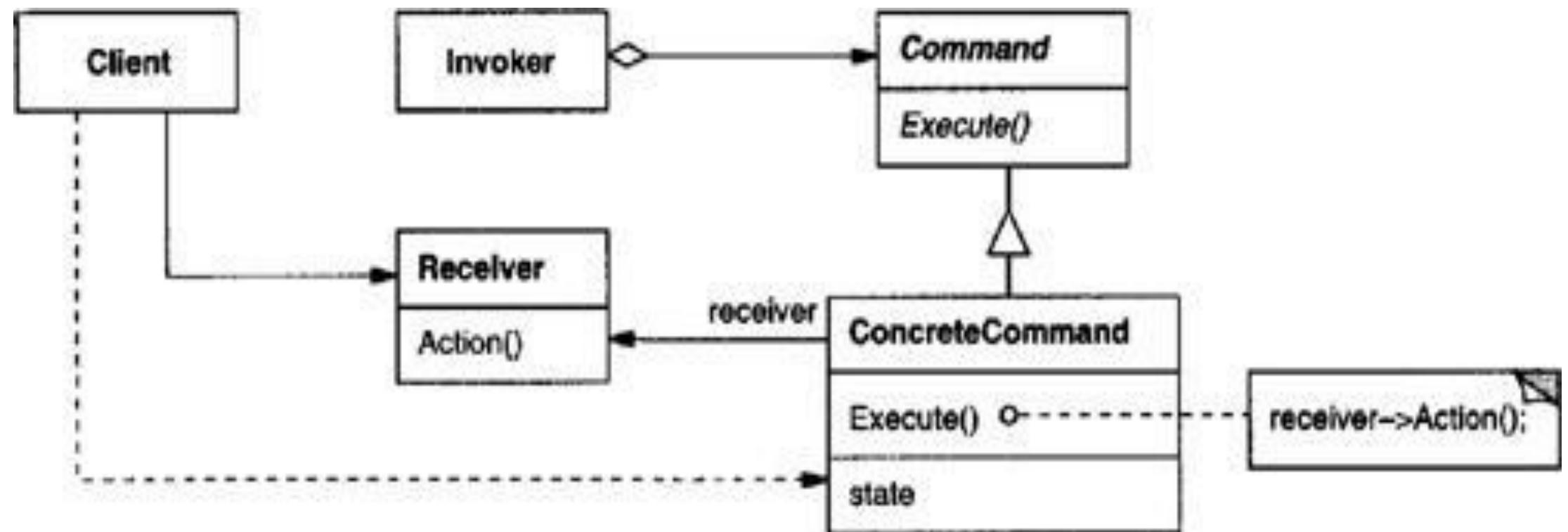
- remember, requests are encapsulated as objects
- four terms are associated with the implementation of this pattern
 - invoker, client, command, and receiver
- a command object is capable of calling a particular method in the receiver
- an invoker only knows about the command interface
 - it is totally unaware about the concrete commands
- the client object holds the invoker object and the command object(s)
 - the client decides which of these commands needs to execute at a particular point in time

Participants



- **Command**
 - declares an interface for all commands
 - a command is invoked through its execute method
 - will ask a receiver to perform an action
 - may include an undo() method
- **ConcreteCommand**
 - defines a binding between a Receiver object and an action
 - invoker makes a request by calling execute() and this class carries it out by calling one or more actions on the Receiver

Participants



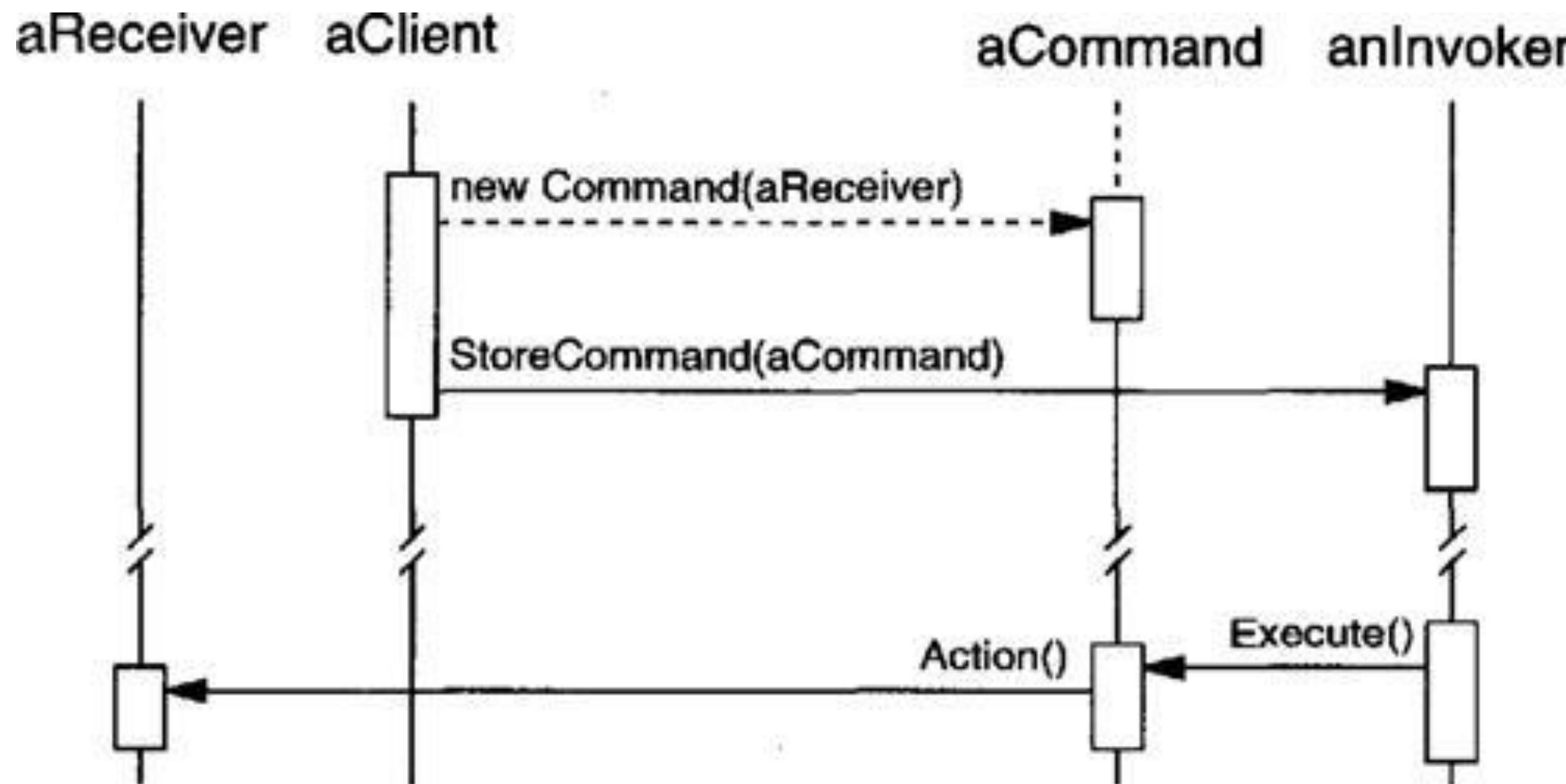
- Client
 - creates a **ConcreteCommand** object and sets its receiver
- Invoker
 - asks the command to carry out the request by calling its **execute()** method
- Receiver
 - knows how to perform the work need to carry out a request
 - any class may serve as a Receiver

Workflow

- the client creates a ConcreteCommand object and specifies its receiver
 - creates the receiver object as well and then attaches it to the Command (in the constructor)
- an Invoker object stores the ConcreteCommand object
 - client creates the invoker object and attaches the command object (in the constructor)
 - may also instead just pass the command to the execute method
- the invoker issues a request by calling Execute, using the command object (previously attached or passed in as a parameter to execute)
 - when commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute
- when client program executes the action (via the invoker), it is processed based on the command and receiver object
 - the ConcreteCommand object invokes the appropriate method on its receiver to carry out the specific request

Sequence Diagram

- shows the interactions between the objects
 - the Command decouples the invoker from the receiver (and the request it carries out)



Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Implementation details

- the Command decouples the object that invokes the operation from the one that knows how to perform it
- Commands are first-class objects
 - can be manipulated and extended like any other object
- Command objects can be:
 - dumb
 - delegates the required action to a receiver object
 - smart
 - Implements everything itself without delegating to a receiver object
- pattern is easily extendible
 - we can add new action methods in receivers and create new Command implementations without changing the client code
- one drawback is that the code gets huge and confusing with a high number of action methods
 - you end up with a lot of small classes

Summary

- the Command object is the core of command design pattern that defines the contract for implementation
- receiver implementation is separate from command implementation
- command implementation classes choose the method to invoke on receiver object
 - for every method in receiver there will be a command implementation
 - works as a bridge between receiver and action methods
- invoker class just forwards the request from client to the command object
- client is responsible for instantiating appropriate command and receiver objects and then associating them together
- client is also responsible for instantiating invoker object and associating command object with it and executing the action method

Example in intelliJ

Jason Fedin

Example

- lets look at a real life scenario where we can implement Command pattern
- this example will implement a File System utility with methods to open, write and close files
- this file system utility will support multiple operating systems such as Windows and Unix

Create the Receiver interface

- to implement our File System utility, first of all we need to create the receiver classes that will actually do all the work (open, close, and write to a file)
- we can have a FileSystemReceiver interface and its implementation classes for different operating system flavors such as Windows, Unix, Solaris etc

```
public interface FileSystemReceiver {  
    void openFile();  
    void writeFile();  
    void closeFile();  
}
```

Create the concrete receiver classes

- FileSystemReceiver interface defines the contract for the implementation classes
- I am creating two flavors of receiver classes to work with Unix and Windows systems

```
public class UnixFileSystemReceiver implements FileSystemReceiver {  
    @Override  
    public void openFile() {  
        System.out.println("Opening file in unix OS");  
    }  
  
    @Override  
    public void writeFile() {  
        System.out.println("Writing file in unix OS");  
    }  
  
    @Override  
    public void closeFile() {  
        System.out.println("Closing file in unix OS");  
    }  
}
```

Create the WindowsFileSystemReceiver class

```
public class WindowsFileSystemReceiver implements FileSystemReceiver {  
  
    @Override  
    public void openFile() {  
        System.out.println("Opening file in Windows OS");  
    }  
  
    @Override  
    public void writeFile() {  
        System.out.println("Writing file in Windows OS");  
    }  
  
    @Override  
    public void closeFile() {  
        System.out.println("Closing file in Windows OS");  
    }  
}
```

Create the command interface

- now that our receiver classes are ready, we can move to implement our Command classes
- we can use interface or abstract class to create our base Command, it's a design decision and depends on your requirement.
- we are going with interface because we don't have any default implementations

```
public interface Command {  
    void execute();  
  
    // could add an undo or redo command to the interface to undo previous  
    // command or redo previous commands  
}
```

Create the concrete command implementations

- now we need to create implementations for all the different types of actions performed by the receiver
- since we have three actions we will create three Command implementations
 - each Command implementation will forward the request to the appropriate method of receiver
- if we wanted to add an undo command, we would add it to the interface and store the previous state in this class

```
public class OpenFileCommand implements Command {  
    private FileSystemReceiver fileSystem;  
    // store previous state for undo, String someState;  
    // myMember variable  
  
    public OpenFileCommand(FileSystemReceiver fs){  
        this.fileSystem=fs;  
    }  
    @Override  
    public void execute() {  
        // save previous state, in case undo called someState = .....;  
        this.fileSystem.openFile();  
    }  
  
    public void undo() {  
        /// restore previous state  
        // myMemberVariable = previousState;  
    }  
}
```

Create the CloseFileCommand implementation

```
public class CloseFileCommand implements Command {  
    private FileSystemReceiver fileSystem;  
  
    public CloseFileCommand(FileSystemReceiver fs){  
        this.fileSystem=fs;  
    }  
    @Override  
    public void execute() {  
        this.fileSystem.closeFile();  
    }  
}
```

Create the WriteFileCommand implementation

```
public class WriteFileCommand implements Command {  
    private FileSystemReceiver fileSystem;  
  
    public WriteFileCommand(FileSystemReceiver fs){  
        this.fileSystem=fs;  
    }  
    @Override  
    public void execute() {  
        this.fileSystem.writeFile();  
    }  
}
```

Create the invoker class

- now we have receiver and command implementations ready, so we can move to implement the invoker class
- invoker is a simple class that encapsulates the Command and passes the request to the command object to process it

```
public class FileInvoker {  
    public Command command;  
  
    public FileInvoker(Command c){  
        this.command=c;  
    }  
  
    public void execute(){  
        this.command.execute();  
    }  
}
```

Create a helper class to get the right file system

- our file system utility implementation is ready and we can move to write a simple command pattern client program
 - but before that I will provide a utility method to create the appropriate FileSystemReceiver object
- since we can use System class to get the operating system information, we will use this or else we can use Factory pattern to return appropriate type based on the input

```
public class FileSystemReceiverUtil {  
    public static FileSystemReceiver getUnderlyingFileSystem(){  
        String osName = System.getProperty("os.name");  
        System.out.println("Underlying OS is:"+osName);  
        if(osName.contains("Windows")){  
            return new WindowsFileSystemReceiver();  
        }else{  
            return new UnixFileSystemReceiver();  
        }  
    }  
}
```

Create the Client

- lets move now to create our command pattern example client program that will consume our file system utility
- notice that client is responsible to create the appropriate type of command object
 - for example if you want to write a file you are not supposed to create CloseFileCommand object
- client program is also responsible to attach receiver to the command and then command to the invoker class

```
public class FileSystemClient {  
  
    public static void main(String[] args) {  
        //Creating the receiver object  
        FileSystemReceiver fs = FileSystemReceiverUtil.getUnderlyingFileSystem();  
  
        //creating command and associating with receiver  
        OpenFileCommand openFileCommand = new OpenFileCommand(fs);  
  
        //Creating invoker and associating with Command  
        FileInvoker file = new FileInvoker(openFileCommand);
```

Client class continued

```
//perform action on invoker object  
file.execute();
```

```
WriteFileCommand writeFileCommand = new WriteFileCommand(fs);  
file = new FileInvoker(writeFileCommand);  
file.execute();
```

```
CloseFileCommand closeFileCommand = new CloseFileCommand(fs);  
file = new FileInvoker(closeFileCommand);  
file.execute();
```

```
}
```

```
}
```

Output

Underlying OS is:Mac OS X

Opening file in unix OS

Writing file in unix OS

Closing file in unix OS

Demonstration

Jason Fedin

Create a command interface (Order.java)

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

Create the receiver class (Stock.java)

```
public class Stock {  
  
    private String name = "Google";  
    private int quantity = 1000;  
  
    public void buy(){  
        System.out.println("Stock [ Name: "+name+",  
                           Quantity: " + quantity +" ] bought");  
    }  
    public void sell(){  
        System.out.println("Stock [ Name: "+name+",  
                           Quantity: " + quantity +" ] sold");  
    }  
}
```

Create the concrete command classes (BuyStock.java)

```
public class BuyStock implements Order {  
    private Stock myStock;  
  
    public BuyStock(Stock someStock){  
        this.myStock = someStock;  
    }  
  
    public void execute() {  
        myStock.buy();  
    }  
}
```

Create the concrete command classes (SellStock.java)

```
public class SellStock implements Order {
```

```
    private Stock myStock;
```

```
    public SellStock(Stock someStock){
```

```
        this.myStock = someStock;
```

```
}
```

```
    public void execute() {
```

```
        myStock.sell();
```

```
}
```

```
}
```

Create the invoker class (Broker.java)

```
import java.util.ArrayList;  
import java.util.List;  
  
public class Broker {  
    private List<Order> orderList = new ArrayList<Order>();  
  
    public void takeOrder(Order order){  
        orderList.add(order);  
    }  
  
    public void placeOrders(){  
  
        for (Order order : orderList) {  
            order.execute();  
        }  
        orderList.clear();  
    }  
}
```

Create the client (use the broker class to take and execute commands)

```
public class CommandPatternDemo {  
    public static void main(String[] args) {  
        Stock googleStock = new Stock();  
  
        BuyStock buyStockOrder = new BuyStock(googleStock);  
        SellStock sellStockOrder = new SellStock(googleStock);  
  
        Broker broker = new Broker();  
        broker.takeOrder(buyStockOrder);  
        broker.takeOrder(sellStockOrder);  
  
        broker.placeOrders();  
    }  
}
```

The Interpreter Design Pattern

Jason Fedin

Overview

- the interpreter design pattern provides a way to evaluate a language grammar or expression
- the formal definition is “Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language”
 - we define a grammatical representation for a language and provide an interpreter to deal with that grammar
- involves implementing an expression interface which tells how to interpret a particular context
- the Interpreter pattern requires some knowledge of formal grammars

Grammars

- a grammar is a way to represent valid sentences in a language
- it defines a language using special symbols and syntax
- it is also defined by specifying a number of rules
 - each rule specifying how a single symbol can be replaced by one of a selection of sequences of atoms and symbols
- a grammar consists of the following
 - a set of variables or non-terminal symbols
 - a set of Terminal symbols
 - S is a special variable called the Start symbol
 - P represents Production rules for Terminals and Non-terminals

Grammar Example

$\{S, A, B\}$
 $\{a, b\}$
 S
 $\{S \rightarrow AB, A \rightarrow a, B \rightarrow b\}$

- S , A , and B are Non-terminal symbols (variables)
- a and b are Terminal symbols (actual characters)
- S is the Start symbol
- Productions, $P : S \rightarrow AB, A \rightarrow a, B \rightarrow b$
 - the production rules are applied in any order, until a string that contains neither the start symbol nor designated nonterminal symbols is produced
- a single rule is applied to a string by replacing one occurrence of the production rule's left-hand side in the string by that production rule's right-hand side
 - the language formed by the grammar consists of all distinct strings that can be generated in this manner
 - any particular sequence of production rules on the start symbol yields a distinct string in the language

Back to the interpreter

- when you need to implement a simple language you can use the Interpreter pattern
 - describes how to define a grammar for simple languages, represent sentences in the language, and interpret these sentences
- to represent the language, you use a class to represent each rule in the language
- searching for strings that match a pattern is a common problem
 - regular expressions are a standard language for specifying patterns of strings
 - search algorithms could interpret a regular expression that specifies a set of strings to match
- the pattern describes how to define a grammar for regular expressions, represent a particular regular expression, and how to interpret that regular expression

Examples

- a language translator who translates a language for us
 - google translator where the input can be in any language and we can get the output interpreted in another language
- we can consider music notes as a grammar and musicians as our interpreters
- a Java compiler interprets the source code into byte code
 - byte code is understandable by JVM (Java virtual machine)
- widely used to interpret the statements in a language as abstract syntax trees
- used in SQL parsing and a symbol processing engine
- `java.util.Pattern` and subclasses of `java.text.Format` are some of the examples of interpreter pattern used in JDK

advantages and drawbacks

- easy to implement if each grammar rule is represented by a class
 - allows you to easily change or extend the language
 - by adding methods to the class structure, you can add new behaviors beyond interpretation
- the pattern makes it easier to evaluate an expression in a new way
 - you can support pretty printing or type-checking an expression by defining a new operation on the expression classes
- one drawback is that when the number of grammar rules is large, it is harder to maintain the code
 - the Interpreter pattern defines at least one class for every rule in the grammar
 - in these cases a parser/compiler generator may be more appropriate
- requires a lot of error checking and a lot of expressions and code to evaluate

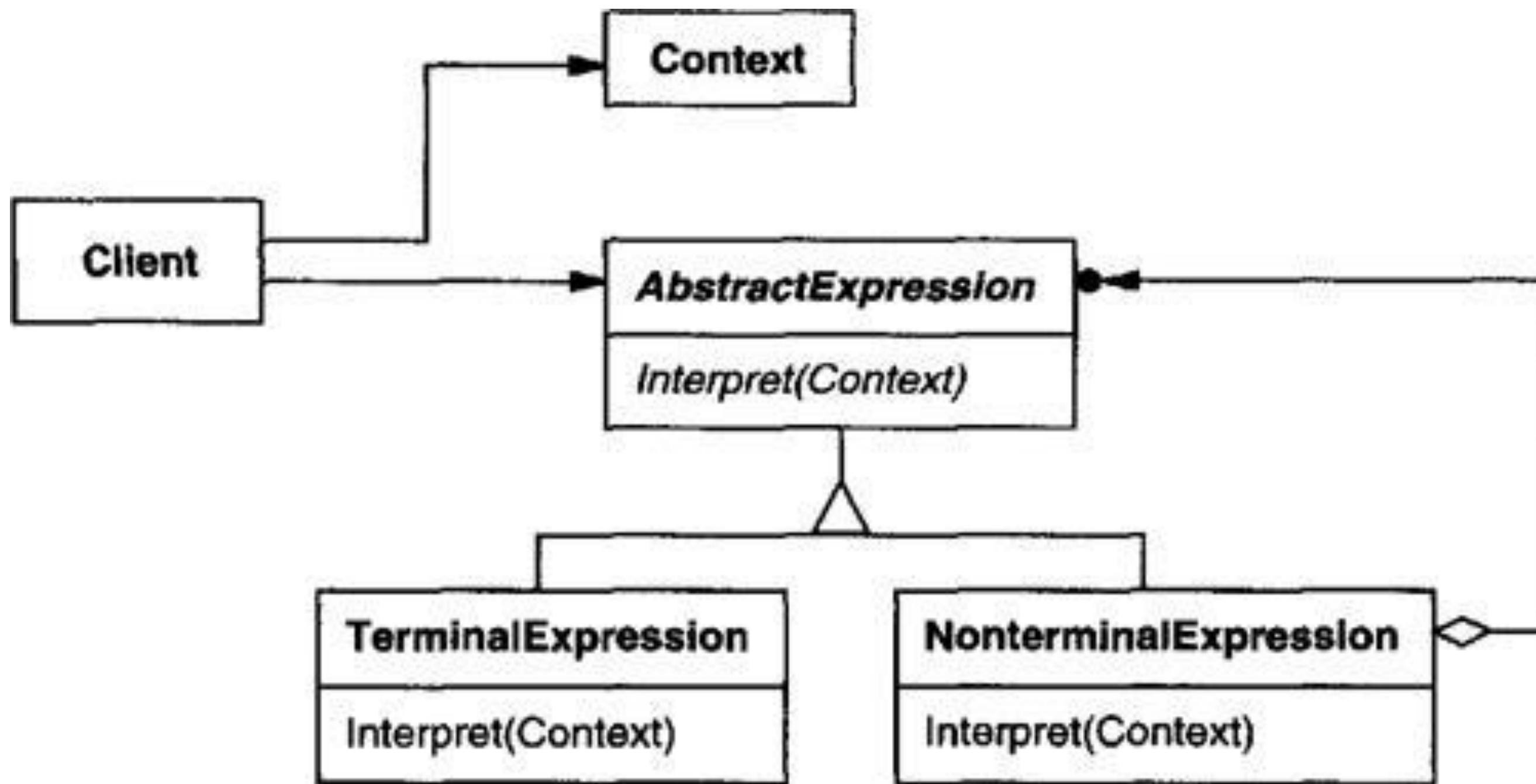
when to use this pattern??

- use the Interpreter pattern when there is a language to interpret and when the language is simple (grammar does not have many rules)
 - should be able to represent statements in the language as abstract syntax trees
- appropriate when simplicity is more important than efficiency
 - most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form
 - regular expressions are often transformed into state machines
- used for scripting and programming languages

Implementing the Interpreter

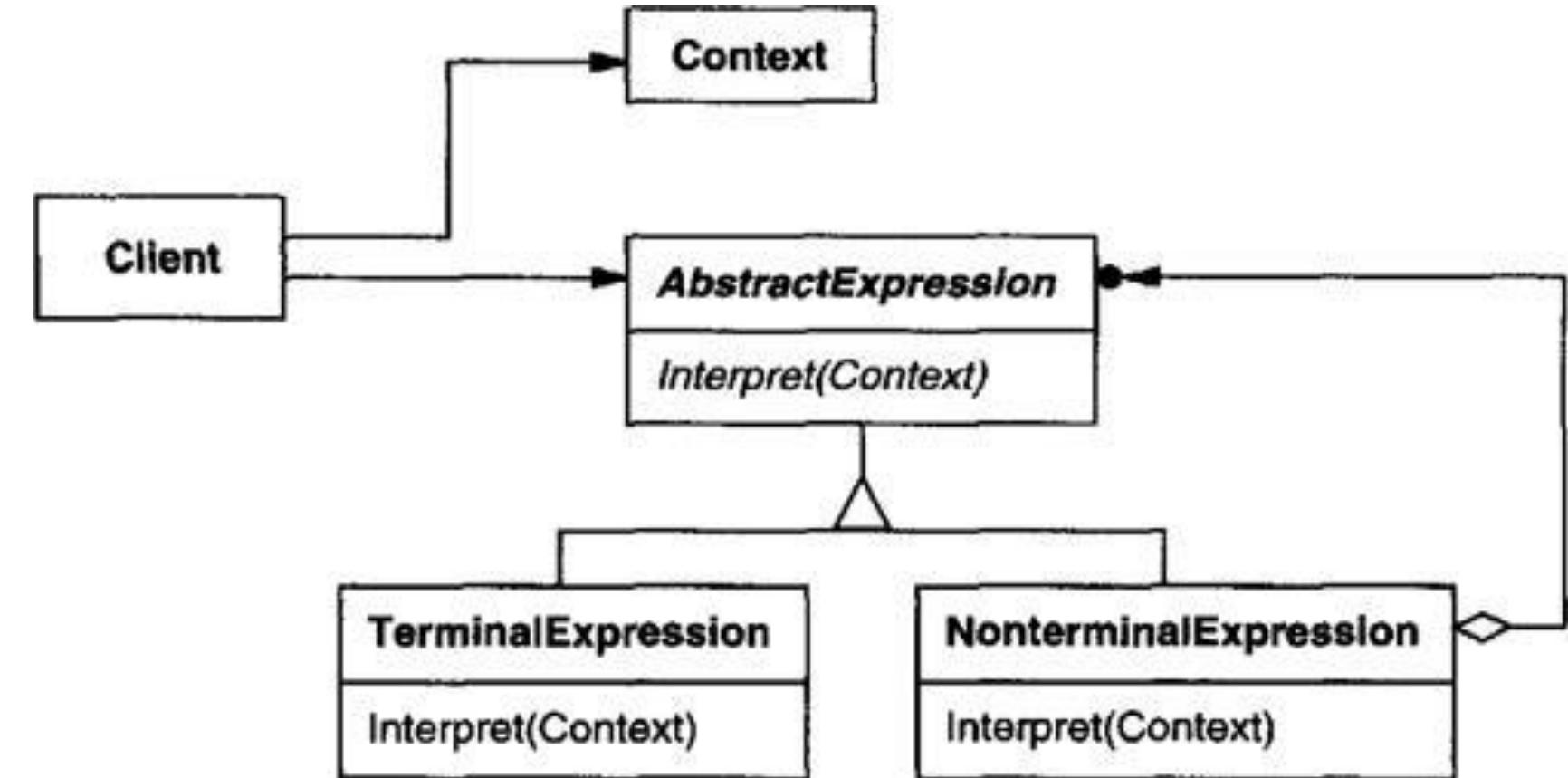
Jason Fedin

Overview



Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

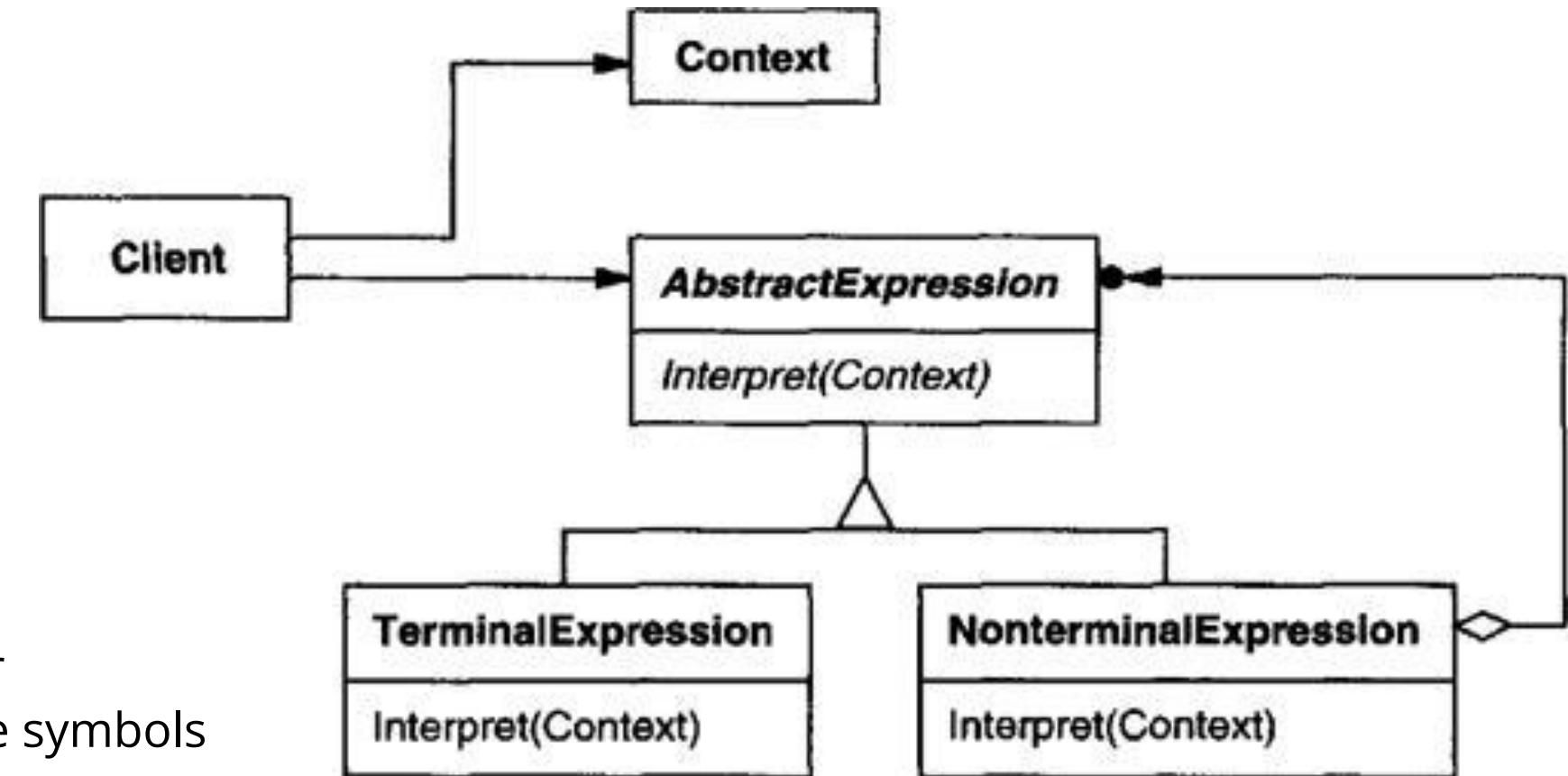
Participants



- **AbstractExpression**
 - declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree
- **TerminalExpression**
 - implements an Interpret operation associated with terminal symbols in the grammar
 - an instance is required for every terminal symbol in a sentence

Participants

- NonterminalExpression
 - one such class is required for every rule $R ::= R_1 R_2 \dots R_n$ in the grammar
 - maintains instance variables of type AbstractExpression for each of the symbols R_1 through R_n
 - implements an Interpret operation for nonterminal symbols in the grammar
 - Interpret typically calls itself recursively on the variables representing R_1 through R_n
- Context
 - contains information that is global to the interpreter
- Client
 - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines
 - the abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes
 - invokes the Interpret operation



Summary of participants roles

- the client builds (or is given) the sentence as an abstract syntax tree of NonTerminalExpression and TerminalExpression instances
 - the client will then initialize the context and invoke the Interpret operation
- each NonTerminalExpression node defines Interpret in terms of Interpret on each subexpression
- the Interpret operation of each TerminalExpression defines the base case in the recursion
- the Interpret operations at each node use the context to store and access the state of the interpreter

Implementation issues

- the Interpreter pattern does not explain how to create an abstract syntax tree
 - does not address parsing
 - will need to be created by a table-driven parser, by a hand-crafted parser, or directly by the client
- you do not have to define the Interpret operation in the expression classes
- grammars whose sentences contain many occurrences of a terminal symbol might benefit from sharing a single copy of that symbol by using the visitor pattern
- terminal nodes generally do not store information about their position in the abstract syntax tree

Interpreter Pattern Example in intellij

- to implement the interpreter pattern, we need to create an Interpreter context engine that will do the interpretation work
- then we need to create different Expression implementations that will consume the functionalities provided by the interpreter context
- finally we need to create the client that will take the input from user and decide which Expression to use and then generate output for the user
- lets understand this with an example where the user input will be of two forms –
 - “<Number> in Binary” or
 - “<Number> in Hexadecimal”
- our interpreter client should return it in format
 - “<Number> in Binary= <Number_Binary_String>” and
 - “<Number> in Hexadecimal= <Number_Hexadecimal_String>” respectively

Code – Create the Context

- our first step will be to write the Interpreter context class that will do the actual interpretation
 - contains information that is global to the interpreter

```
public class InterpreterContext {  
  
    public String getBinaryFormat(int i){  
        return Integer.toBinaryString(i);  
    }  
  
    public String getHexadecimalFormat(int i){  
        return Integer.toHexString(i);  
    }  
}
```

Code – Create the AbstractExpression

- now we need to create different types of Expressions that will consume the interpreter context class
- declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree

```
public interface Expression {  
    String interpret(InterpreterContext ic);  
}
```

Code - Create the Terminal expression classes

- we will have two expression implementations, one to convert int to binary and other to convert int to hexadecimal format
- implements an Interpret operation associated with terminal symbols in the grammar
 - the Interpret operation of each TerminalExpression defines the base case in the recursion
- an instance is required for every terminal symbol in a sentence
- the Interpret method at each node use the context to store and access the state of the interpreter

```
public class IntToBinaryExpression implements Expression {  
    private int i;  
  
    public IntToBinaryExpression(int c){  
        this.i=c;  
    }  
    @Override  
    public String interpret(InterpreterContext ic) {  
        return ic.getBinaryFormat(this.i);  
    }  
}
```

Code - Create the Terminal expression classes

- implements an Interpret operation associated with terminal symbols in the grammar
 - the Interpret operation of each TerminalExpression defines the base case in the recursion
- an instance is required for every terminal symbol in a sentence
- the Interpret method at each node use the context to store and access the state of the interpreter

```
public class IntToHexExpression implements Expression {  
    private int i;  
  
    public IntToHexExpression(int c){  
        this.i=c;  
    }  
  
    @Override  
    public String interpret(InterpreterContext ic) {  
        return ic.getHexadecimalFormat(i);  
    }  
}
```

Create the Client

- we can create our client application that will have the logic to parse the user input and pass it to the correct expression
- the client will initialize the context and invoke the Interpret operation and then uses the output to generate the user response
- builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines
 - the abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes

Create the Client

```
public class InterpreterClient {  
    public InterpreterContext ic;  
  
    public InterpreterClient(InterpreterContext i) {  
        this.ic=i;  
    }  
  
    public String interpret(String str)  
    {  
        Expression exp = null;  
  
        // perform the parsing  
        if(str.contains("Hexadecimal")){  
            exp=new IntToHexExpression(Integer.parseInt(str.substring(0,str.indexOf(" "))));  
        }else if(str.contains("Binary")) {  
            exp=new IntToBinaryExpression(Integer.parseInt(str.substring(0,str.indexOf(" "))));  
        }else  
            return str;  
  
        return exp.interpret(ic);  
    }  
}
```

Create the Client

```
public class InterpreterClient {  
    public static void main(String args[]){  
        String str1 = "28 in Binary";  
        String str2 = "28 in Hexadecimal";  
  
        InterpreterClient ec = new InterpreterClient(new InterpreterContext());  
        System.out.println(str1+"= "+ec.interpret(str1));  
        System.out.println(str2+"= "+ec.interpret(str2));  
    }  
}
```

Output

28 in Binary= 11100

28 in Hexadecimal= 1c

Demonstration

Jason Fedin

Code – Create the Context

- our first step will be to write the Interpreter context class that will do the actual interpretation
 - contains information that is global to the interpreter

```
public class Context {  
    String input;  
  
    public Context(String input) {  
        this.input = input;  
    }  
  
    public boolean getResult(String data){  
        if(input.contains(data)) {  
            return true;  
        }  
        else {  
            return false;  
        }  
    }  
}
```

Step 1 - Create an expression interface (Expression.java)

- now we need to create different types of Expressions that will consume the interpreter context class
- declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree

```
public interface Expression {  
    public boolean interpret(Context context);  
}
```

Step 2 - Create the Terminal concrete class

- implements an Interpret operation associated with terminal symbols in the grammar
 - the Interpret operation of each TerminalExpression defines the base case in the recursion
- an instance is required for every terminal symbol in a sentence
- the Interpret method at each node use the context to store and access the state of the interpreter

```
public class TerminalExpression implements Expression {
```

```
    private String data;
```

```
    public TerminalExpression(String data){
```

```
        this.data = data;
    }
```

```
    @Override
```

```
    public boolean interpret(Context context) {
```

```
        return context.getResult(data);
```

```
}
```

```
}
```

Create the non-terminal expression classes

- one such class is required for every rule $R ::= R_1 R_2 \dots R_n$ in the grammar
- maintains instance variables of type AbstractExpression for each of the symbols R_1 through R_n
- implements an Interpret operation for nonterminal symbols in the grammar
- each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression
- the Interpret method at each node use the context to store and access the state of the interpreter

```
public class OrExpression implements Expression {  
  
    private Expression expr1 = null;  
    private Expression expr2 = null;  
  
    public OrExpression(Expression expr1, Expression expr2) {  
        this.expr1 = expr1;  
        this.expr2 = expr2;  
    }  
  
    @Override  
    public boolean interpret(Context context) {  
        return expr1.interpret(context) || expr2.interpret(context);  
    }  
}
```

Create the non-terminal expression classes

```
public class AndExpression implements Expression {  
  
    private Expression expr1 = null;  
    private Expression expr2 = null;  
  
    public AndExpression(Expression expr1, Expression expr2) {  
        this.expr1 = expr1;  
        this.expr2 = expr2;  
    }  
  
    @Override  
    public boolean interpret(Context context) {  
        return expr1.interpret(context) && expr2.interpret(context);  
    }  
}
```

Create the client application

- builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines
 - the abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes
- the grammars for my language (using the examples on previous slides) would be
 - AND_EXPR -> exp AND exp
 - OR_EXPR -> exp OR exp
 - exp -> "Robert", "John", "Julie", "Married"

Create the client application

```
public class InterpreterPatternDemo {  
  
    public static Expression getMaleExpression(){  
        Expression robert = new TerminalExpression("Robert");  
        Expression john = new TerminalExpression("John");  
        return new OrExpression(robert, john);  
    }  
  
    public static Expression getMarriedWomanExpression(){  
        Expression julie = new TerminalExpression("Julie");  
        Expression married = new TerminalExpression("Married");  
        return new AndExpression(julie, married);  
    }  
}
```

Create the client application

```
public static void main(String[] args) {  
    Expression isMale = getMaleExpression();  
    Expression isMarriedWoman = getMarriedWomanExpression();  
  
    Context ic = new Context("John");  
    System.out.println("John is male? " + isMale.interpret(ic));  
  
    Context ic2 = new Context("Married Julie");  
    System.out.println("Julie is a married women? " + isMarriedWoman.interpret(ic2));  
}  
}
```

Output

John is male? true

Julie is a married women? true

The Iterator Design Pattern

Jason Fedin

Overview

- the iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- an aggregate object is an object that contains other objects for the purpose of grouping those objects as a unit
 - also called a container or a collection
 - examples are a linked list and a hash table
- iterators are generally used to traverse a container to access its elements
- it is a very commonly used design pattern in Java with the collection framework
 - used to access the elements of a collection object
- the pattern hides the actual implementation of traversal through the collection
 - client programs just use iterator methods

Examples

- suppose there are two companies, company A and company B
- company A stores its employee records (name, etc.) in a linked list
- company B stores its employee data in a big array
- one day the two companies decide to work together
 - the iterator pattern will allow us to have a common interface through which we can access data for both companies
 - we will simply call the same methods without rewriting any code
- another example would be in a college
 - the arts department may use an array data structure
 - the science department may use a linked list data structure to store their students' records
- the main administrative department will access those data through common methods using the iterator
 - it does not care which data structure is used by individual departments

More Overview

- as mentioned, this pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers
- an iterator object is responsible for keeping track of the current element
 - it knows which elements have been traversed already
- once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with any of these aggregates
- the iterator allows different traversal methods (forwards and backwards)
- allows multiple traversals to be in progress concurrently
- places the task of traversal on the iterator object, not on the aggregate
 - simplifies the aggregate interface and implementation
 - places the responsibility where it should be
 - keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration

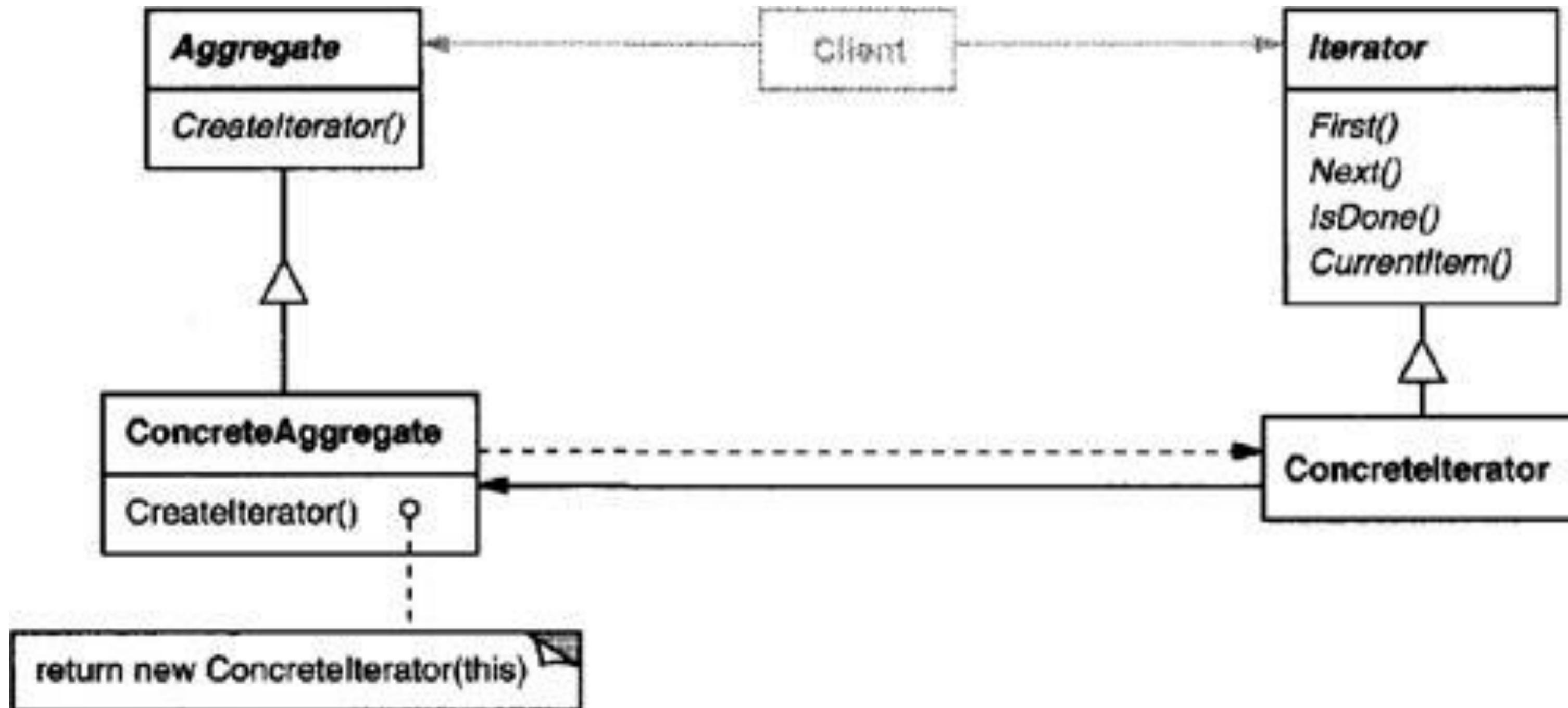
When to use the iterator pattern?

- when you want to provide a standard way to iterate over a collection and hide the implementation logic from client program
 - logic for iteration is embedded in the collection itself and it helps client program to iterate over them easily
- use the pattern to support multiple traversals of aggregate objects
- use the pattern to support polymorphic iteration

Implementing the Iterator

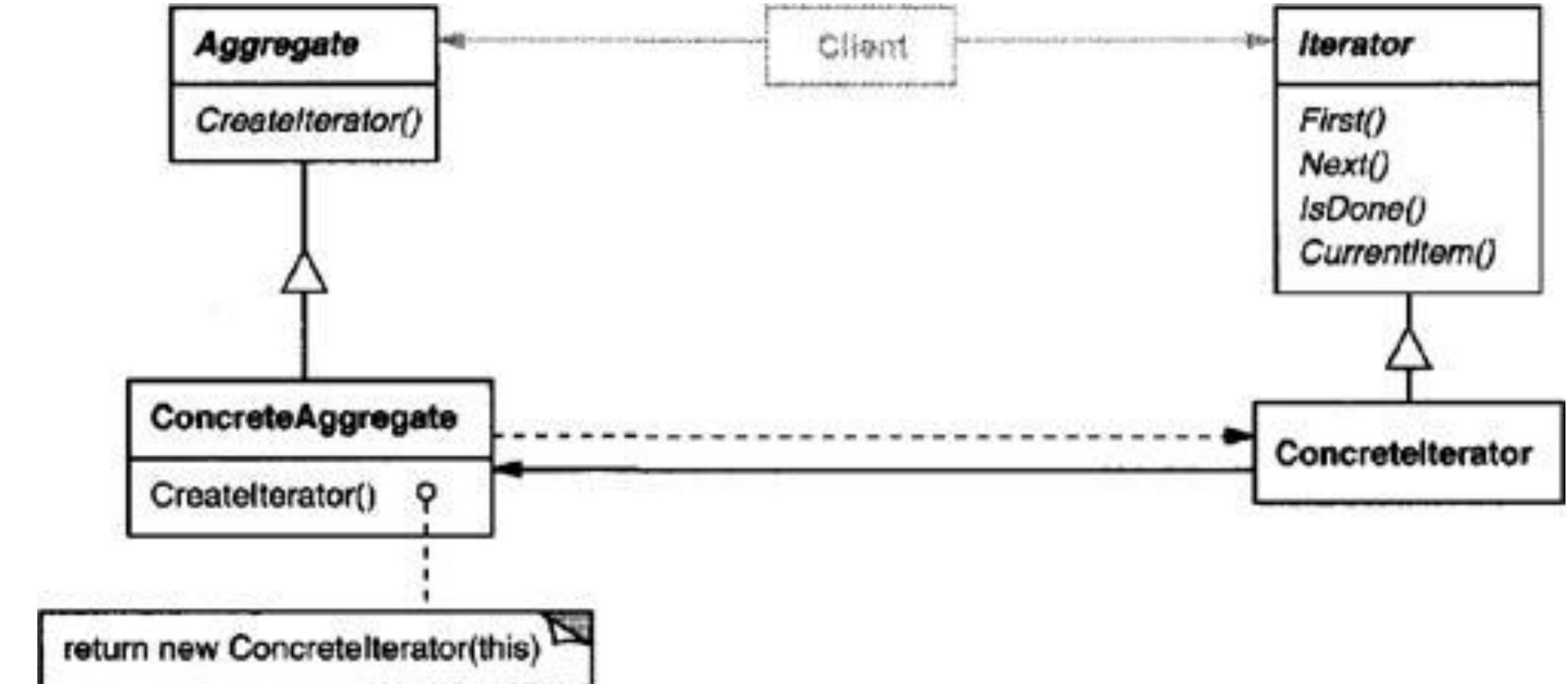
Jason Fedin

Class Diagram



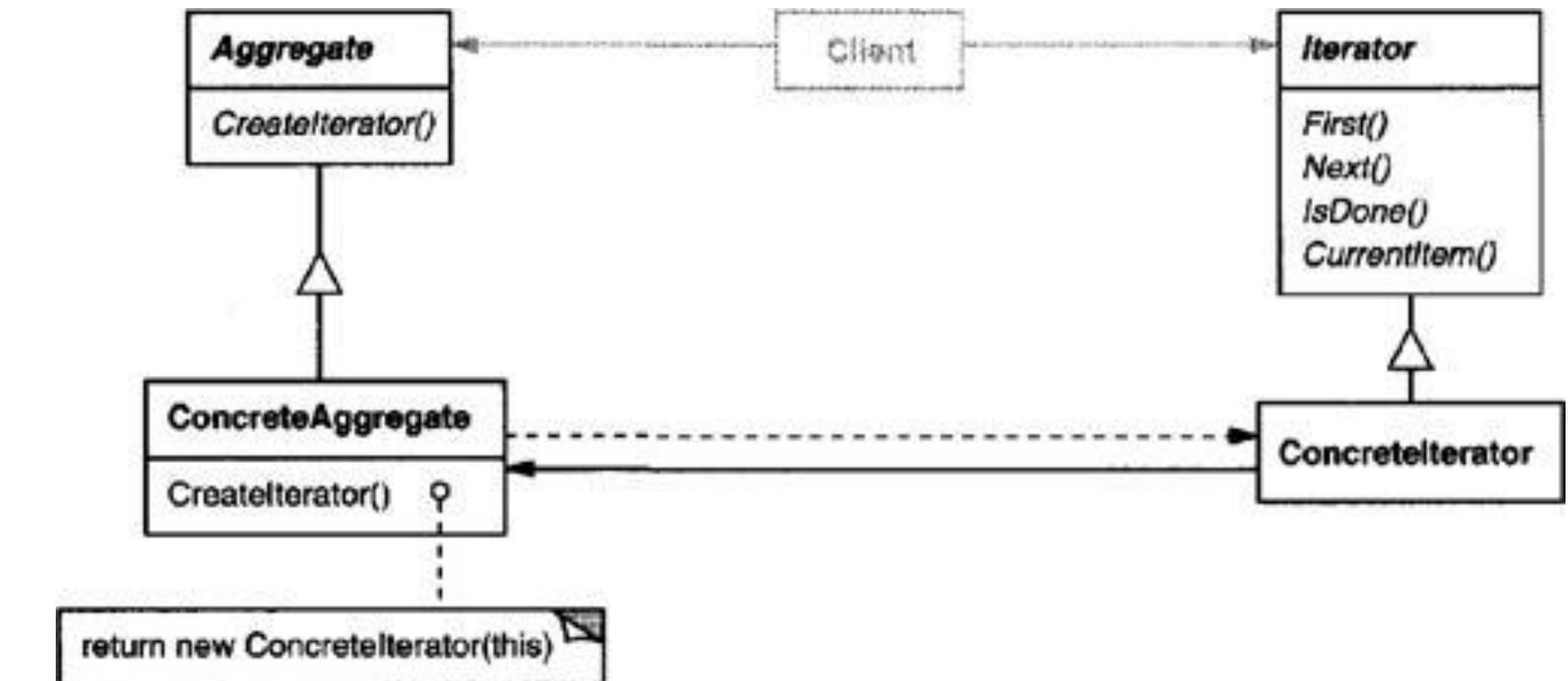
Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Participants



- **Iterator**
 - defines an interface for accessing and traversing elements
- **Concreterator**
 - implements the **Iterator** interface
 - keeps track of the current position in the traversal of the aggregate
- **Aggregate**
 - defines an interface for creating an **Iterator** object

Participants



- **ConcreteAggregate**
 - implements the Iterator creation interface to return an instance of the proper **ConcretIterator**
- a **ConcretIterator** keeps track of the current object in the aggregate and can compute the succeeding object in the traversal

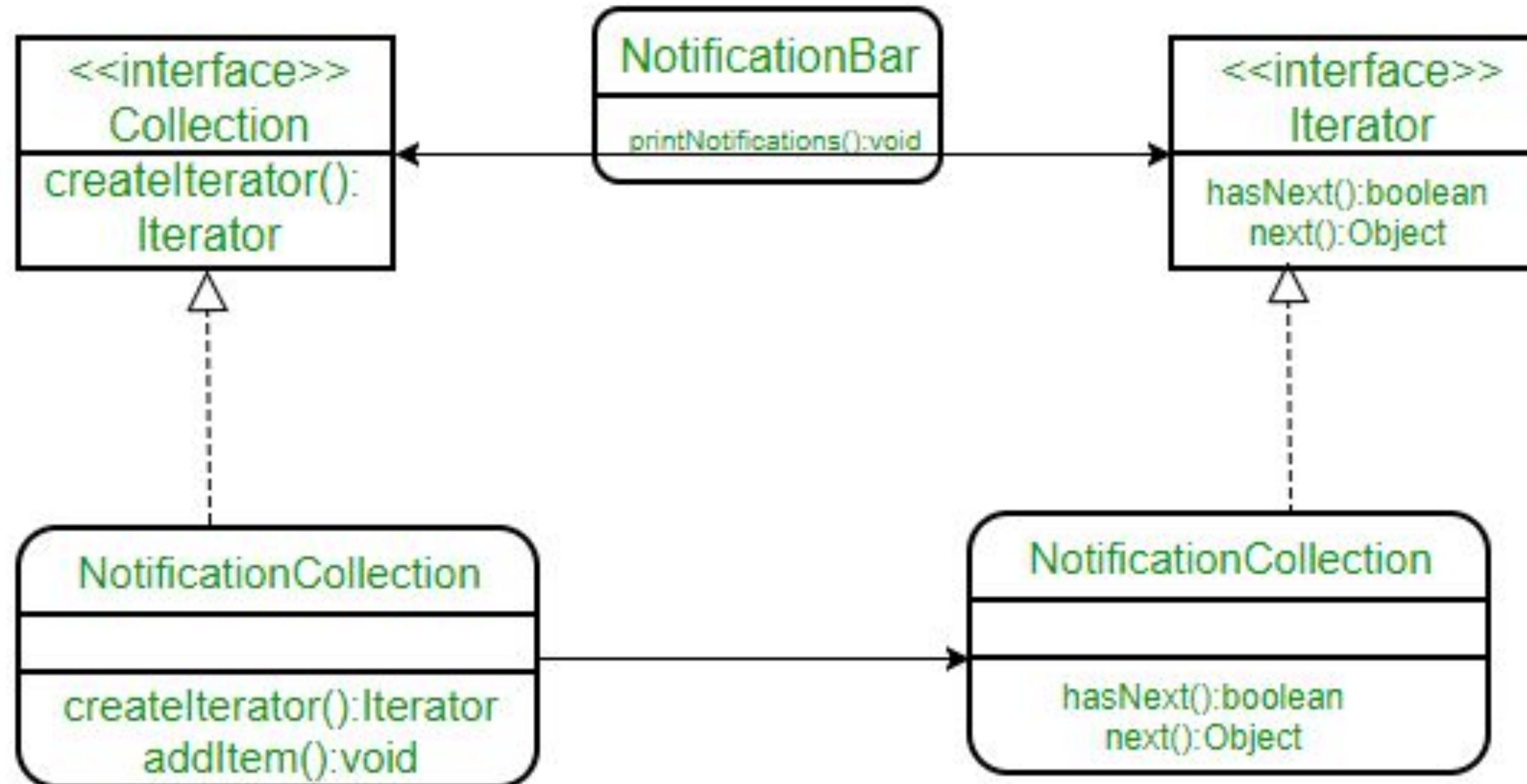
Implementation consequences

- the implementation supports variations in the traversal of an aggregate
 - may traverse the parse tree inorder or preorder
- Iterators make it easy to change the traversal algorithm
 - just replace the iterator instance with a different one
 - you can also define Iterator subclasses to support new traversals
- more than one traversal can be pending on an aggregate
 - an iterator keeps track of its own traversal state
 - you can have more than one traversal in progress at once

Example in intellij

- suppose we are creating a notification bar in our application that displays all the notifications
 - held in a notification collection
- we can provide a collection of notifications to the client and let them write the logic to traverse through the notifications and decide whether to process them
 - this solution has lots of issues such as client has to come up with the logic for traversal
 - we cannot ensure that the client logic is correct
 - if the number of clients grow, then it will become very hard to maintain
- so, we can use the Iterator pattern and provide an iteration
- NotificationCollection provides an iterator to iterate over its elements without exposing how it has implemented the collection (array in this case) to the Client (NotificationBar)
 - the client program can only access the list of notifications through the iterator
- we will first define the contract for our collection and iterator interfaces

Class Diagram



Create a data class to store notifications

```
class Notification  
{  
    // To store notification message  
    String notification;  
  
    public Notification(String notification) {  
        this.notification = notification;  
    }  
  
    public String getNotification() {  
        return notification;  
    }  
}
```

Create the Aggregate interface

- defines an interface for creating an Iterator object

```
interface Collection {  
    public Iterator createIterator();  
}
```

Create the Iterator Interface

- defines an interface for accessing and traversing elements

```
interface Iterator {  
    // indicates whether there are more elements to iterate over  
    boolean hasNext();  
  
    // returns the next element  
    Object next();  
}
```

Create the ConcreteAggregate

- implements the Iterator creation interface to return an instance of the proper ConcreterIterator

```
// Collection of notifications
class NotificationCollection implements Collection {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    Notification[] notificationList;

    public NotificationCollection() {
        notificationList = new Notification[MAX_ITEMS];

        // Let us add some dummy notifications
        addItem("Notification 1");
        addItem("Notification 2");
        addItem("Notification 3");
    }
}
```

Create the ConcreteAggregate

```
public void addItem(String str) {  
    Notification notification = new Notification(str);  
    if (numberOfItems >= MAX_ITEMS)  
        System.err.println("Full");  
    else {  
        notificationList[numberOfItems] = notification;  
        numberOfItems = numberOfItems + 1;  
    }  
}
```

```
public Iterator createIterator() {  
    return new NotificationIterator(notificationList);  
}  
}
```

Create the Concretelterator

- implements the Iterator interface
- keeps track of the current position in the traversal of the aggregate

```
// Notification iterator
class NotificationIterator implements Iterator {
    Notification[] notificationList;

    // maintains curr pos of iterator over the array
    int pos = 0;

    // Constructor takes the array of notificationList are going to iterate over
    public NotificationIterator (Notification[] notificationList) {
        this.notificationList = notificationList;
    }
```

Create the Concretelterator

```
public Object next() {  
    // return next element in the array and increment pos  
    Notification notification = notificationList[pos];  
    pos += 1;  
    return notification;  
}  
  
public boolean hasNext() {  
    if (pos >= notificationList.length || notificationList[pos] == null)  
        return false;  
    else  
        return true;  
}  
}
```

Create a helper class to store collections

```
// Contains collection of notifications as an object of NotificationCollection
class NotificationBar {
    NotificationCollection notifications;

    public NotificationBar(NotificationCollection notifications) {
        this.notifications = notifications;
    }

    public void printNotifications() {
        Iterator iterator = notifications.createIterator();
        System.out.println("-----NOTIFICATION BAR-----");
        while (iterator.hasNext()) {
            Notification n = (Notification)iterator.next();
            System.out.println(n.getNotification());
        }
    }
}
```

Create the client

```
class Main
{
    public static void main(String args[])
    {
        NotificationCollection nc = new NotificationCollection();
        NotificationBar nb = new NotificationBar(nc);
        nb.printNotifications();
    }
}
```

Output

-----NOTIFICATION BAR-----

Notification 1

Notification 2

Notification 3

- notice that if we would have used an ArrayList instead of Array there will not be any change in the client (notification bar) code due to the decoupling achieved by the use of iterator interface

Demonstration

Jason Fedin

Create the Aggregate Interface

```
import iterator.*;  
  
public interface ISubject  
{  
    public IIterator CreateIterator();  
}
```

Create the Iterator interface

```
public interface iterator {  
  
    void First();//Reset to first element  
    String Next();//get next element  
    Boolean IsDone();//End of collection check  
    String CurrentItem();//Retrieve Current Item  
  
}
```

Create each concrete aggregate

```
import iterator.*;

public class Arts implements Isubject {
    private String[] subjects;

    public Arts() {
        subjects = new String[2];
        subjects[0] = "Bengali";
        subjects[1] = "English" ;
    }

    public IIterator CreateIterator() {
        return new ArtsIterator(subjects);
    }
}
```

Create each concrete aggregate

```
//Containing the ArtsIterator
public class ArtsIterator implements Iterator {
    private String[] subjects;
    private int position;

    public ArtsIterator(String[] subjects) {
        this.subjects = subjects;
        position = 0;
    }

    public void First() {
        position = 0;
    }
}
```

JDP-2180-1

Create each concrete aggregate

```
public String Next() {  
    return subjects[position++];  
}
```

```
public Boolean IsDone() {  
    return position >= subjects.length;  
}
```

```
public String CurrentItem() {  
    return subjects[position];  
}  
}
```

Create each concrete aggregate

```
import java.util.LinkedList;  
import iterator.*;  
  
public class Science implements Isubject {  
    private LinkedList<String> subjects;  
  
    public Science() {  
        subjects = new LinkedList<String>();  
        subjects.addLast("Maths");  
        subjects.addLast("Comp. Sc.");  
        subjects.addLast("Physics");  
    }  
  
    @Override  
    public Iterator CreateIterator() {  
        return new ScienceIterator(subjects);  
    }  
}
```

Create each concrete aggregate

```
//Containing the Sciencelterator
public class Sciencelterator implements iterator {
    private LinkedList<String> subjects;
    private int position;

    public Sciencelterator(LinkedList<String> subjects) {
        this.subjects = subjects;
        position = 0;
    }

    public void First() {
        position = 0;
    }
```

Create each concrete aggregate

```
public String Next() {  
    return subjects.get(position++);  
}
```

```
public Boolean IsDone() {  
    return position >= subjects.size();  
}
```

```
public String CurrentItem() {  
    return subjects.get(position);  
}  
}
```

Create the client

```
import iterator.*;
import aggregate.*;

class IteratorPatternEx {
    public static void main(String[] args) {
        System.out.println("***Iterator Pattern Demo***\n");
        ISubject Sc_subject = new Science();
        ISubject Ar_subjects = new Arts();

        IIterator Sc_iterator = Sc_subject.CreateIterator();
        IIterator Ar_iterator = Ar_subjects.CreateIterator();

        System.out.println("\nScience subjects :");
        Print(Sc_iterator);

        System.out.println("\nArts subjects :");
        Print(Ar_iterator);
    }
}
```

Create the client

```
public static void Print(IIterator iterator) {  
    while (!iterator.IsDone()) {  
        System.out.println(iterator.Next());  
    }  
}
```

Output

```
Console X
<terminated> IteratorPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 14, 2015, 12:15:08 PM)
***Iterator Pattern Demo***

Science subjects :
Maths
Comp. Sc.
Physics

Arts subjects :
Bengali
English
```

The Mediator Design Pattern

Jason Fedin

Overview

- the mediator design pattern defines an object that encapsulates how a set of objects interact
 - promotes loose coupling by keeping objects from referring to each other explicitly
 - we can reduce the direct interconnections among the objects
 - lets you vary their interaction independently
- a mediator is the one who takes the responsibility of communication among a group of objects
 - acts as an intermediary who can track the communication between two objects
 - the other objects in the system are also aware of this mediator
 - they know that if they need to communicate among themselves, they need to go through the mediator
- used to reduce communication complexity between multiple objects or classes

Examples

- an air traffic controller is a great example
 - the airport control room works as a mediator for communication between different flights
 - mediator works as a router between objects
 - has its own logic to provide way of communication
- in an airplane application, before taking off the flight undergoes a series of checks
 - checks confirm that all components/parts (which are dependent on each other) are in perfect condition
- you could use a mediator for a house of the future where appliances talk to each other
 - when a user stops hitting the snooze button, the alarm clock tells the coffee maker to start brewing
 - turn off the sprinkler 15 minutes before a shower is scheduled
 - set the alarm early on trash days
- commonly used to coordinate related GUI components

JDK examples

- java.util.Timer class scheduleXXX() methods
- Java Concurrency Executor execute() method
- java.lang.reflect.Method invoke() method
- Java Message Service (JMS) uses Mediator pattern along with Observer pattern to allow applications to subscribe and publish data to other applications

What problems the mediator can solve?

- object-oriented design encourages the distribution of behavior among objects
- such distribution can result in an object structure with many connections between objects
 - in the worst case, every object ends up knowing about every other
- lots of interconnections make it less likely that an object can work without the support of others
 - it can be difficult to change the system's behavior in any significant way
 - since behavior is distributed among many objects
- if we need to make some kind of change, it becomes a challenging task
- the mediator pattern will help solve the above problems

Advantages

- increases the reusability of the objects supported by the mediator by decoupling them from the system
- simplifies maintenance of the system by centralizing control logic
- simplifies and reduces the variety of messages sent between objects in the system
- allows you to replace one object in the structure with a different one without affecting the classes and the interfaces
- we should not use mediator pattern just to achieve lose-coupling
 - if the number of mediators grows, it then becomes hard to maintain them

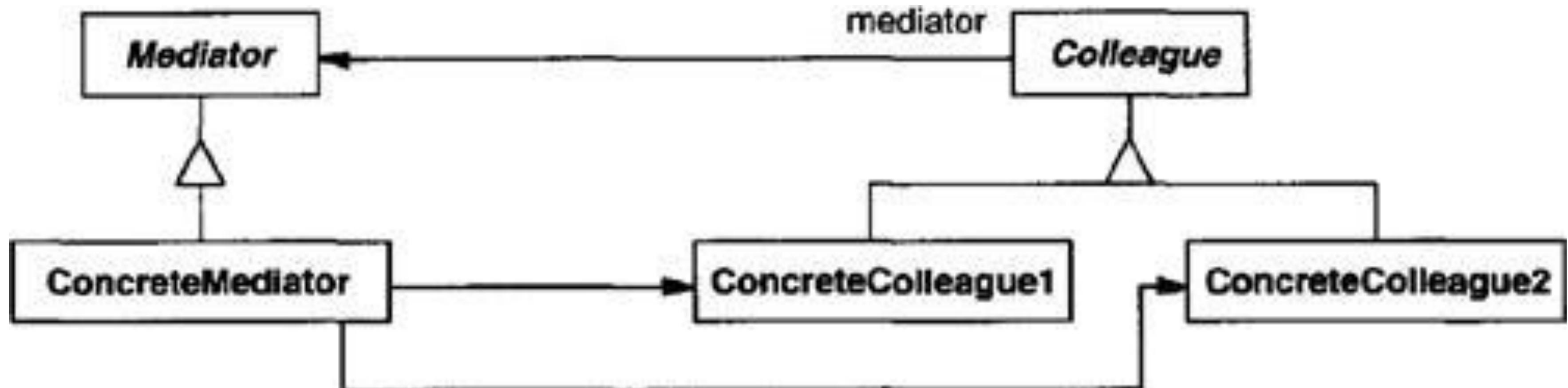
When to use the Mediator?

- use the mediator pattern to centralize complex communications and control between related objects
- when a set of objects communicate in well-defined but complex ways
 - the resulting interdependencies are unstructured and difficult to understand
- when reusing an object is difficult because it refers to and communicates with many other objects
- when a behavior that is distributed between several classes should be customizable without a lot of sub-classing

Implementing the Mediator

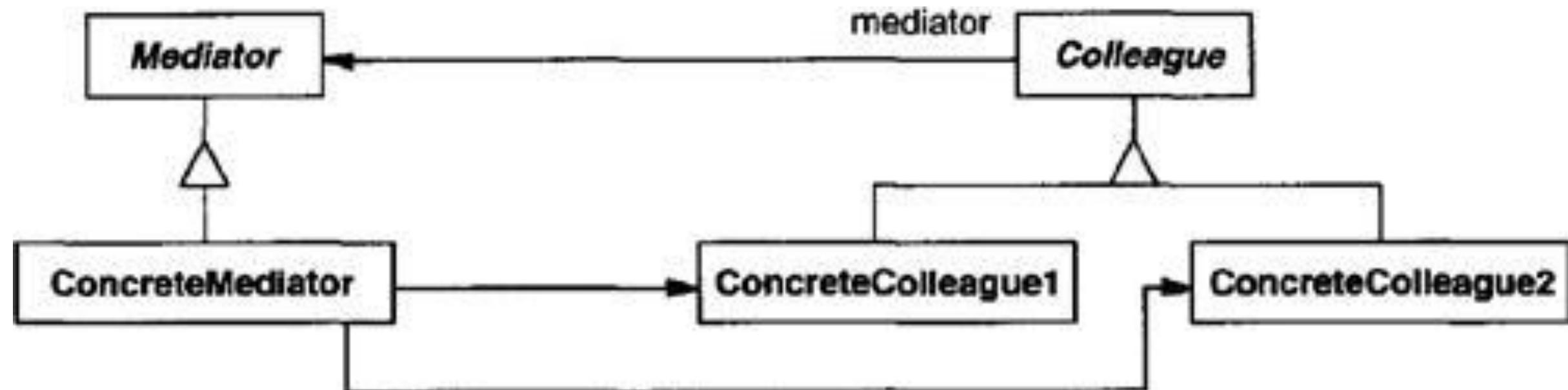
Jason Fedin

Class Diagram



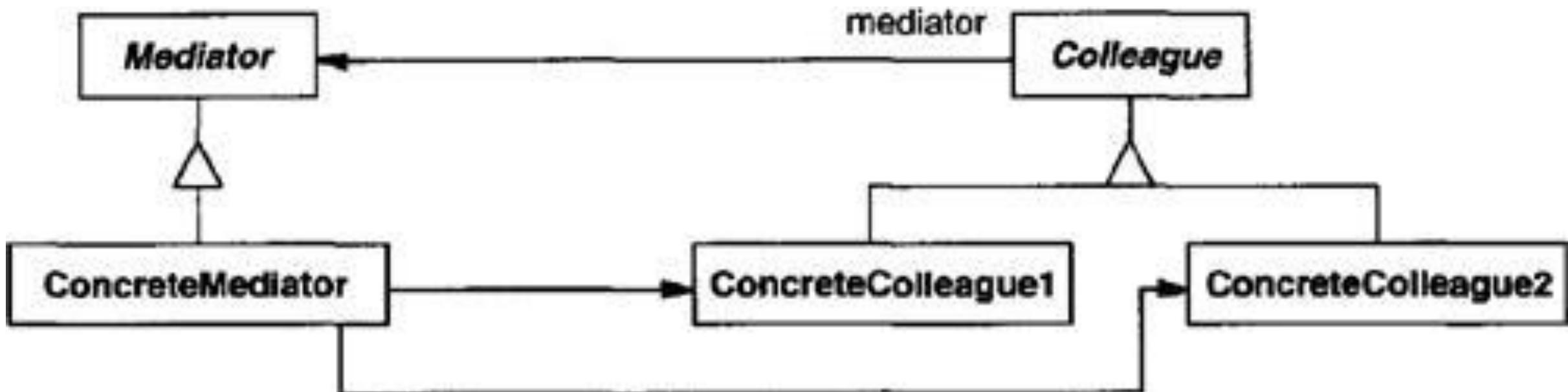
Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Participants



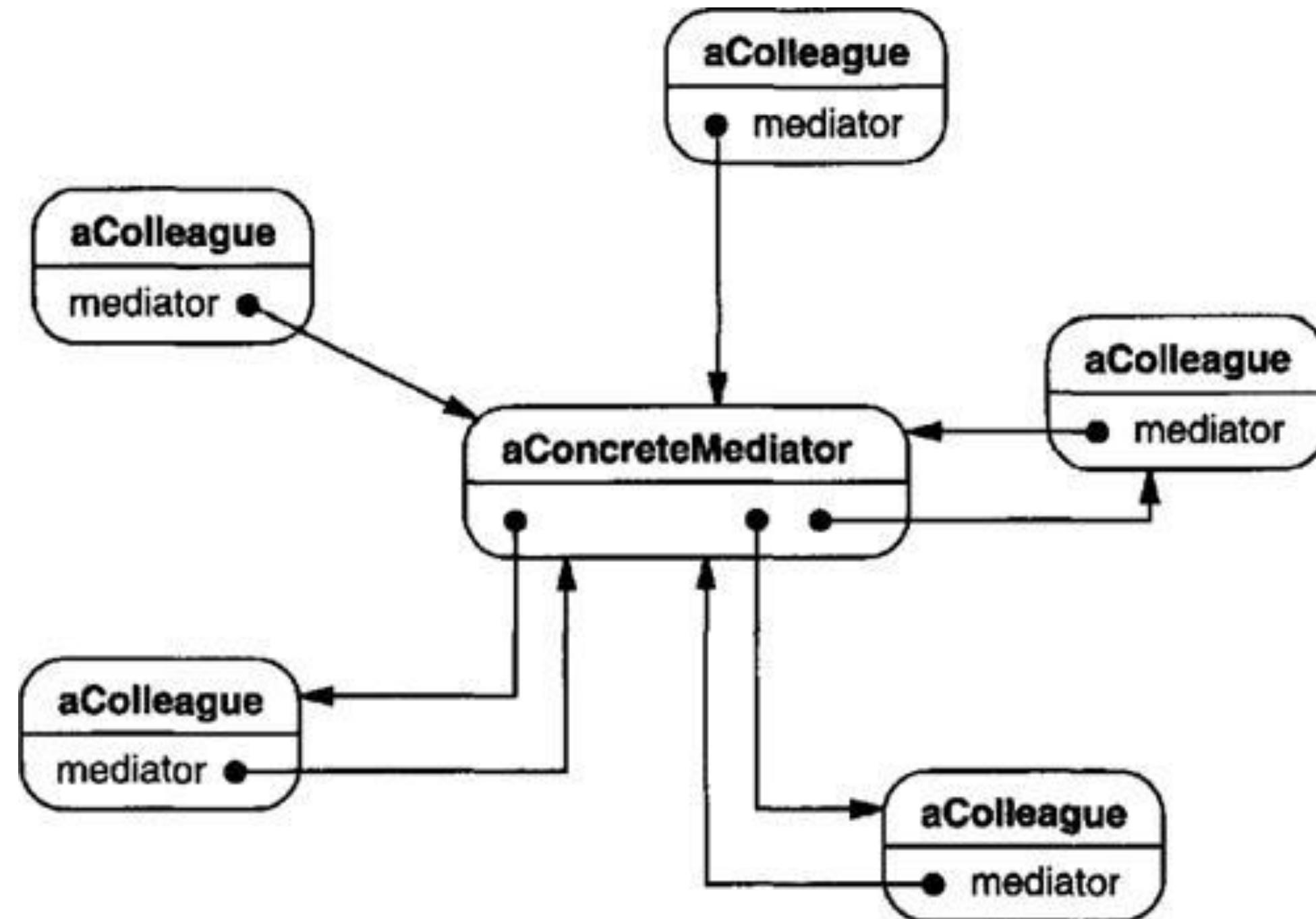
- Mediator
 - defines an interface for communicating with Colleague objects
- ConcreteMediator
 - implements cooperative behavior by coordinating Colleague objects
 - knows and maintains its colleagues

Participants



- Colleague classes
 - each Colleague class knows its Mediator object
 - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague
- Colleagues send and receive requests from a Mediator object
 - mediator implements the cooperative behavior by routing requests between the appropriate colleague(s)
- the system objects that communicate with each other are called Colleagues
 - usually we have an interface or abstract class that provides the contract for communication and then we have concrete implementation of mediators

Object Structure



Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Implementation issues

- no need to define an abstract Mediator class when colleagues work with only one mediator
 - the abstract coupling that the Mediator class provides lets colleagues work with different Mediator subclasses, and vice versa
- colleagues have to communicate with their mediator when an event of interest occurs.
 - one approach is to implement the Mediator as an Observer
 - Colleague classes act as Subjects
 - send notifications to the mediator whenever they change state
 - mediator responds by propagating the effects of the change to other colleagues

Implementation advantages

- the implementation limits sub-classing
 - a mediator localizes behavior that otherwise would be distributed among several objects
 - changing this behavior requires subclassing Mediator only
 - colleague classes can be reused as is
- it decouples colleagues
 - a mediator promotes loose coupling between colleagues
 - you can vary and reuse Colleague and Mediator classes independently

Implementation advantages (cont'd)

- it simplifies object protocols
 - replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues
 - one-to-many relationships are easier to understand, maintain, and extend
- it abstracts how objects cooperate
 - making mediation an independent concept and encapsulating it in an object lets you focus on how objects interact apart from their individual behavior
 - can help clarify how objects interact in a system
- It centralizes control
 - trades complexity of interaction for complexity in the mediator
 - a mediator can become more complex than any individual colleague
 - can make the mediator itself a monolith that is hard to maintain

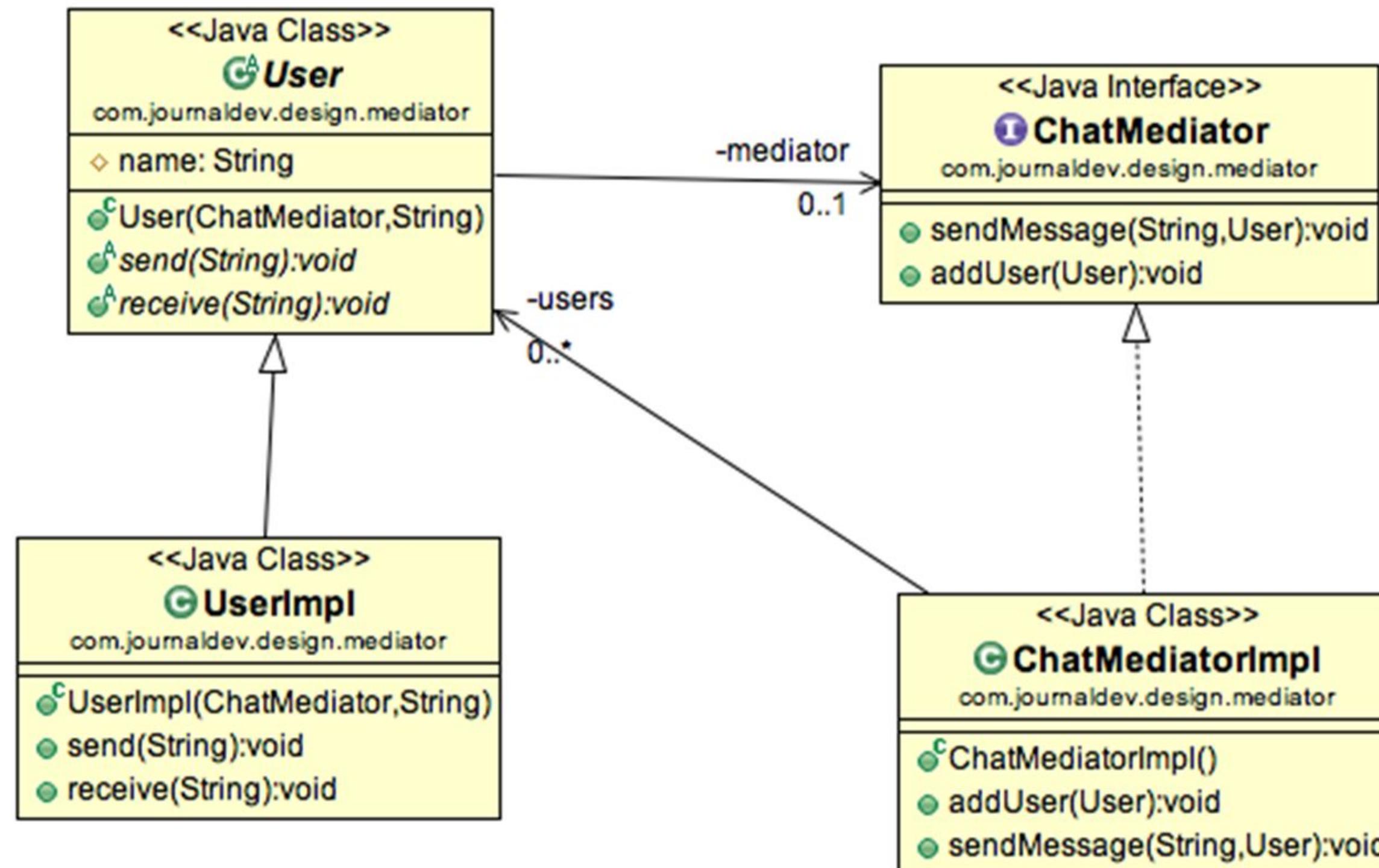
Implementing the Mediator

Jason Fedin

Example in intellij

- lets create a chat application
 - users can do a group chat
- every user will be identified by their name and they can send and receive messages
- messages sent by any user should be received by all the other users in the group
- the first step is to create the Mediator interface that will define the contract for concrete mediators

Class Diagram



Create the Mediator interface

- defines an interface for communicating with Colleague objects

```
public interface ChatMediator {  
  
    public void sendMessage(String msg, User user);  
  
    void addUser(User user);  
}
```

Create the Colleague Interface

- users can send and receive messages
- a User has a reference to the mediator object, it's required for the communication between different users

```
public abstract class User {  
    protected ChatMediator mediator;  
    protected String name;  
  
    public User(ChatMediator med, String name){  
        this.mediator=med;  
        this.name=name;  
    }  
  
    public abstract void send(String msg);  
  
    public abstract void receive(String msg);  
}
```

Create the ConcreteMediator

- we will create concrete mediator class, it will have a list of users in the group and provide logic for the communication between the users
- implements cooperative behavior by coordinating Colleague objects
- knows and maintains its colleagues

```
import java.util.ArrayList;  
import java.util.List;  
  
public class ChatMediatorImpl implements ChatMediator {  
  
    private List<User> users;  
  
    public ChatMediatorImpl(){  
        this.users=new ArrayList<>();  
    }  
}
```

Create the ConcreteMediator

- we will create concrete mediator class, it will have a list of users in the group and provide logic for the communication between the users

```
@Override  
public void addUser(User user){  
    this.users.add(user);  
}
```

```
@Override  
public void sendMessage(String msg, User user) {  
    for(User u : this.users){  
        //message should not be received by the user sending it  
        if(u != user){  
            u.receive(msg);  
        }  
    }  
}
```

Create the concreteColleague classes

- we can create concrete User classes to be used by client system
 - each Colleague class knows its Mediator object
 - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague
-
- Colleagues send and receive requests from a Mediator object
 - mediator implements the cooperative behavior by routing requests between the appropriate colleague(s)
-
- the system objects that communicate each with other are called Colleagues
 - usually we have an interface or abstract class that provides the contract for communication and then we have concrete implementation of mediators

```
public class UserImpl extends User {  
  
    public UserImpl(ChatMediator med, String name) {  
        super(med, name);  
    }  
}
```

Create the concreteColleague classes

```
@Override  
public void send(String msg){  
    System.out.println(this.name+": Sending Message="+msg);  
    mediator.sendMessage(msg, this);  
}  
  
@Override  
public void receive(String msg) {  
    System.out.println(this.name+": Received Message:"+msg);  
}  
  
}
```

- the send() method is using mediator to send the message to the users and it has no idea how it will be handled by the mediator

Create the client

- Let's test this our chat application with a simple program where we will create mediator and add users to the group and one of the user will send a message
- notice that client program is very simple and it has no idea how the message is getting handled and if mediator is getting user or not

```
public class ChatClient {  
  
    public static void main(String[] args) {  
        ChatMediator mediator = new ChatMediatorImpl();  
        User user1 = new UserImpl(mediator, "Pankaj");  
        User user2 = new UserImpl(mediator, "Lisa");  
        User user3 = new UserImpl(mediator, "Saurabh");  
        User user4 = new UserImpl(mediator, "David");  
        mediator.addUser(user1);  
        mediator.addUser(user2);  
        mediator.addUser(user3);  
        mediator.addUser(user4);  
        user1.send("Hi All");  
    }  
}
```

Output

Pankaj: Sending Message=Hi All

Lisa: Received Message:Hi All

Saurabh: Received Message:Hi All

David: Received Message:Hi All

Demonstration

Jason Fedin

Create the Mediator interface

```
public interface Mediator {  
  
    // The mediator interface  
    public void addBuyer(Buyer buyer);  
    public void findHighestBidder();  
}
```

Create the concreteMediator

```
public class AuctionMediator implements Mediator {  
  
    // this class implements the interface and holds all the buyers in a Array list. We can add buyers and find the highest bidder  
    private ArrayList buyers;  
  
    public AuctionMediator()  {  
        buyers = new ArrayList<>();  
    }  
  
    @Override  
    public void addBuyer(Buyer buyer) {  
        buyers.add(buyer);  
        System.out.println(buyer.name + " was added to the buyers list.");  
    }  
}
```

Create the concreteMediator

```
@Override  
public void findHighestBidder() {  
    int maxBid = 0;  
    Buyer winner = null;  
    for (Buyer b : buyers) {  
        if (b.price > maxBid) {  
            maxBid = b.price;  
            winner = b;  
        }  
        b.auctionHasEnded();  
    }  
    System.out.println("The auction winner is " + winner.name + ". He paid " + winner.price + "$ for the item.");  
}
```

Create the Colleague interface

```
public abstract class Buyer {  
  
    // this class holds the buyer  
    protected Mediator mediator;  
    protected String name;  
    protected int price;  
  
    public Buyer(Mediator med, String name) {  
        this.mediator = med;  
        this.name = name;  
    }  
  
    public abstract void bid(int price);  
  
    public abstract void cancelTheBid();  
  
    public abstract void auctionHasEnded();  
}
```

Create the concrete colleagues

```
public class AuctionBuyer extends Buyer {  
  
    // implementation of the bidding process. There is an option to bid and an option to cancel the bidding  
    public AuctionBuyer(Mediator mediator, String name) {  
        super(mediator, name);  
    }  
  
    @Override  
    public void bid(int price) {  
        this.price = price;  
    }  
  
    @Override  
    public void cancelTheBid() {  
        this.price = -1;  
    }  
  
    @Override  
    public abstract void auctionHasEnded() {  
        System.out.println("Received message that the auction is over: " + name);  
    }  
}
```

Create the client

```
public class Main {  
  
    /* This program illustrate an auction. The AuctionMediator is responsible for adding the buyers, and after each buyer bid a certain amount for the item, the mediator know who won the auction. */  
    public static void main(String[] args) {  
  
        AuctionMediator med = new AuctionMediator();  
        Buyer b1 = new AuctionBuyer(med, "Tal Baum");  
        Buyer b2 = new AuctionBuyer(med, "Elad Shamailov");  
        Buyer b3 = new AuctionBuyer(med, "John Smith");  
  
        // Crate and add buyers  
        med.addBuyer(b1);  
        med.addBuyer(b2);  
        med.addBuyer(b3);  
  
        System.out.println("Welcome to the auction. Tonight we are selling a vacation to Vegas. please Bid your offers.");  
        System.out.println("-----");
```

Create the client

```
System.out.println("Waiting for the buyer's offers...");  
  
// Making bids  
b1.bid(1800);  
b2.bid(2000);  
b3.bid(780);  
System.out.println("-----");  
med.findHighestBidder();  
  
b2.cancelTheBid();  
System.out.print(b2.name + " Has canceled his bid!, in that case ");  
med.findHighestBidder();  
}  
}
```

Output

Tal Baum was added to the buyers list.

Elad Shamilov was added to the buyers list.

John Smith was added to the buyers list.

Welcome to the auction. Tonight we are
selling a vacation to Vegas. please Bid your offers.

Waiting for the buyer's offers...

The auction winner is Elad Shamilov.

He paid 2000\$ for the item.

Elad Shamilov Has canceled his bid!, In that
case The auction winner is Tal Baum.

He paid 1800\$ for the item.

Summary

- you should have a clear idea that this pattern is very useful when we observe complex communications in a system
 - Communication (among objects) is much simpler with this pattern
- this pattern reduces the number of subclasses in the system and it also enhances the loose coupling in the system
- here the “many-to-many” relationship is replaced with the “one-to-many” relationship—which is much easier to read and understand
- we can provide a centralized control with this pattern
- sometimes the encapsulation process becomes tricky and we find it difficult to maintain or implement

The Memento Design Pattern

Jason Fedin

Overview

- the memento design pattern will capture and externalize an object's internal state so that the object can be restored to this state later, without violating encapsulation
- the goal is to save the state of an object, so that in the future, we can go back to the specified state
- implemented in such a way that the saved state data of the object is not accessible outside of the object
 - protects the integrity of saved state data
 - does not break encapsulation
- you can use this pattern when you want to be able to return an object to one of its previous states
 - if your user requests an "undo"

Examples

- in notepad we use undo frequently by pressing ctrl+Z
- a classic example includes the state in a finite state machine
- in real-world database programming we often need to roll back a transaction operation
- in a video game, a “save progress” option
 - players can store their game progress and at least recover most of their efforts when their character dies
 - “save progress” function would return a player to the last level he/she completed successfully
- a graphical editor that supports connectivity between objects
 - a user can connect two rectangles with a line
 - the rectangles stay connected when the user moves either of them
 - the editor ensures that the line stretches to maintain the connection

Why the memento?

- sometimes it is necessary to record the internal state of an object
 - required when implementing checkpoints and undo mechanisms
 - let users back out of tentative operations or recover from errors
- you must save state information somewhere so that you can restore objects to their previous states
- objects normally encapsulate some or all of their state
 - making it inaccessible to other objects and impossible to save externally
 - exposing this state would violate encapsulation
 - can compromise the application's reliability and extensibility
- the memento addresses the above issues without violating encapsulation

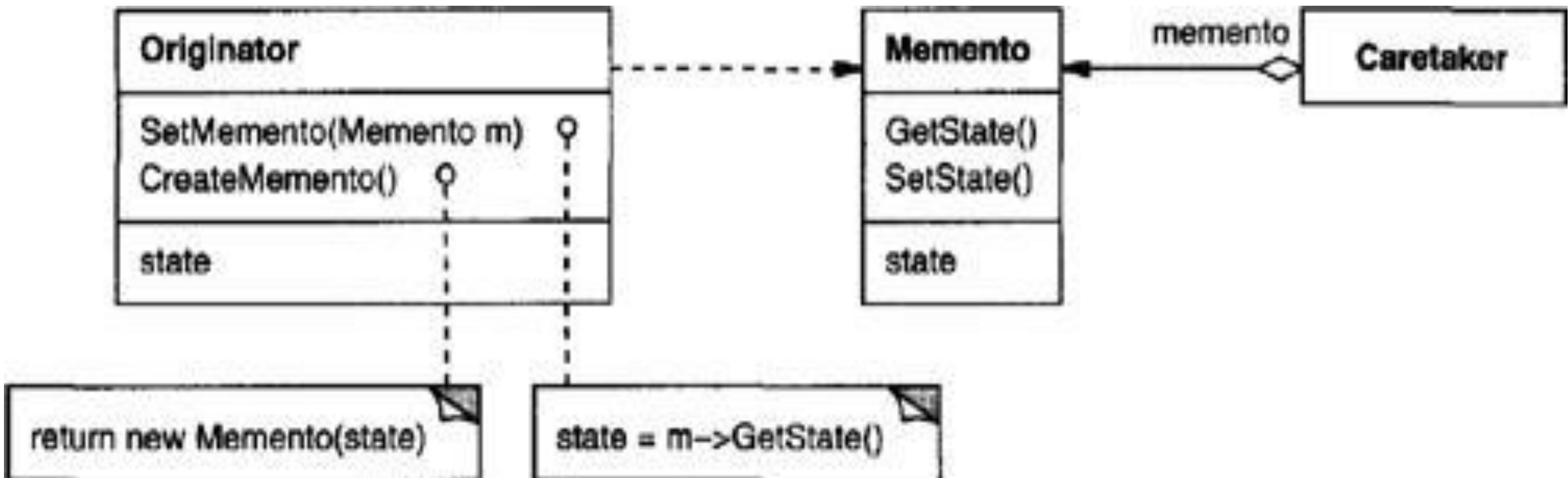
advantages and drawbacks

- simple and easy to implement
- provides easy-to-implement recovery capability
- a drawback to using Memento is that saving and restoring state can be time consuming
 - consider using Serialization to save a system's state

Implementing the Memento

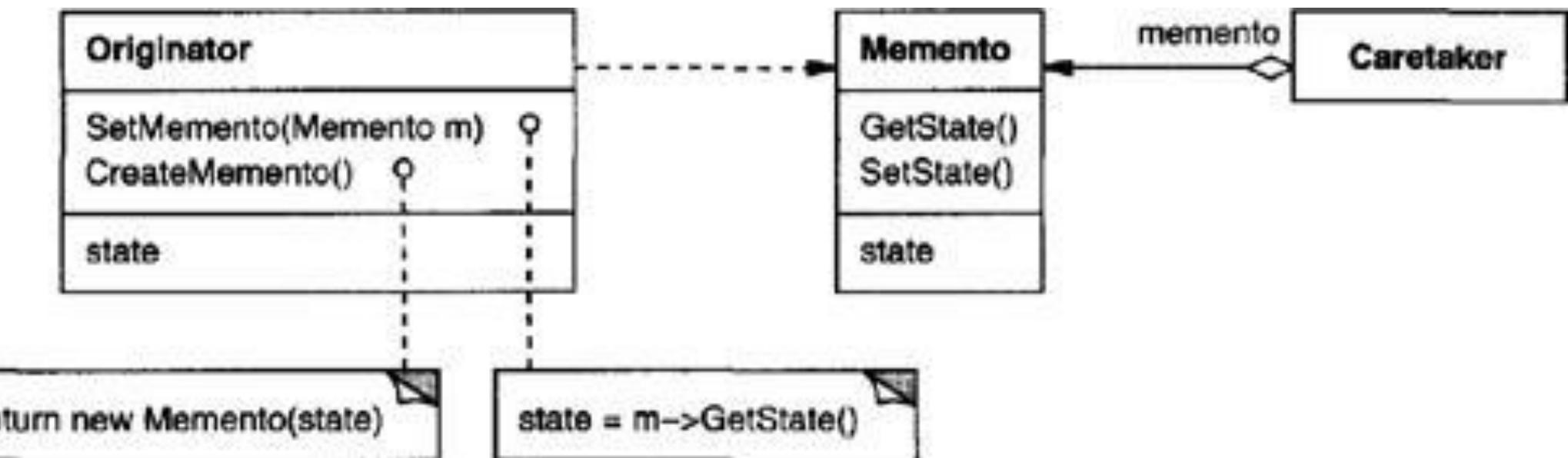
Jason Fedin

Overview



Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

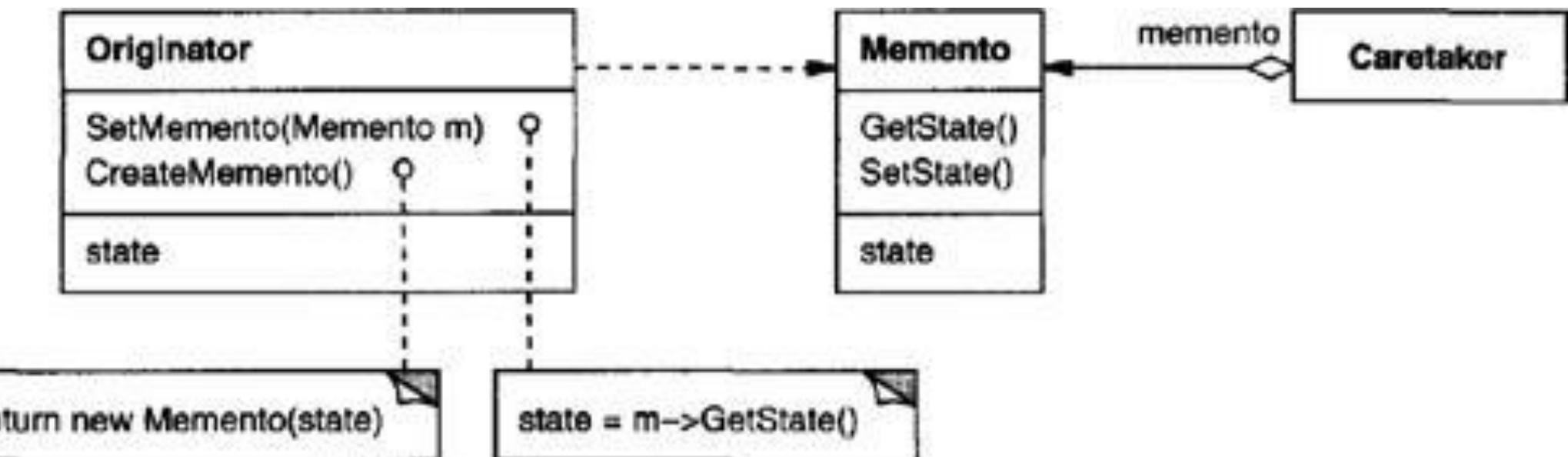
Participants



- **Memento**

- stores the internal state of the Originator object
- may store as much or as little of the originator's internal state as necessary
- protects against access by objects other than the originator
- has effectively two interfaces
 - Caretaker sees a narrow interface to the Memento
 - can only pass the memento to other objects
 - Originator sees a wide interface
 - lets it access all the data necessary to restore itself to its previous state
 - only the originator that produced the memento would be permitted to access the memento's internal state

Participants

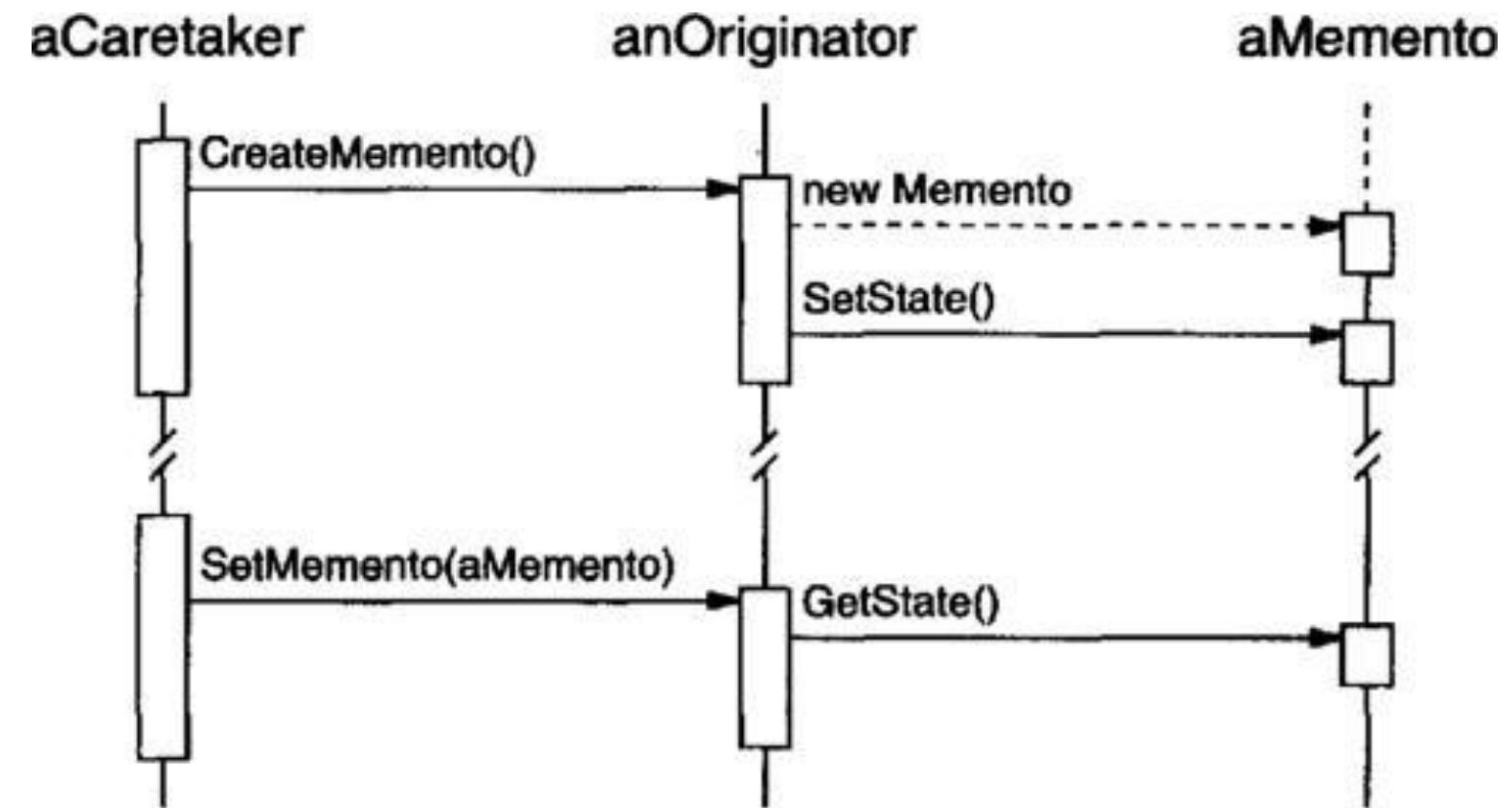


- Originator
 - creates a memento containing a snapshot of its current internal state
 - uses the memento to restore its internal state
- Caretaker
 - responsible for the memento's safekeeping
 - never operates on or examines the contents of a memento
- this design pattern is implemented with the two above objects

Originator and Caretaker details

- Originator is the object whose state needs to be saved and restored
- Caretaker is the helper class that is responsible for storing and restoring the Originator's state through the Memento object
 - keeps track of multiple mementos (maintaining save points)
 - memento is stored as an Object within the caretaker
- a caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator
- sometimes the caretaker will not pass the memento back to the originator
 - the originator may never need to revert to an earlier state
- mementos are passive
 - only the originator that created a memento will assign or retrieve its state

Sequence Diagram



- a Caretaker would like to perform an operation on the Originator while having the possibility to rollback
 - caretaker calls the **createMemento()** method on the originator asking the originator to pass it a memento object
 - the originator creates a memento object saving its internal state and passes the memento to the caretaker
 - the caretaker maintains the memento object and performs the operation
- in case of the need to undo the operation, the caretaker calls the **setMemento()** method on the originator passing the maintained memento object
 - originator would accept the memento, using it to restore its previous state

Implementation consequences

- the memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator
 - shields other objects from potentially complex Originator internals, thereby preserving encapsulation boundaries
- having clients manage the state they ask for simplifies the Originator and keeps clients from having to notify originators when they are done
- mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento
 - unless encapsulating and restoring Originator state is cheap, the pattern might not be appropriate

Summary

- the memento will store the internal states of the originator
- the originator can access the internal states of the memento and it has the ability to restore into its earlier state
 - can also retrieve information from the memento
- the caretaker takes care of the memento's safekeeping or protection and it should not examine the contents of the memento

Implementing the Memento

Jason Fedin

Example in intellij

- one of the best real life examples of needing a memento is a text editor
 - we can save data anytime and use undo to restore it to previous saved state
- lets implement an application that will implement this feature
- this application will provide a utility where we can write and save contents to a File anytime and we can restore it to its last saved state
- for simplicity, I will not use any IO operations to write data into file

Lets first create the Originator class

- Originator is the object whose state needs to be saved and restored
 - uses an inner class (Memento) to save the state of the Object (does not have to use an inner class)
 - private so that it cannot be accessed from other objects
 - creates a memento containing a snapshot of its current internal state
 - uses the memento to restore its internal state

```
public class FileWriterUtil {  
    private String fileName;  
    private StringBuilder content;  
  
    public FileWriterUtil(String file) {  
        this.fileName=file;  
        this.content=new StringBuilder();  
    }  
}
```

Lets first create the Originator class

```
@Override  
public String toString(){  
    return this.content.toString();  
}  
  
public void write(String str){  
    content.append(str);  
}
```

Lets first create the Originator class

```
// creates the memento  
public Memento save(){  
    return new Memento(this.fileName,this.content);  
}  
  
// restore into its earlier state  
public void undoToLastSave(Object obj){  
    Memento memento = (Memento) obj;  
    this.fileName= memento.fileName;  
    this.content=memento.content;  
}
```

Create the memento class (inner class of originator)

- stores the internal state of the Originator object
 - uses an inner class (Memento) to save the state of the Object (does not have to use an inner class, can be public)
 - private so that it cannot be accessed from other objects
 - only the originator that produced the memento would be permitted to access the memento's internal state

```
private class Memento{  
    private String fileName;  
    private StringBuilder content;  
  
    public Memento(String file, StringBuilder content){  
        this.fileName=file;  
        // notice the deep copy so that Memento and FileWriterUtil  
        // content variables don't refer to same object  
        this.content=new StringBuilder(content);  
    }  
}  
}
```

Create the CareTaker

class responsible for the memento's safekeeping

- never operates on or examines the contents of a memento
- Caretaker is the helper class that is responsible for storing and restoring the Originator's state through the Memento object
 - keeps track of multiple mementos (maintaining save points)
 - memento is stored as an Object within the caretaker
- caretaker object contains the saved state in the form of Object, so it can't alter its data and also it has no knowledge of its structure

Create the CareTaker

```
public class FileWriterCaretaker {  
  
    private Object obj;  
  
    public void save(FileWriterUtil fileWriter){  
        this.obj=fileWriter.save();  
    }  
  
    public void undo(FileWriterUtil fileWriter){  
        fileWriter.undoToLastSave(obj);  
    }  
}
```

Create the client

```
public class FileWriterClient {  
  
    public static void main(String[] args) {  
  
        FileWriterCaretaker caretaker = new FileWriterCaretaker();  
  
        FileWriterUtil fileWriter = new FileWriterUtil("data.txt");  
        fileWriter.write("First Set of Data:\nMyra\nLucy\n");  
        System.out.println(fileWriter+"\n\n");  
    }  
}
```

Create the client

- caretaker calls the save() method causing the originator save method to return a memento object to caretaker (caretaker saves it as a member variable)
- the caretaker calls the undo() method on the originator passing the maintained memento object
 - originator accepts the memento, using it to restore its previous state

```
// lets save the file  
caretaker.save(fileWriter);  
//now write something else  
fileWriter.write("Second Set of Data:\nJason\n");  
  
//checking file contents  
System.out.println(fileWriter+"\n\n");  
  
//lets undo to last save  
caretaker.undo(fileWriter);  
  
//checking file content again  
System.out.println(fileWriter+"\n\n");  
  
}  
}
```

Output

First Set of Data

Myra

Lucy

First Set of Data

Second Set of Data

Myra

Lucy

Jason

First Set of Data

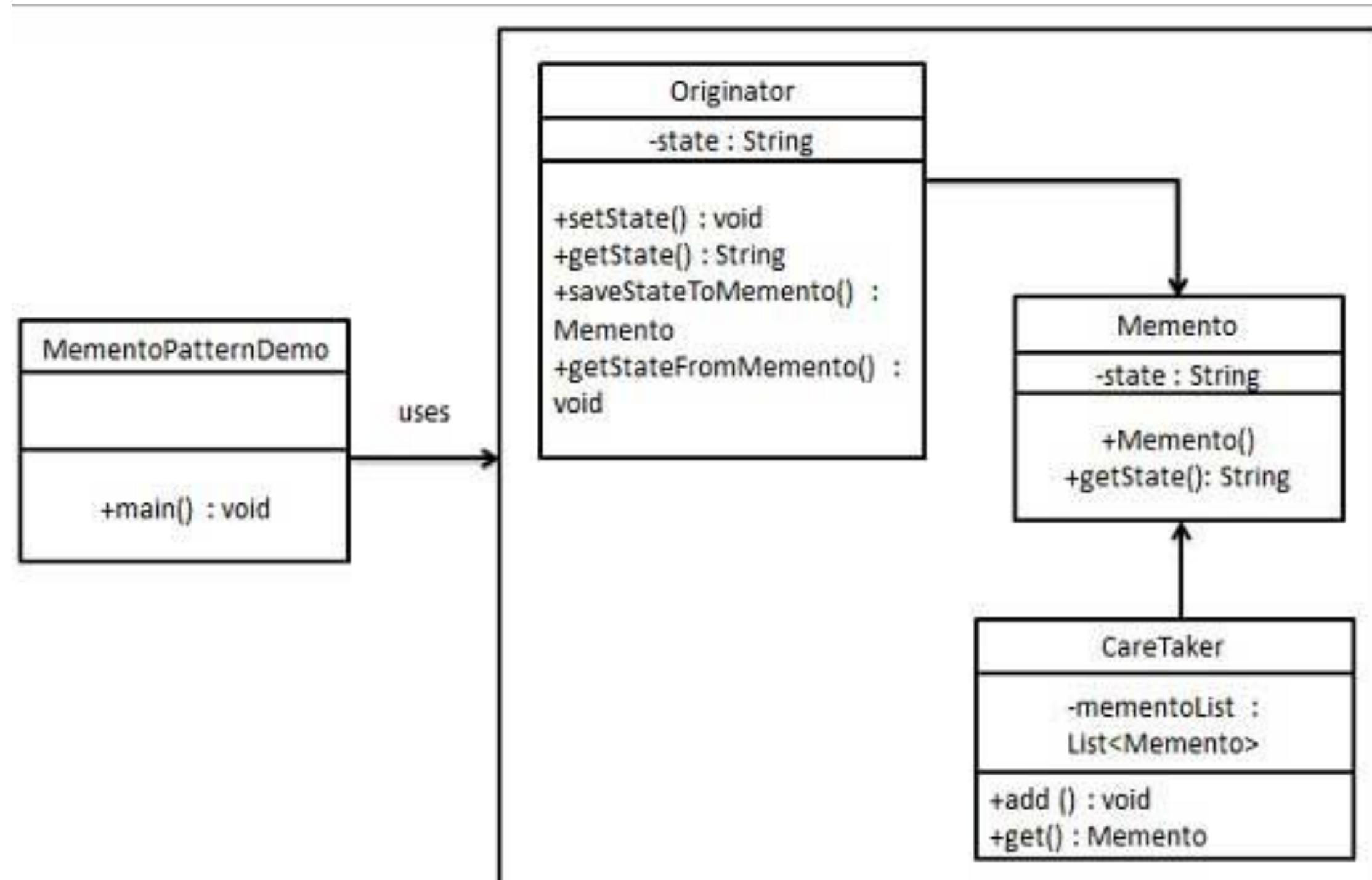
Myra

Lucy

Demonstration

Jason Fedin

Class Diagram



Step 1 – Create the memento class (Memento.java)

```
public class Memento {  
    private String state;  
  
    public Memento(String state) {  
        this.state = state;  
    }  
  
    public String getState() {  
        return state;  
    }  
}
```

Step 2 - Create Originator class (Originator.java)

```
public class Originator {  
    private String state;  
  
    public void setState(String state){  
        this.state = state;  
    }  
  
    public String getState(){  
        return state;  
    }  
  
    public Memento saveStateToMemento(){  
        return new Memento(state);  
    }  
  
    public void getStateFromMemento(Memento memento){  
        state = memento.getState();  
    }  
}
```

Step 3 – Create the CareTaker class

```
import java.util.ArrayList;  
import java.util.List;  
  
public class CareTaker {  
    private List<Memento> mementoList = new ArrayList<Memento>();  
  
    public void add(Memento state){  
        mementoList.add(state);  
    }  
  
    public Memento get(int index){  
        return mementoList.get(index);  
    }  
}
```

Step 4 – Create the Client

```
public class MementoPatternDemo {  
    public static void main(String[] args) {  
        Originator originator = new Originator();  
        CareTaker careTaker = new CareTaker();  
  
        originator.setState("State #1");  
        originator.setState("State #2");  
        careTaker.add(originator.saveStateToMemento());  
  
        originator.setState("State #3");  
        careTaker.add(originator.saveStateToMemento());  
  
        originator.setState("State #4");  
        System.out.println("Current State: " + originator.getState());  
  
        originator.getStateFromMemento(careTaker.get(0));  
        System.out.println("First saved State: " + originator.getState());  
        originator.getStateFromMemento(careTaker.get(1));  
        System.out.println("Second saved State: " + originator.getState());  
    }  
}
```

Output

Current State: State #4

First saved State: State #2

Second saved State: State #3

The Observer pattern

Jason Fedin

Overview

- the observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically
- many objects need to be notified whenever an event occurs and want to be notified automatically
 - however, I do not want to change the broadcasting object every time there is a change to the set of objects listening to the broadcast
 - would be like having to change a radio transmitter every time a new car radio comes to town
 - want to decouple the notify-ers and the notify-ees
 - this is what the observer pattern solves

Overview

- in this pattern, there are many observers (objects) which are observing a particular subject (object)
 - observers register themselves to a subject and are automatically notified when the subject changes
 - when they lose interest in the subject they simply unregister from the subject
- provides a loosely coupled design between objects that interact
 - more flexibility with changing requirements
- one of the most common patterns that is used in software development

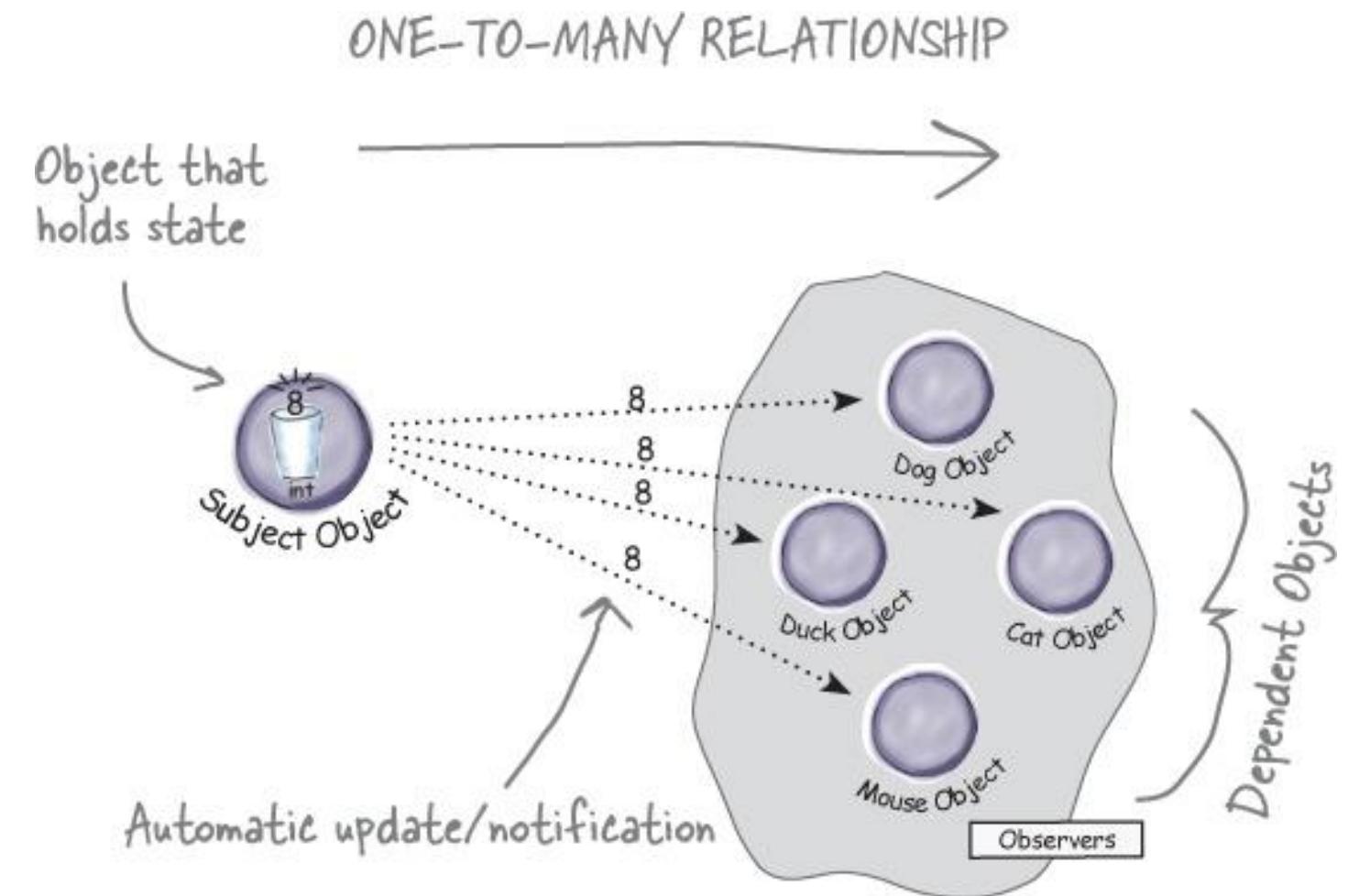
Examples

- a great example is how newspapers or magazines subscriptions work
- a newspaper publisher goes into business and begins publishing newspapers
- you subscribe to a particular publisher, and every time there is a new edition it gets delivered to you
 - as long as you remain a subscriber, you get new newspapers
- you unsubscribe when you do not want papers anymore
 - they stop being delivered
- while the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper

More examples

- how about a celebrity who has many fans
 - each of these fans wants to get all the latest updates of his/her favorite celebrity
 - he/she can follow the celebrity as long as his/her interest persists
 - when they lose interest, they simply stop following that celebrity
 - the fan is an observer and the celebrity is the subject
- consider a simple UI-based example, where this UI is connected with some database
 - a user can execute some query through that UI and after searching the database, the result is reflected back in the UI
 - if a change occurs in the database, the UI should be notified so that it can update its display according to the change
- heavily used in GUI toolkits and event listeners
 - the button(subject) and onClickListener(observer) are modelled with observer pattern
- social media, RSS feeds, email subscriptions in which you have the option to follow or subscribe and you receive latest notification
- all users of an app on the google play store get notified if there is an update

One-to-many relationship



Head First Design Patterns by: Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra

- the Observer Pattern defines a one-to-many relationship between a set of objects
 - when the state of one object changes, all of its dependents are notified
- the subject and observers define the one-to-many relationship
 - observers are dependent on the subject such that when the subject's state changes, the observers get notified

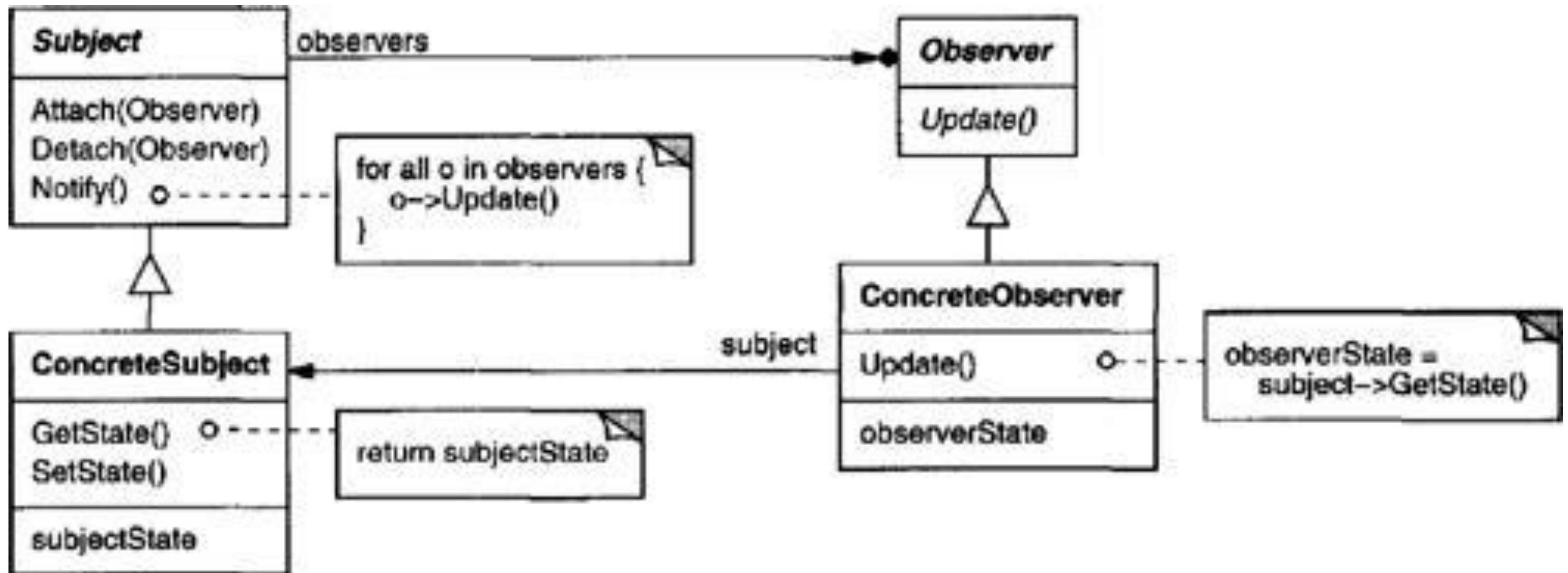
When to use this pattern?

- when a change to one object requires changing others, and you do not know how many objects need to be changed
- when multiple objects are dependent on the state of one object
- when an object should be able to notify other objects without making assumptions about who these objects are
 - you do not want these objects tightly coupled

Implementing the Observer

Jason Fedin

Class Diagram

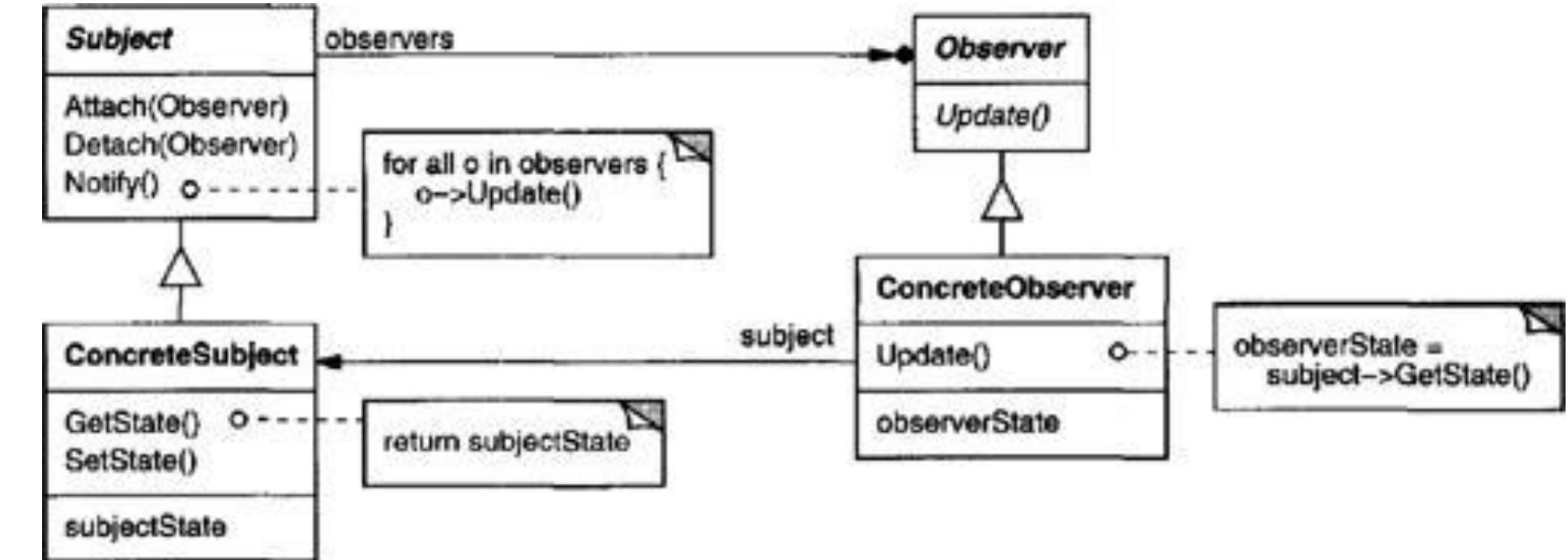


Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Overview

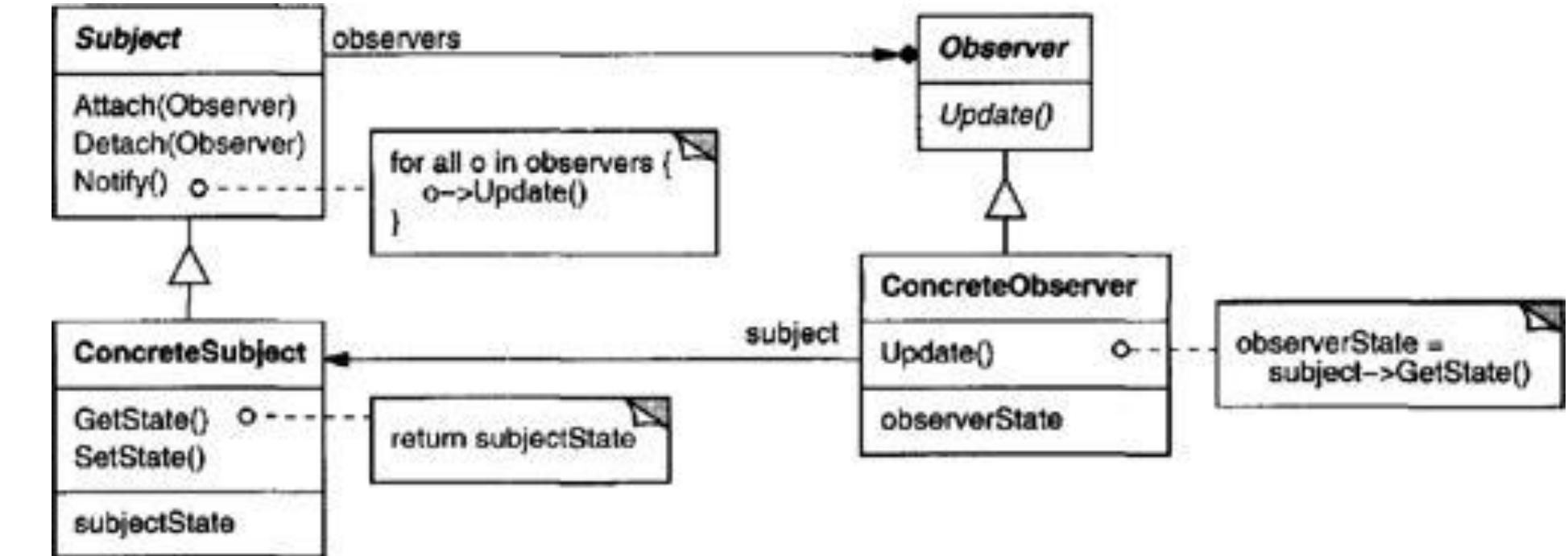
- to understand observer pattern implementation, first you need to understand the subject and observer objects
- the relation between subject and observer can easily be understood as an analogy to magazine subscription
 - a magazine publisher(subject) is in the business and publishes magazines (data)
 - If you (user of data/observer) are interested in the magazine you subscribe(register), and if a new edition is published it gets delivered to you
 - If you unsubscribe(unregister) you stop getting new editions
 - publisher does not know who you are and how you use the magazine, it just delivers it to you because you are a subscriber (loose coupling)
- Observer and Subject are interfaces or abstract classes

Participants



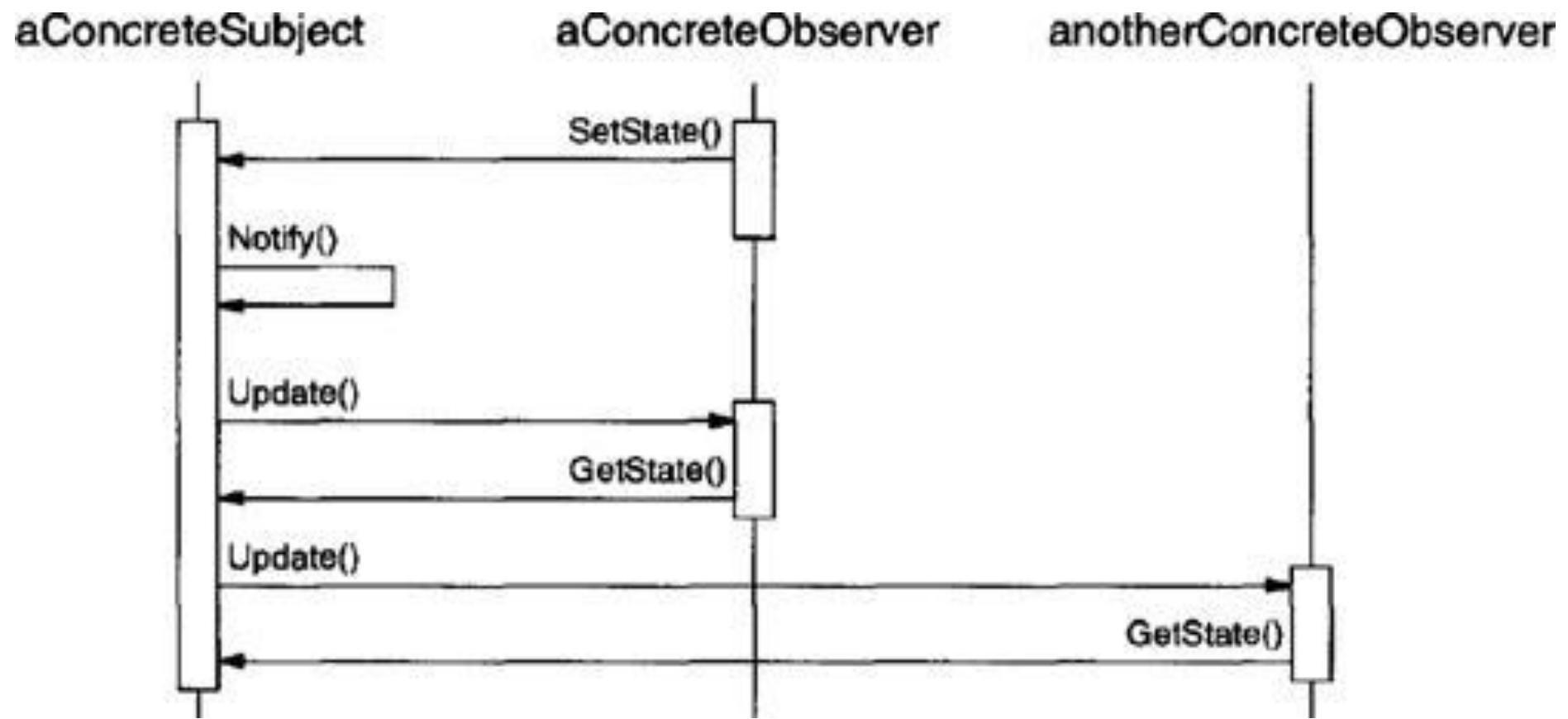
- **Subject**
 - knows its observers (contains a list of observers to notify)
 - any number of Observer objects may observe a subject
 - provides an interface for attaching and detaching Observer objects
 - methods allowing observers to register and unregister themselves
 - also contains a method to notify all the observers of any change
- **Observer**
 - defines an updating interface for objects that should be notified of changes in a subject
 - all potential observers need to implement this interface

Participants



- **ConcreteSubject**
 - stores state of interest to **ConcreteObserver** objects
 - sends a notification to its observers when its state changes
- **ConcreteObserver**
 - maintains a reference to a **ConcreteSubject** object
 - stores state that should stay consistent with the subject's
 - implements the **Observer** updating interface to keep its state consistent with the subject's

Sequence Diagram



- the `ConcreteSubject` notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own
- after being informed of a change in the concrete subject, a `ConcreteObserver` object may query the subject for information
 - `ConcreteObserver` uses this information to reconcile its state with that of the subject
- the Observer object that initiates the change request postpones its update until it gets a notification from the subject
 - `Notify` is not always called by the subject

Advantages of this implementation

- the Observer Pattern provides an object design where subjects and observers are loosely coupled
- the only thing the subject knows about an observer is that it implements a certain interface (the Observer interface)
 - does not need to know the concrete class of the observer, what it does, or anything else about it
- we can add new observers at any time
 - the only thing the subject depends on is a list of objects that implement the Observer interface
 - we can replace any observer at runtime with another observer and the subject will keep purring along

Advantages of this implementation

- we never need to modify the subject to add new types of observers
 - all we have to do is implement the Observer interface in the new class and register as an observer
 - subject does not care, it will deliver notifications to any object that implements the Observer interface
- we can reuse subjects or observers independently of each other
 - if we have another use for a subject or an observer, we can easily reuse them because the two are not tightly coupled
- changes to either the subject or an observer will not affect the other
 - as long as the objects still meet their obligations to implement the subject or observer interfaces

Observer pattern in java.util package

Jason Fedin

Java Implementation

- the Observer pattern is so useful that Java contains an implementation of it in its packages
- these are quite similar to our Subject and Observer interfaces, but give you a lot of functionality out of the box
 - you can also implement either a push or pull style of update to your observers
- the Observable class and the Observer interface make up the pattern
 - Observable class plays the role of the subject
- instead of the methods attach, detach, and notify, Java uses addObserver, deleteObserver, and notifyObservers

Java Implementation

- for an Object to become an observer, you will need to implement the Observer interface
 - call addObserver() on any Observable object
 - to remove yourself as an observer, just call deleteObserver()
- for the Observable to send notifications you need to be Observable by extending the java.util.Observable superclass
 - you first must call the setChanged() method to signify that the state has changed in your object
 - then call one of two notifyObservers() methods (notifyObservers() or notifyObservers(Object arg))

Java Implementation

- for an Observer to receive notifications, It must implement the update method
 - the signature of the method is a bit different
 - if you want to “push” data to the observers, you can pass the data as a data object to the notifyObservers(arg) method
 - If not, then the Observer has to “pull” the data it wants from the Observable object passed to it
- the setChanged() method is used to signify that the state has changed
 - when notifyObservers() is called it should update its observers
 - if notifyObservers() is called without first calling setChanged(), the observers will NOT be notified
- you need to call setChanged() for notifications to work
 - you may also want to use the clearChanged() method, which sets the changed state back to false
 - and the hasChanged() method, which tells you the current state of the changed flag

Observations on Java implementation

- the `java.util.Observable` implementation has a number of problems that limit its usefulness and reuse
 - not widely used
 - implementation is really simple
- `Observable` is a class
 - you have to subclass it
 - means you can not add on the `Observable` behavior to an existing class that already extends another superclass
 - limits its reuse potential (and isn't that why we are using patterns in the first place?)
- there is no `Observable` interface
 - you cannot create your own implementation that plays well with Java's built-in Observer API
 - you do not have the option of swapping out the `java.util` implementation for another (say, a new, multithreaded implementation)
- `Observable` may serve your needs if you can extend `java.util.Observable`
- On the other hand, more often than not, you will need to create your own implementation

Example in intelliJ

Jason Fedin

Overview

- Lets create a simple java program that demonstrates the observer pattern
- we will implement a simple topic that observers can register for
- whenever any new message is posted to the topic, all the registered observers will be notified and they can consume the message
- I am going to implement this application so that observers can also ask subjects if there are any updates
 - add a getUpdate to subject interface
 - add a reference to the subject in the concrete observer
 - add a setSubject (attach) method in the concrete observer so that the observer knows its subject
 - if update method of concrete observer, return latest subject string

Create the subject interface

- knows its observers (contains a list of observers to notify)
- any number of Observer objects may observe a subject
- provides an interface for attaching and detaching Observer objects
 - methods allowing observers to register and unregister themselves
 - also contains a method to notify all the observers of any change

```
public interface Subject {  
    //methods to register and unregister observers  
    public void register(Observer obj);  
    public void unregister(Observer obj);  
  
    //method to notify observers of change  
    public void notifyObservers();  
  
    //method to get updates from subject, not required, but, added so observers can query to see if there is an update  
    public Object getUpdate(Observer obj);  
}
```

Create the Observer Interface

- defines an updating interface for objects that should be notified of changes in a subject
- all potential observers need to implement this interface

```
public interface Observer {  
  
    //method to update the observer, used by subject  
    public void update();  
  
    // attach with subject to observe, not required, but, added so that the  
    // observer can query the subject to see if an update has occurred  
    public void setSubject(Subject sub);  
}
```

Create the concreteSubject

- stores state of interest to ConcreteObserver objects
- sends a notification to its observers when its state changes

```
import java.util.ArrayList;  
import java.util.List;  
  
public class MyTopic implements Subject {  
  
    private List<Observer> observers;  
    private String message;  
    private boolean changed;  
  
    public MyTopic(){  
        this.observers=new ArrayList<>();  
    }  
}
```

Create the concreteSubject

```
@Override  
public void register(Observer obj) {  
    if(obj == null) throw new NullPointerException("Null Observer");  
    if(!observers.contains(obj)) observers.add(obj);  
}
```

```
@Override  
public void unregister(Observer obj) {  
    observers.remove(obj);  
}
```

Create the concrete subject

```
    @Override  
    public void notifyObservers() {  
        List<Observer> observersLocal = null;  
        if (!changed)  
            return;  
        observersLocal = new ArrayList<>(this.observers);  
        this.changed=false;  
        for (Observer obj : observersLocal) {  
            obj.update();  
        }  
    }
```

Create the concrete subject

```
@Override  
public Object getUpdate(Observer obj) {  
    return this.message;  
}  
  
//method to post message to the topic, change state (trigger notifications)  
public void postMessage(String msg){  
    System.out.println("Message Posted to Topic:"+msg);  
    this.message=msg;  
    this.changed=true;  
    notifyObservers();  
}  
}
```

Overview of implementing concrete subject

- the method implementation to register and unregister an observer is very simple
- the extra method is postMessage() that will be used by client application to post String message to the topic
 - notice the boolean variable to keep track of the change in the state of topic and used in notifying observers
 - variable is required so that if there is no update and somebody calls notifyObservers() method, it does not send false notifications to the observers

Create the concrete observer

- maintains a reference to a ConcreteSubject object
- stores state that should stay consistent with the subject's
- implements the Observer updating interface to keep its state consistent with the subject's

```
public class MyTopicSubscriber implements Observer {
```

```
    private String name;
```

```
    // not required, could just pass subjects state to update method, but, also used to attach  
    private Subject topic;
```

```
    public MyTopicSubscriber(String nm){  
        this.name=nm;  
    }
```

Create the concrete observer

```
@Override  
public void update() {  
    // this method could take data of subjects state that was changed, do not need to ask topic for it  
    String msg = (String) topic.getUpdate(this);  
    if(msg == null){  
        System.out.println(name+": No new message");  
    }else  
        System.out.println(name+": Consuming message:"+msg);  
}  
  
@Override  
public void setSubject(Subject sub) {  
    // not required, added so observer can ask subject for state  
    this.topic=sub;  
}  
}
```

Concrete observer summary

- notice the implementation of update() method where it's calling Subject getUpdate() method to get the message to consume
 - we could have avoided this call by passing message as argument to update() method
 - would not need a subject reference (however, this reference allows us to have observer ask subject for state)

Create the client

```
public class ObserverPatternTest {  
  
    public static void main(String[] args) {  
        //create subject  
        MyTopic topic = new MyTopic();  
  
        //create observers  
        Observer obj1 = new MyTopicSubscriber("Obj1");  
        Observer obj2 = new MyTopicSubscriber("Obj2");  
        Observer obj3 = new MyTopicSubscriber("Obj3");
```

Create the client

```
//register observers to the subject, could have added the registration as part of the observer constructor (passing in the subject)
topic.register(obj1);
topic.register(obj2);
topic.register(obj3);

//attach observer to subject (not required, could have passed in subject state to update method of observer)
obj1.setSubject(topic);
obj2.setSubject(topic);
obj3.setSubject(topic);

//check if any update is available, not required
obj1.update();

//now send message to subject, trigger notifyAll
topic.postMessage("New Message");
}

}
```

Output

Obj1:: No new message

Message Posted to Topic:New Message

Obj1:: Consuming message::New Message

Obj2:: Consuming message::New Message

Obj3:: Consuming message::New Message

Demonstration

Jason Fedin

Create the subject interface

```
import java.util.ArrayList;  
import java.util.Iterator;  
  
// Implemented by Cricket data to communicate with observers  
interface Subject  
{  
    public void registerObserver(Observer o);  
    public void unregisterObserver(Observer o);  
    public void notifyObservers();  
}
```

Create the Observer interface

```
// This interface is implemented by all those  
// classes that are to be updated whenever there  
// is an update from CricketData  
interface Observer  
{  
    public void update(int runs, int wickets,  
                      float overs);  
}
```

Create the concrete subject

```
class CricketData implements Subject {  
    int runs;  
    int wickets;  
    float overs;  
    ArrayList<Observer> observerList;  
  
    public CricketData() {  
        observerList = new ArrayList<Observer>();  
    }  
  
    @Override  
    public void registerObserver(Observer o) {  
        observerList.add(o);  
    }  
  
    @Override  
    public void unregisterObserver(Observer o) {  
        observerList.remove(observerList.indexOf(o));  
    }  
}
```

Create the concrete subject

```
@Override  
public void notifyObservers() {  
    for (Iterator<Observer> it = observerList.iterator(); it.hasNext();) {  
        Observer o = it.next();  
        o.update(runs,wickets,overs);  
    }  
}  
  
// get latest runs from stadium  
private int getLatestRuns() {  
    return 90;  
}  
  
// get latest wickets from stadium  
private int getLatestWickets() {  
    return 2;  
}
```

Create the concrete subject

```
// get latest overs from stadium  
private float getLatestOvers() {  
    return (float)10.2;  
}  
  
// This method is used update displays when data changes  
public void dataChanged() {  
    //get latest data  
    runs = getLatestRuns();  
    wickets = getLatestWickets();  
    overs = getLatestOvers();  
  
    notifyObservers();  
}  
}
```

Create the concrete observers

```
class AverageScoreDisplay implements Observer {  
    private float runRate;  
    private int predictedScore;  
  
    public void update(int runs, int wickets, float overs) {  
        this.runRate = (float)runs/overs;  
        this.predictedScore = (int)(this.runRate * 50);  
        display();  
    }  
  
    public void display() {  
        System.out.println("\nAverage Score Display: \n"  
            + "Run Rate: " + runRate +  
            "\nPredictedScore: " +  
            predictedScore);  
    }  
}
```

Create the concrete observer

```
class CurrentScoreDisplay implements Observer {  
    private int runs, wickets;  
    private float overs;  
  
    public void update(int runs, int wickets, float overs) {  
        this.runs = runs;  
        this.wickets = wickets;  
        this.overs = overs;  
        display();  
    }  
  
    public void display() {  
        System.out.println("\nCurrent Score Display:\n"  
            + "Runs: " + runs +  
            "\nWickets:" + wickets +  
            "\nOvers: " + overs);  
    }  
}
```

Create the client

```
class Main {  
    public static void main(String args[]) {  
        // create objects for testing  
        AverageScoreDisplay averageScoreDisplay = new AverageScoreDisplay();  
        CurrentScoreDisplay currentScoreDisplay = new CurrentScoreDisplay();  
  
        // pass the displays to Cricket data  
        CricketData cricketData = new CricketData();  
  
        // register display elements  
        cricketData.registerObserver(averageScoreDisplay);  
        cricketData.registerObserver(currentScoreDisplay);  
  
        // in real app you would have some logic to call this function when data changes  
        cricketData.dataChanged();  
  
        //remove an observer  
        cricketData.unregisterObserver(averageScoreDisplay);  
  
        // now only currentScoreDisplay gets the notification  
        cricketData.dataChanged();  
    }  
}
```

Example output

Average Score Display:

Run Rate: 8.823529

PredictedScore: 441

Current Score Display:

Runs: 90

Wickets:2

Overs: 10.2

Current Score Display:

Runs: 90

Wickets:2

Overs: 10.2

The State design pattern

Jason Fedin

Overview

- the state design pattern allows an object to alter its behavior when its internal state changes
 - object will appear to change its class
 - an object's behavior is the result of the function of its state
 - behavior gets changed at runtime depending on the state (polymorphism)
- normally, If we have to change behavior of an object based on its state, we have a state variable in the Object and use if-else condition block to perform different actions based on the state
 - with the state pattern, we can remove dependencies on if/else or switch/case conditional logic
- we create objects which represent various states and a context object whose behavior varies as its state object changes
- used to provide a systematic and loosely coupled way to achieve state changes through a Context and State implementations
- is very similar to Strategy Pattern which we will discuss in a future lecture

Examples

- a traffic signal
 - stop, go, and slow down states
- consider a TCP network connection
 - an object that is responsible for communication can be in various states
 - established, listening, and closed
 - when a TCPConnection object receives requests from other objects, it responds differently depending on its current state
 - e.g. an Open request depends on whether the connection is in its Closed state or its Established state
- how about a job processing application where we can process only one job at a time
 - if a new job appears, either the application will process that job (accepted)
 - or it will signal that the new job cannot be processed at this moment because the system is already processing the maximum number of jobs in it (queued)

when to use the state pattern?

- when an object's behavior depends on its state, and it must change its behavior at run-time depending on that state
- when operations have large, multipart conditional statements that depend on the object's state
 - state is usually represented by one or more enumerated constants
 - often, several operations will contain this same conditional structure
 - state pattern puts each branch of the conditional in a separate class
 - lets you treat the object's state as an object in its own right that can vary independently from other objects

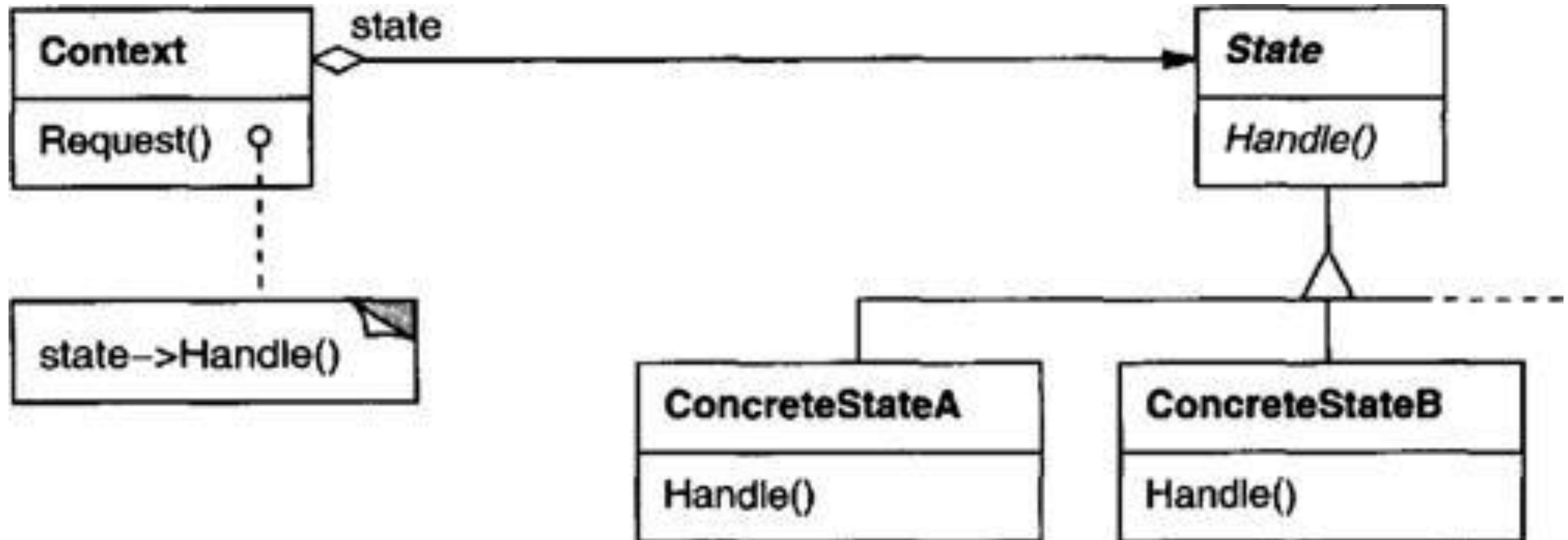
Advantages

- puts all behavior associated with a state into one object
 - Improves cohesion
- allows state transition logic to be incorporated into a state object rather than in a monolithic if or switch statement
- helps avoid inconsistent states since state changes occur by rebinding one variable rather than several
- very easy to add more states for additional behavior
 - makes code more robust, easily maintainable and flexible
- one drawback is that the pattern does increase the number of objects (one for each state)

Implementing the state design pattern

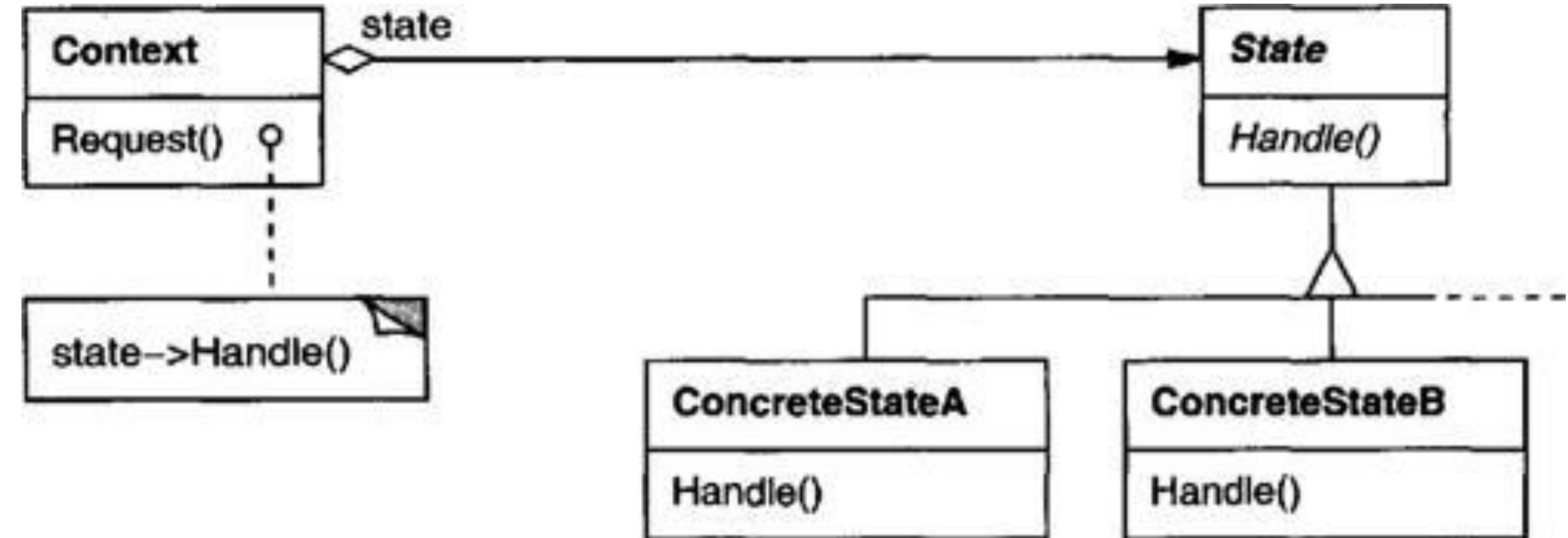
Jason Fedin

Class Diagram



Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Participants



- Context
 - defines the interface of interest to clients
 - maintains an instance of a **ConcreteState** subclass that defines the current state
- State
 - defines an interface for encapsulating the behavior associated with a particular state of the Context
- **ConcreteState** subclasses
 - each subclass implements a behavior associated with a state of the Context

Workflow

- the Context delegates state-specific requests to the current ConcreteState object
 - has a reference to the concrete state object
- a context may pass itself as an argument to the State object handling the request
 - lets the State object access the context if necessary
- clients can configure a context with State objects
 - once a context is configured, its clients don't have to deal with the State objects directly
- either Context or the ConcreteState subclasses can decide which state takes priority over another and under what circumstances

Advantages of implementation

- as mentioned previously, this pattern puts all behavior associated with a particular state into one object
 - new states and transitions can be added easily by defining new subclasses
- large conditional statements are undesirable
 - monolithic and tend to make the code less explicit, which in turn makes them difficult to modify and extend
 - state pattern offers a better way to structure state-specific code
 - logic that determines the state transitions does not reside in monolithic if or switch statements but instead is partitioned between the State subclasses
- makes state transitions explicit by introducing separate objects for different states
- state objects can protect the Context from inconsistent internal states
 - state transitions are atomic from the Context's perspective
 - happen by rebinding one variable (the Context's State object variable)
- state objects can be shared
 - if State objects have no instance variables

Summary

- allows an object to have many different behaviors that are based on its internal state
- represents state as an object
- the context gets its behavior by delegating to the current state object it is composed with
- by encapsulating each state into a class, we localize any changes that will need to be made
- state transitions can be controlled by the State classes or by the Context classes

Implementing the state design pattern

Jason Fedin

Example in intellij

- suppose we want to implement a TV Remote with a simple button to perform action
 - If the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV
- normally, if we have to change behavior of an object based on its state, we have a state variable in the Object and use if-else condition block to perform different actions based on the state
- lets look at that in the next few slides

Implementation without state pattern

```
public class TVRemoteBasic {  
  
    private String state="";  
  
    public void setState(String state){  
        this.state=state;  
    }  
  
    public void doAction(){  
        if(state.equalsIgnoreCase("ON")){  
            System.out.println("TV is turned ON");  
        }else if(state.equalsIgnoreCase("OFF")){  
            System.out.println("TV is turned OFF");  
        }  
    }  
}
```

Implementation without state pattern

```
public static void main(String args[]){
    TVRemoteBasic remote = new TVRemoteBasic();

    remote.setState("ON");
    remote.doAction();

    remote.setState("OFF");
    remote.doAction();
}

}
```

- Notice that client code should know the specific values to use for setting the state of remote. Further more if number of states increase then the tight coupling between implementation and the client code will be very hard to maintain and extend

Using the State pattern to implement (Create the State interface)

- defines an interface for encapsulating the behavior associated with a particular state of the Context

abstract class RemoteControl

```
{  
    public abstract void pressSwitch(TV context);  
}
```

Create the concrete state implementations

- In this example, we can have two states – one for turning TV on and another to turn it off. So we will create two concrete state implementations for these behaviors
- each subclass implements a behavior associated with a state of the Context

```
class On extends RemoteControl {  
    @Override  
    public void pressSwitch(TV context) {  
        System.out.println("I am already On .Going to be Off now");  
        context.setState(new Off());  
    }  
}
```

Create the concrete state implementations

```
class Off extends RemoteControl  
{  
    @Override  
    public void pressSwitch(TV context){  
        System.out.println("I am Off .Going to be On now");  
        context.setState(new On());  
    }  
}
```

Create the context class

- defines the interface of interest to clients
- the Context keeps a reference of its current state and forwards the request to the state implementation

```
class TV {  
    private RemoteControl state;  
  
    public RemoteControl getState() {  
        return state;  
    }  
    public void setState(RemoteControl state) {  
        this.state = state;  
    }  
    public TV(RemoteControl state) {  
        this.state=state;  
    }  
  
    public void pressButton() {  
        state.pressSwitch(this);  
    }  
}
```

Create the client

```
class StatePatternEx {  
    public static void main(String[] args) {  
        System.out.println("***State Pattern Demo***\n");  
  
        //Initially TV is Off  
        Off initialState = new Off();  
  
        TV tv = new TV(initialState);  
  
        //First time press  
        tv.pressButton();  
        //Second time press  
        tv.pressButton();  
    }  
}
```

Demonstration

Jason Fedin

Create the State interface or abstract class

```
interface MobileAlertState
```

```
{
```

```
    public void alert();
```

```
}
```

Create the context

```
class AlertStateContext {  
    private MobileAlertState currentState;  
  
    public AlertStateContext() {  
        currentState = new Vibration(); // default state  
    }  
  
    public void setState(MobileAlertState state) {  
        currentState = state;  
    }  
  
    public void alert() {  
        currentState.alert();  
    }  
}
```

Create the concrete state classes

```
class Vibration implements MobileAlertState {  
    @Override  
    public void alert() {  
        System.out.println("vibration...");  
    }  
}
```

Create the concrete state class

```
class Silent implements MobileAlertState
```

```
{  
    @Override  
    public void alert() {  
        System.out.println("silent...");  
    }  
}
```

```
}
```

Create the client

```
class StatePattern {  
    public static void main(String[] args) {  
        AlertStateContext stateContext = new AlertStateContext();  
        stateContext.alert();  
        stateContext.alert();  
        stateContext.setState(new Silent());  
        stateContext.alert();  
        stateContext.alert();  
        stateContext.alert();  
    }  
}
```

Output

vibration...

vibration...

silent...

silent...

silent...

Strategy design pattern

Jason Fedin

Overview

- the strategy design pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable
 - lets the algorithm vary independently from client to client
 - conceptually, all of these algorithms do the same things
 - just have different implementations
- we can select the behavior of an algorithm dynamically at runtime
 - can help us to avoid dealing with complex algorithm-specific data structures
 - we let the client application pass the algorithm to be used as a parameter
- allows us to create objects which represent various strategies and a context object whose behavior varies as per its strategy object
 - the strategy object changes the executing algorithm of the context object

Examples

- in a soccer match, strategies will differ
 - if Team A is leading Team B by a score of 1-0
 - instead of attacking, Team A becomes defensive
 - Team B goes for an all-out attack to score
- we can think of two dedicated storage devices
 - when one device becomes “full”, we start storing the data in the second available device
 - a runtime check is necessary before the storing of data, and based on the situation, we will proceed
- the Collections.sort() method from the java API uses the strategy pattern
 - takes a Comparator parameter
 - based on the different implementations of Comparator interfaces
 - the Objects are getting sorted in different ways

Another example

- suppose we have different algorithms for breaking a stream of text into lines
- hard-wiring all such algorithms into the classes that require them is not desirable for several reasons
 - clients that need line breaking get more complex if they include the line-breaking code
 - makes clients bigger and harder to maintain
 - different algorithms will be appropriate at different times
 - do not want to support multiple line breaking algorithms if we do not use them all
 - difficult to add new algorithms and vary existing ones when line breaking is an integral part of a client
- we can avoid the above problems by defining classes that encapsulate different line-breaking algorithms (the strategy pattern)

Principles of the Strategy pattern

- objects have responsibilities
- different, specific implementations of these responsibilities are manifested through the use of polymorphism
- there is a need to manage several different implementations of what is, conceptually, the same algorithm
- It is a good design practice to separate behaviors that occur in the problem domain from each other (decoupling)
 - allows me to change the class responsible for one behavior without adversely affecting another

Strategy vs. the State pattern

- the strategy and state pattern are very similar
- think of the Strategy Pattern as subclasses decide how to implement steps in an algorithm
- think of the State Pattern as an alternative to putting lots of conditionals in your context
 - encapsulate interchangeable behaviors and use delegation to decide which behavior to use
- one difference between the two is that with the state pattern the Context contains state as an instance variable
 - there can be multiple tasks whose implementation can be dependent on the state
 - in the strategy pattern, strategy is passed as argument to the method and the context object does not have any variable to store it

Advantages

- by encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced
- applications can switch strategies at run-time (polymorphism)
- enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements
- simplifies unit testing because each algorithm is in its own class and can be tested through its interface alone
 - the developer does not need to worry about interactions caused by coupling
 - developer is able to test each algorithm independently and not worry about all the combinations possible

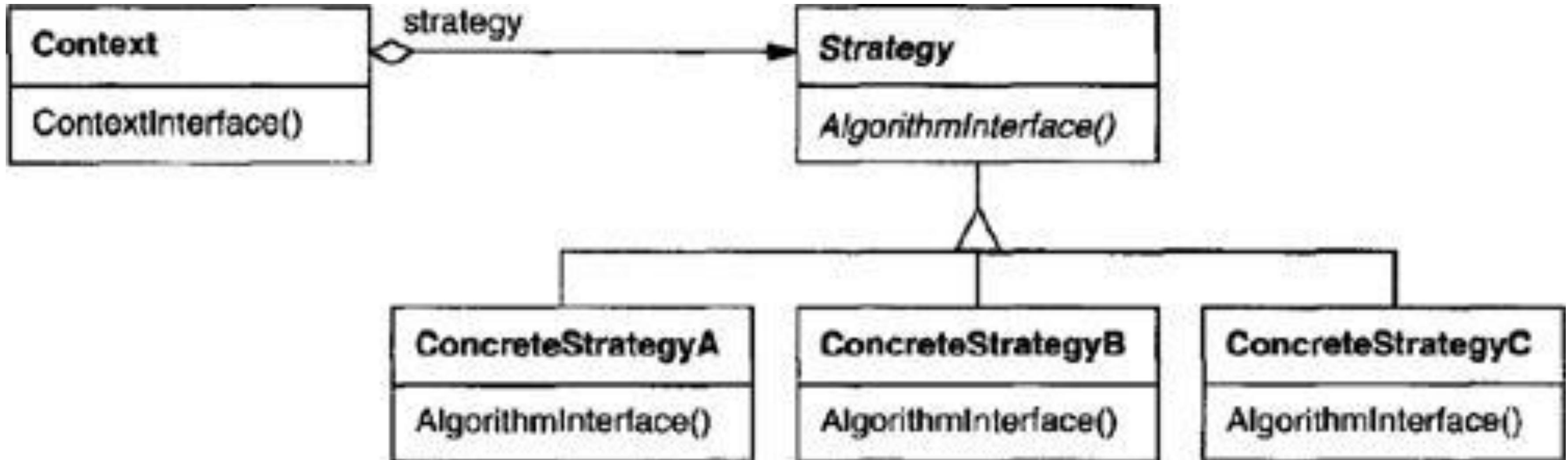
When to use the strategy pattern?

- when many related classes differ only in their behavior
 - strategies provide a way to configure a class with one of many behaviors
- when you need different variants of an algorithm
 - we want our application to be flexible to chose any of the algorithm at runtime for specific task
- when an algorithm uses data that clients should not know about
 - use the Strategy pattern to avoid exposing complex, algorithm-specific data structures
- when a class defines many behaviors, and these appear as multiple conditional statements in its operations
 - instead of many conditionals, move related conditional branches into their own Strategy class

Implementing the Strategy pattern

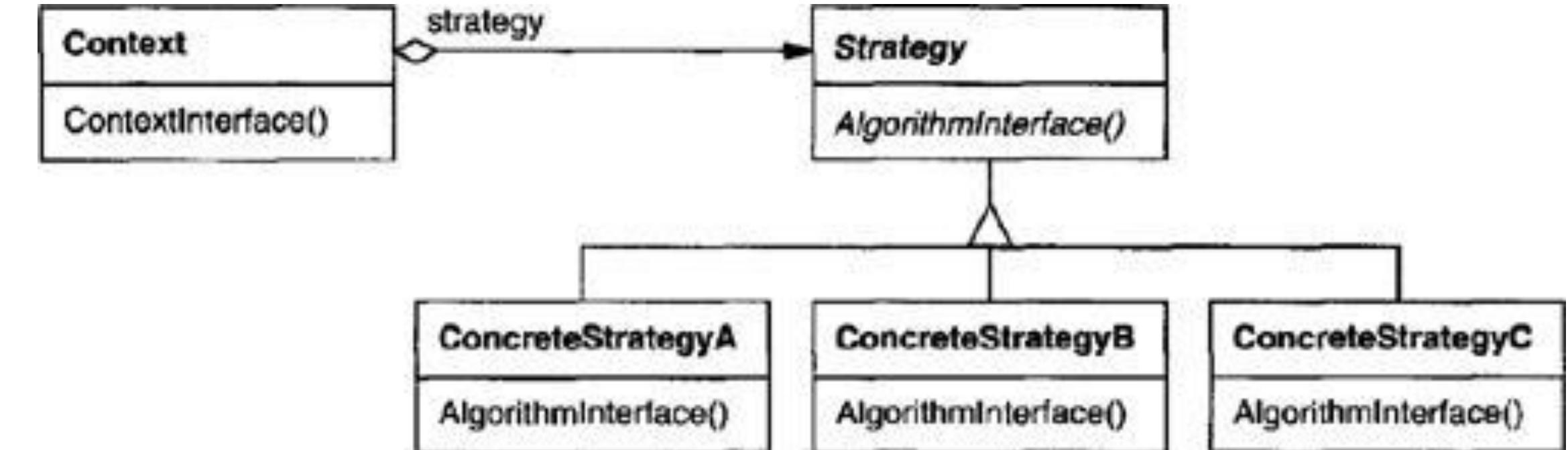
Jason Fedin

Class Diagram



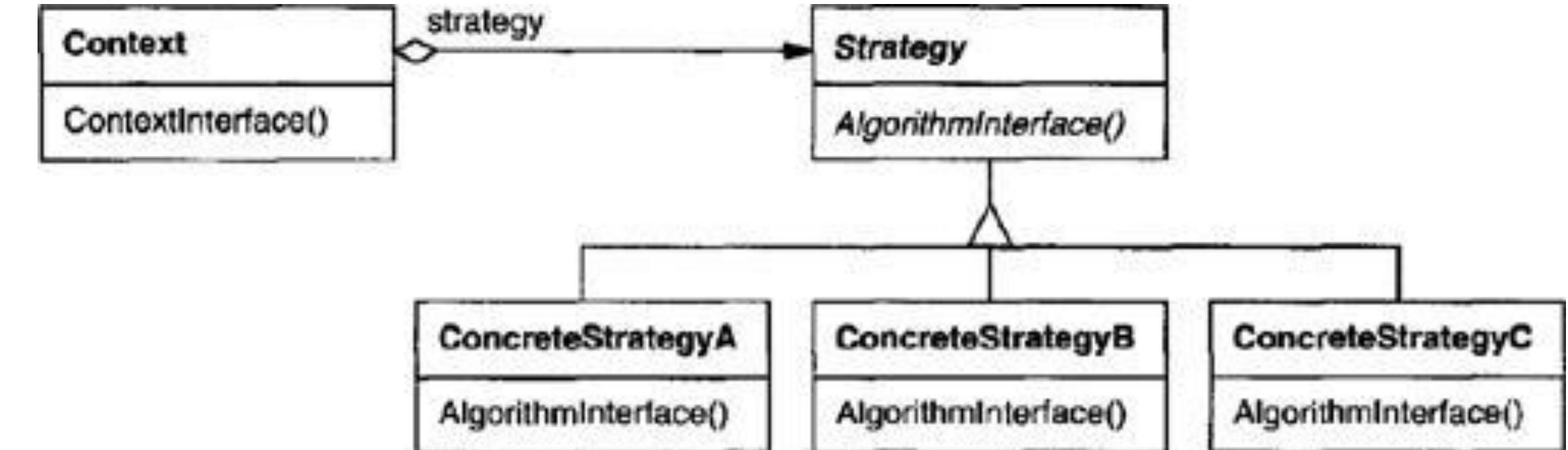
Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Participants



- **Strategy**
 - declares an interface common to all supported algorithms
 - Context uses this interface to call the algorithm defined by a ConcreteStrategy
 - specifies how the different algorithms are used
- **ConcreteStrategy**
 - implements the algorithm using the Strategy interface
- **Context**
 - is configured with a ConcreteStrategy object
 - maintains a reference to a Strategy object or gets passed as a parameter to a method
 - may define an interface that lets Strategy access its data

Participants



- Context uses a specific `ConcreteStrategy`
- Strategy and Context interact to implement the chosen algorithm
 - sometimes Strategy must query Context
 - Context forwards requests from its client to Strategy
 - clients usually create and pass a `ConcreteStrategy` object to the context
 - clients interact with the context exclusively
 - often a family of `ConcreteStrategy` classes for a client to choose from

Advantages of implementation

- a family of algorithms can be defined as a class hierarchy
 - can be used interchangeably to alter application behavior without changing its architecture
 - inheritance can help factor out common functionality of the algorithms
- an alternative to subclassing
 - inheritance offers another way to support a variety of algorithms or behaviors
 - you can subclass a Context class directly to give it different behaviors
 - hard-wires the behavior into Context
 - mixes the algorithm implementation with Context's
 - Context is harder to understand, maintain, and extend
 - you cannot vary the algorithm dynamically
 - you wind up with many related classes whose only difference is the algorithm or behavior they employ
 - encapsulating the algorithm in separate Strategy classes
 - lets you vary the algorithm independently of its context
 - easier to switch, understand, and extend

Advantages of implementation

- data structures used for implementing the algorithm are completely encapsulated in Strategy classes
 - the implementation of an algorithm can be changed without affecting the Context class
 - eliminates conditional statements
- a choice of implementations
 - strategies can provide different implementations of the same behavior
 - client can choose among strategies with different time and space trade-offs
- clients must be aware of different Strategies
 - has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one
 - clients might be exposed to implementation issues

Implementation issues

- the Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa
 - have Context pass data in parameters to Strategy operations
 - keeps Strategy and Context decoupled
 - context can pass itself as an argument to Strategy operations
 - lets the strategy call back on the context as required
- Context class may be simplified if it is meaningful not to have a Strategy object
 - Context checks to see if it has a Strategy object before accessing it
 - If there is one, then Context uses it normally
 - If there is not a strategy, then Context carries out default behavior
 - the benefit of this approach is that clients do not have to deal with Strategy objects at all unless they do not like the default behavior

Implementing the Strategy pattern

Jason Fedin

Example in intelliJ

- lets implement a simple Shopping Cart application where we have two payment strategies
 - Credit Card or PayPal
- First, we will create the interface for our strategy pattern example, in our case to pay the amount passed as an argument

Create the strategy interface

- declares an interface common to all supported algorithms
- Context uses this interface to call the algorithm defined by a ConcreteStrategy
- specifies how the different algorithms are used

```
public interface PaymentStrategy {  
    public void pay(int amount);  
}
```

Create the concrete strategy classes

```
public class CreditCardStrategy implements PaymentStrategy {  
    private String name;  
    private String cardNumber;  
    private String cvv;  
    private String dateOfExpiry;  
  
    public CreditCardStrategy(String nm, String ccNum, String cvv, String expiryDate) {  
        this.name=nm;  
        this.cardNumber=ccNum;  
        this.cvv=cvv;  
        this.dateOfExpiry=expiryDate;  
    }  
  
    @Override  
    public void pay(int amount) {  
        System.out.println(amount +" paid with credit/debit card");  
    }  
}
```

Create the concrete strategy classes

```
public class PaypalStrategy implements PaymentStrategy {  
    private String emailld;  
    private String password;
```

```
    public PaypalStrategy(String email, String pwd){  
        this.emailld=email;  
        this.password=pwd;  
    }
```

```
    @Override  
    public void pay(int amount) {  
        System.out.println(amount + " paid using Paypal.");  
    }  
}
```

Create a helper class for our context

```
public class Item {  
    private String upcCode;  
    private int price;  
  
    public Item(String upc, int cost){  
        this.upcCode=upc;  
        this.price=cost;  
    }  
  
    public String getUpcCode() {  
        return upcCode;  
    }  
  
    public int getPrice() {  
        return price;  
    }  
}
```

Create our context class

- Context forwards requests from its client to Strategy
 - client creates and passes a ConcreteStrategy object to the context
 - will require input as Payment strategy (passed as a parameter to pay method)

```
import java.text.DecimalFormat;  
import java.util.ArrayList;  
import java.util.List;  
  
public class ShoppingCart {  
    //List of items  
    List<Item> items;  
    public ShoppingCart(){  
        this.items=new ArrayList<Item>();  
    }  
    public void addItem(Item item){  
        this.items.add(item);  
    }
```

Create our context class

```
public void removeItem(Item item){  
    this.items.remove(item);  
}  
  
public int calculateTotal(){  
    int sum = 0;  
    for(Item item : items){  
        sum += item.getPrice();  
    }  
    return sum;  
}  
  
public void pay(PaymentStrategy paymentMethod){  
    int amount = calculateTotal();  
    paymentMethod.pay(amount);  
}
```

- Notice that payment method of shopping cart requires payment algorithm as argument and doesn't store it anywhere as instance variable.

Create the client

- clients create and pass a ConcreteStrategy object to the context
 - clients interact with the context exclusively
 - often a family of ConcreteStrategy classes for a client to choose from

```
public class ShoppingCartTest {  
    public static void main(String[] args) {  
        ShoppingCart cart = new ShoppingCart(); // the context  
  
        Item item1 = new Item("1234",10);  
        Item item2 = new Item("5678",40);  
  
        cart.addItem(item1);  
        cart.addItem(item2);  
  
        //pay by paypal  
        cart.pay(new PaypalStrategy("myemail@example.com", "mypwd"));  
  
        //pay by credit card  
        cart.pay(new CreditCardStrategy("Jason Fedin", "1234567890123456", "786", "12/15"));  
    }  
}
```

Output

50 paid using Paypal

50 paid with credit/debit card

- we could have used composition to create instance variable for strategies but we should avoid that as we want the specific strategy to be applied for a particular task

Demonstration

Jason Fedin

Create the Strategy interface

```
public interface Strategy {  
    public int doOperation(int num1, int num2);  
}
```

Create the concrete strategy classes

```
public class OperationAdd implements Strategy{  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 + num2;  
    }  
}
```

```
public class OperationSubstract implements Strategy{  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 - num2;  
    }  
}
```

Create the concrete strategy classes

```
public class OperationMultiply implements Strategy{  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 * num2;  
    }  
}
```

Create the context class (have a strategy as a reference)

```
public class Context {  
    private Strategy strategy;  
  
    public Context(Strategy strategy){  
        this.strategy = strategy;  
    }  
  
    public int executeStrategy(int num1, int num2){  
        return strategy.doOperation(num1, num2);  
    }  
}
```

Create the client

- Use the Context to see change in behaviour when it changes its Strategy.

```
public class StrategyPatternDemo {  
    public static void main(String[] args) {  
        Context context = new Context(new OperationAdd());  
        System.out.println("10 + 5 = " + context.executeStrategy(10, 5));  
  
        context = new Context(new OperationSubtract());  
        System.out.println("10 - 5 = " + context.executeStrategy(10, 5));  
  
        context = new Context(new OperationMultiply());  
        System.out.println("10 * 5 = " + context.executeStrategy(10, 5));  
    }  
}
```

Output

$10 + 5 = 15$

$10 - 5 = 5$

$10 * 5 = 50$

The template method design pattern

Jason Fedin

Overview

- the template method defines the skeleton of an algorithm in an operation, deferring some steps to subclasses
 - lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
 - helps us generalize a common process, at an abstract level, from a set of different procedures
- all about creating a template for an algorithm
 - a template is just a method that defines an algorithm as a set of steps
 - one or more of these steps is defined to be abstract (a method stub) and implemented by a subclass
 - ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation
- enables you to define the sequence of steps and then override those steps that need to change
- software reuse is the fundamental goal of this method
 - this is why the pattern is used in many class libraries and framework development
- one of the more common patterns in use today

examples

- in the real world templates are used all the time
 - for architectural plans, and throughout the engineering domain
 - a template plan may be defined which is then built on with further variations
- suppose we want to make pizza
 - the basic mechanism is the same, but extra materials are added based upon the customer's choice
 - whether he/she wants a vegetarian pizza or a non-vegetarian pizza
- for an engineering student, in general, most of the subjects in the first semester are common for all concentrations
 - later, additional papers are added in his/her course based on his/her specialization (Computer Science, Electronics, etc.)

more examples

- suppose we want to provide an algorithm to build a house
 - the steps need to be performed to build a house are building foundation, building pillars, building walls and windows
 - the important point is that we can not change the order of execution
 - we cannot build windows before building the foundation
 - so in this case we can create a template method that will use different methods to build the house
 - building the foundation for a house is same for all type of houses, whether it's a wooden house or a glass house
 - we can provide a base implementation for this, if subclasses want to override this method, they can but mostly it is common for all the types of houses
- another example would be the methods for connecting and querying Oracle or SQL Server databases
 - the methods may be different but, they share the same conceptual process
- examples in the JDK include
 - non-abstract methods of `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader` and `java.io.Writer`
 - non-abstract methods of `java.util.AbstractList`, `java.util.AbstractSet` and `java.util.AbstractMap`
 - the `Arrays` class uses it for sorting, `JFrame` uses `update()` as a template method,
 - subclasses of the `JFrame` use `paint(Graphics g)` as their hook method

Often used in frameworks

- this pattern shows up so often because it is a great design tool for creating frameworks
 - the framework controls how something gets done
 - leaves the person using the framework to specify your own details about what is actually happening at each step of the framework's algorithm
- consider an application framework that provides Application and Document classes
 - the Application class is responsible for opening existing documents stored in an external format, such as a file
 - a Document object represents the information in a document once it is read from the file
- applications built with the framework can subclass Application and Document to suit specific needs
 - a drawing application defines Draw Application and DrawDocument subclasses
 - a spreadsheet application defines Spreadsheet-Application and SpreadsheetDocument subclasses

When to use this pattern?

- when you need to support multiple algorithms that behave conceptually the same but have different implementations for each of their steps
- when you want to avoid code duplication in an algorithm
 - implement variations of the algorithm in subclasses
 - a good example of “refactoring to generalize”
 - first identify the differences in the existing code and then separate the differences into new operations
 - you then replace the differing code with a template method that calls one of these new operations
- when you want to control at what points sub classing is allowed
 - define a template method that calls “hook” operations at specific points, thereby permitting extensions only at those points
- when behavior of an algorithm can vary, you let subclasses implement the behavior through overriding

Summary

- template method should consists of certain steps whose order is fixed
- for some of the methods, implementation differs from base class to subclass
- the template method gives us an important technique for code reuse
- the strategy and template Method Patterns both encapsulate algorithms
 - one by inheritance and one by composition
- the Factory Method is a specialization of Template Method

Implementing the template method

Jason Fedin

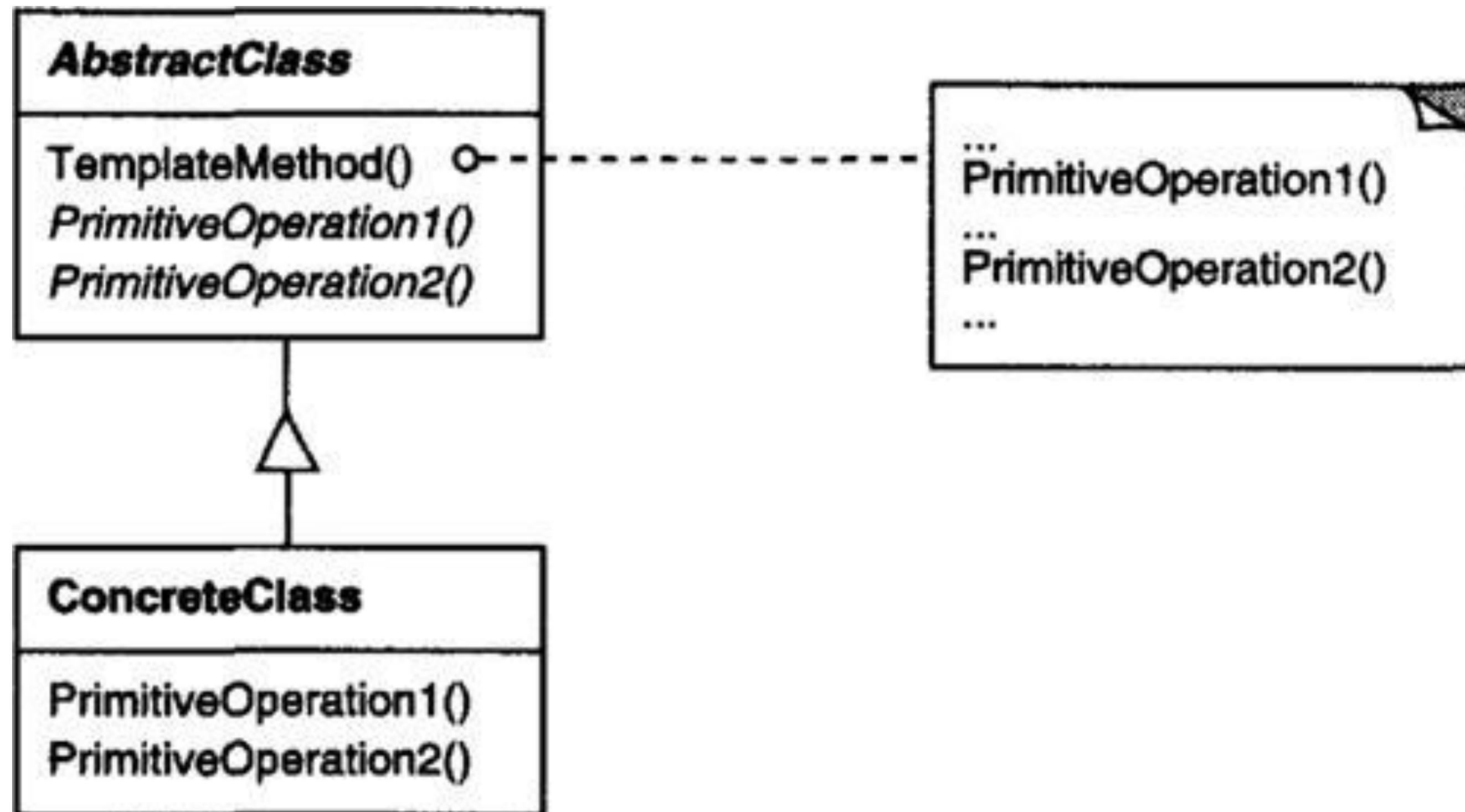
Implementation Overview

- the Template Method Pattern is a very common pattern
 - there are many implementations of the template methods that do not quite look like the textbook design of the pattern
 - it will take practice to identify when and how to use this pattern
- most of the time, subclasses call methods in the super class
 - in the template pattern, the superclass template method calls methods (abstract in base class) in the subclasses
 - known as the Hollywood Principle – “don’t call us, we’ll call you.”
- the Hollywood Principle guides us to put decision making in high-level modules that can decide how and when to call low-level modules

Implementation Overview

- the main template method is in the base class which is an abstract class
 - the abstract class may define concrete methods, abstract methods, and hooks
 - abstract methods are implemented by the subclasses
- a hook is a method that is declared in the abstract class
 - given an empty or default implementation
 - gives subclasses the ability to “hook into” the algorithm at various points
 - a subclass is also free to ignore the hook

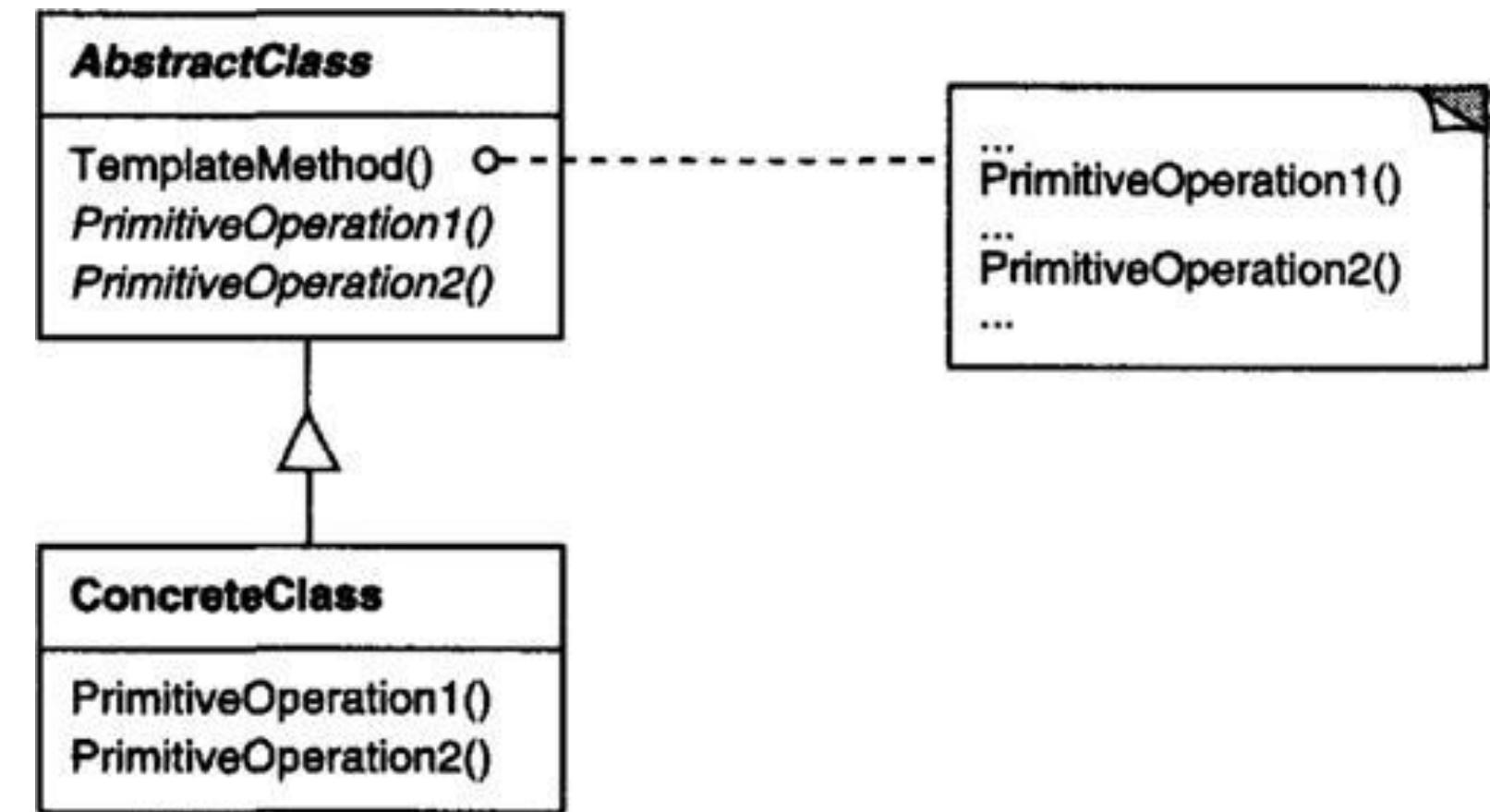
Class Diagram



Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Participants

- **AbstractClass**
 - contains the template method and abstract versions of the operations used in the template method
 - the template method defines the skeleton of an algorithm
 - the template method makes use of the primitive operations to implement an algorithm
 - calls primitive operations as well as operations defined in the AbstractClass or those of other objects
 - decoupled from the actual implementation of these operations
- **ConcreteClass**
 - implements the abstract operations, which are called when the templateMethod() needs them
 - may be many concrete classes, each implementing the full set of operations required by the template method



Here we have our abstract class; it is declared abstract and meant to be subclassed by classes that provide implementations of the operations.



```
abstract class AbstractClass {  
  
    final void templateMethod() {  
        primitiveOperation1();  
        primitiveOperation2();  
        concreteOperation();  
    }  
  
}
```

Here's the template method. It's declared final to prevent subclasses from reworking the sequence of steps in the algorithm.



The template method defines the sequence of steps, each represented by a method.

```
    abstract void primitiveOperation1();  
  
    abstract void primitiveOperation2();  
  
    void concreteOperation() {  
        // implementation here  
    }  
  
}
```

In this example, two of the primitive operations must be implemented by concrete subclasses.

We also have a concrete operation defined in the abstract class. More about these kinds of methods in a bit...

Implementation issues

- the primitive operations that a template method calls can be declared as protected methods
 - ensures that they are only called by the template method
- the template method itself should not be overridden
 - to prevent subclasses from changing the algorithm, declare the template method as final
- an important goal in designing template methods is to minimize the number of abstract operations used to flesh out the algorithm
 - we do not have a choice, overriding is a must in this case
 - to reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding
 - the more operations that need overriding, the more tedious things get for clients

Summary

- templates provide a good platform for code reuse
 - are also helpful in ensuring the required steps are implemented
- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm
- when broken down, there are four different types of methods used in the parent/abstract class
 - **concrete methods**
 - standard complete methods that are useful to the subclasses (usually utility methods)
 - **abstract methods**
 - methods containing no implementation that must be implemented in subclasses
 - **hook methods** (not shown on the previous slide)
 - methods containing a default implementation that may be overridden in some classes
 - often does nothing by default
 - **template method**
 - a method that calls any of the methods listed above in order to describe the algorithm without needing to implement the details (loose coupling)
 - should be made final so that it cannot be overridden

Implementing the template method

Jason Fedin

Example in intellij

- suppose we want to provide an algorithm to build a house
 - the steps need to be performed to build a house are building foundation, building pillars, building walls and windows
 - the important point is that we can not change the order of execution
 - we cannot build windows before building the foundation
- so in this case we can create a template method that will use different methods to build the house
 - building the foundation for a house is same for all type of houses, whether its a wooden house or a glass house
 - we can provide a base implementation for this, if subclasses want to override this method, they can but mostly it is common for all the types of houses

Create the abstract class

- contains the template method and abstract versions of the operations used in the template method
 - the template method defines the skeleton of an algorithm
- the template method makes use of the primitive operations to implement an algorithm
 - calls primitive operations as well as operations defined in the AbstractClass or those of other objects
 - decoupled from the actual implementation of these operations
- `buildHouse()` is the template method and defines the order of execution for performing several steps

```
public abstract class HouseTemplate {  
    //template method, final so subclasses can't override  
    public final void buildHouse(){  
        buildFoundation();  
        buildPillars();  
        buildWalls();  
        buildWindows();  
        System.out.println("House is built.");  
    }  
}
```

Create the abstract class

```
//default implementation, hook method  
private void buildWindows() {  
    System.out.println("Building Glass Windows");  
}  
  
//methods to be implemented by subclasses  
public abstract void buildWalls();  
public abstract void buildPillars();  
  
private void buildFoundation() {  
    System.out.println("Building foundation with cement, iron rods and sand");  
}  
}
```

Create the concrete classes

- implements the abstract operations, which are called when the templateMethod() needs them
- may be many concrete classes, each implementing the full set of operations required by the template method
- we can have different type of houses, such as Wooden House and Glass House

```
public class WoodenHouse extends HouseTemplate {  
  
    @Override  
    public void buildWalls() {  
        System.out.println("Building Wooden Walls");  
    }  
  
    @Override  
    public void buildPillars() {  
        System.out.println("Building Pillars with Wood coating");  
    }  
}
```

Create the concrete classes

```
public class GlassHouse extends HouseTemplate {
```

```
    @Override  
    public void buildWalls() {  
        System.out.println("Building Glass Walls");  
    }
```

```
    @Override  
    public void buildPillars() {  
        System.out.println("Building Pillars with glass coating");  
    }
```

```
}
```

Create the client

```
public class HousingClient {  
    public static void main(String[] args) {  
        HouseTemplate houseType = new WoodenHouse();  
  
        //using template method  
        houseType.buildHouse();  
        System.out.println("*****");  
  
        houseType = new GlassHouse();  
        houseType.buildHouse();  
    }  
  
}
```

- notice that the client is invoking the template method of base class and depending of implementation of different steps, it is using some of the methods from base class and some of them from subclass

Output

Building foundation with cement, iron rods and sand

Building Pillars with Wood coating

Building Wooden Walls

Building Glass Windows

House is built.

Building foundation with cement, iron rods and sand

Building Pillars with glass coating

Building Glass Walls

Building Glass Windows

House is built.

Demonstration

Jason Fedin

Create the abstract class

```
abstract class OrderProcessTemplate {  
    public boolean isGift;  
  
    public abstract void doSelect();  
  
    public abstract void doPayment();  
  
    public final void giftWrap() {  
        try {  
            System.out.println("Gift wrap successfull");  
        }  
        catch (Exception e) {  
            System.out.println("Gift wrap unsuccessful");  
        }  
    }  
}
```

Create the abstract class

```
public abstract void doDelivery();

// the actual template method
public final void processOrder(boolean isGift) {
    doSelect();
    doPayment();
    if (isGift) {
        giftWrap();
    }
    doDelivery();
}
}
```

Create the concrete classes for ordering

```
class NetOrder extends OrderProcessTemplate {  
    @Override  
    public void doSelect() {  
        System.out.println("Item added to online shopping cart");  
        System.out.println("Get gift wrap preference");  
        System.out.println("Get delivery address.");  
    }  
  
    @Override  
    public void doPayment() {  
        System.out.println("Online Payment through Netbanking, card or Pay pal");  
    }  
  
    @Override  
    public void doDelivery(){  
        System.out.println("Ship the item through post office to delivery address");  
    }  
}
```

Create the concrete classes for ordering

```
class StoreOrder extends OrderProcessTemplate {  
  
    @Override  
    public void doSelect() {  
        System.out.println("Customer chooses the item from shelf.");  
    }  
  
    @Override  
    public void doPayment() {  
        System.out.println("Pays at counter through cash/POS");  
    }  
  
    @Override  
    public void doDelivery() {  
        System.out.println("Item delivered to in delivery counter.");  
    }  
}
```

Create the client

```
class TemplateMethodPatternClient
{
    public static void main(String[] args)
    {
        OrderProcessTemplate netOrder = new NetOrder();
        netOrder.processOrder(true);
        System.out.println();
        OrderProcessTemplate storeOrder = new StoreOrder();
        storeOrder.processOrder(true);
    }
}
```

Output

Item added to online shopping cart

Get gift wrap preference

Get delivery address.

Online Payment through Netbanking, card or Paypal

Gift wrap successful

Ship the item through post to delivery address

Customer chooses the item from shelf.

Pays at counter through cash/POS

Gift wrap successful

Item delivered to in delivery counter.

The Visitor Design Pattern

Jason Fedin

Overview

- the visitor design pattern represents an operation (method) to be performed on the elements of an object structure (collection, list, etc.)
 - lets you define a new operation without changing the classes of the elements on which it operates
- helps us to add new functionalities to an existing object structure in such a way that the old structure remains unaffected by these changes
 - we can follow the open/close principle here

Examples

- consider a taxi booking scenario
 - taxi arrives at the defined location for the pickup
 - once we enter into it, the visiting taxi takes control of the transportation
 - it can choose a different way toward our destination and we may or may not have any prior knowledge of that way
- a shopping cart where we can add different type of items (Elements)
 - when we click on checkout button, it calculates the total amount to be paid
 - we can move out the calculation logic from an item class to another class using the visitor pattern (less coupling)
- this pattern is very useful when plugging into public APIs
 - clients can perform operations on a class with a visiting class without modifying the source

More Overview

- allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure
- used when we have to perform an operation on a group of similar kinds of objects
 - we can move the operational logic from the objects to another class
- useful for adding new operations without affecting the existing structure

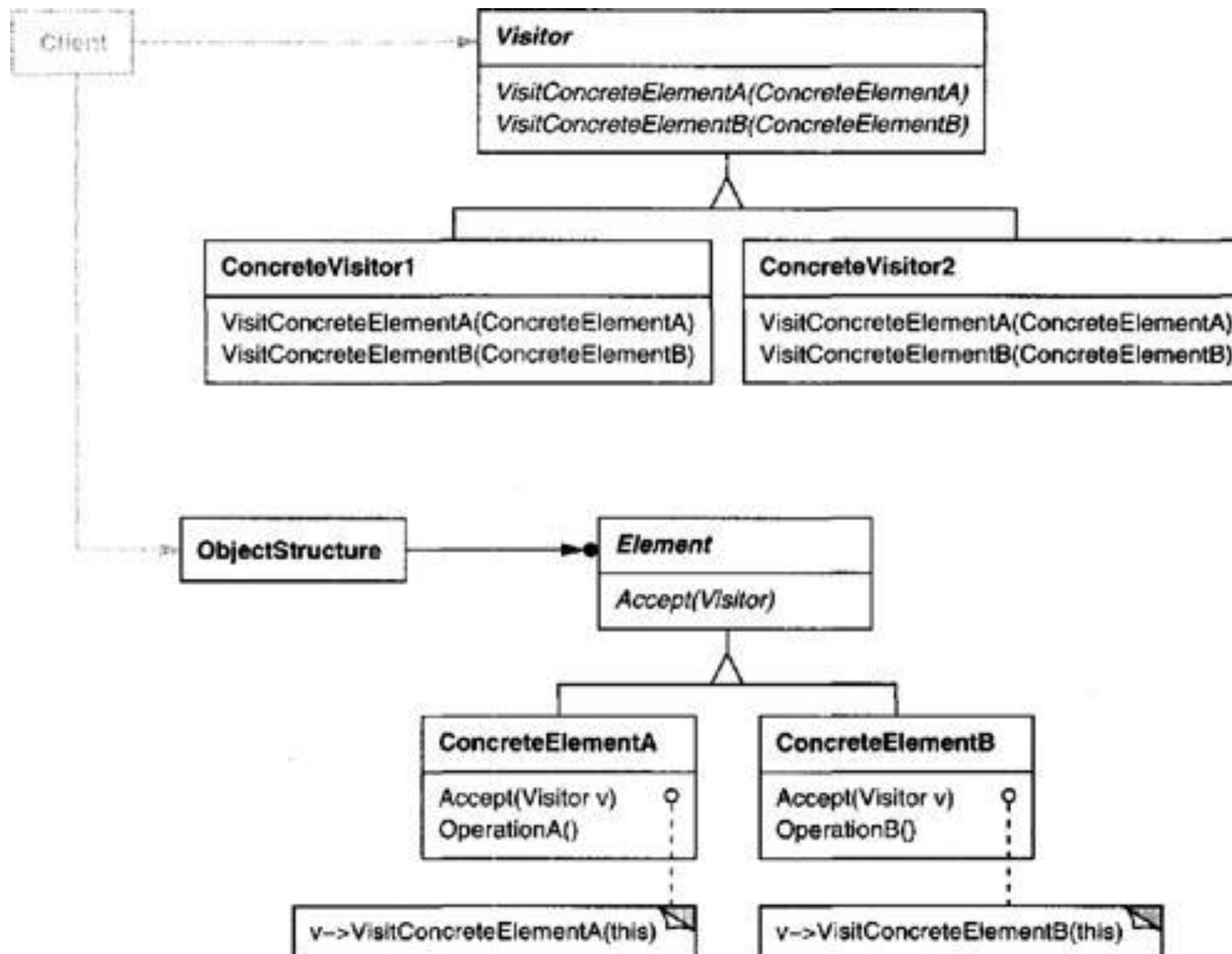
When to use the visitor pattern

- when an object structure (collection or list) contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes
 - visitor lets you keep related operations together by defining them in one class
 - encourages cleaner code
- when you want to decouple some logical code from the elements that you are using as input
- when you want to add capabilities to a composite of objects and encapsulation is not important
- when the object structure is shared by many applications, use the visitor pattern to put operations in just those applications that need them

Implementing the Visitor Pattern

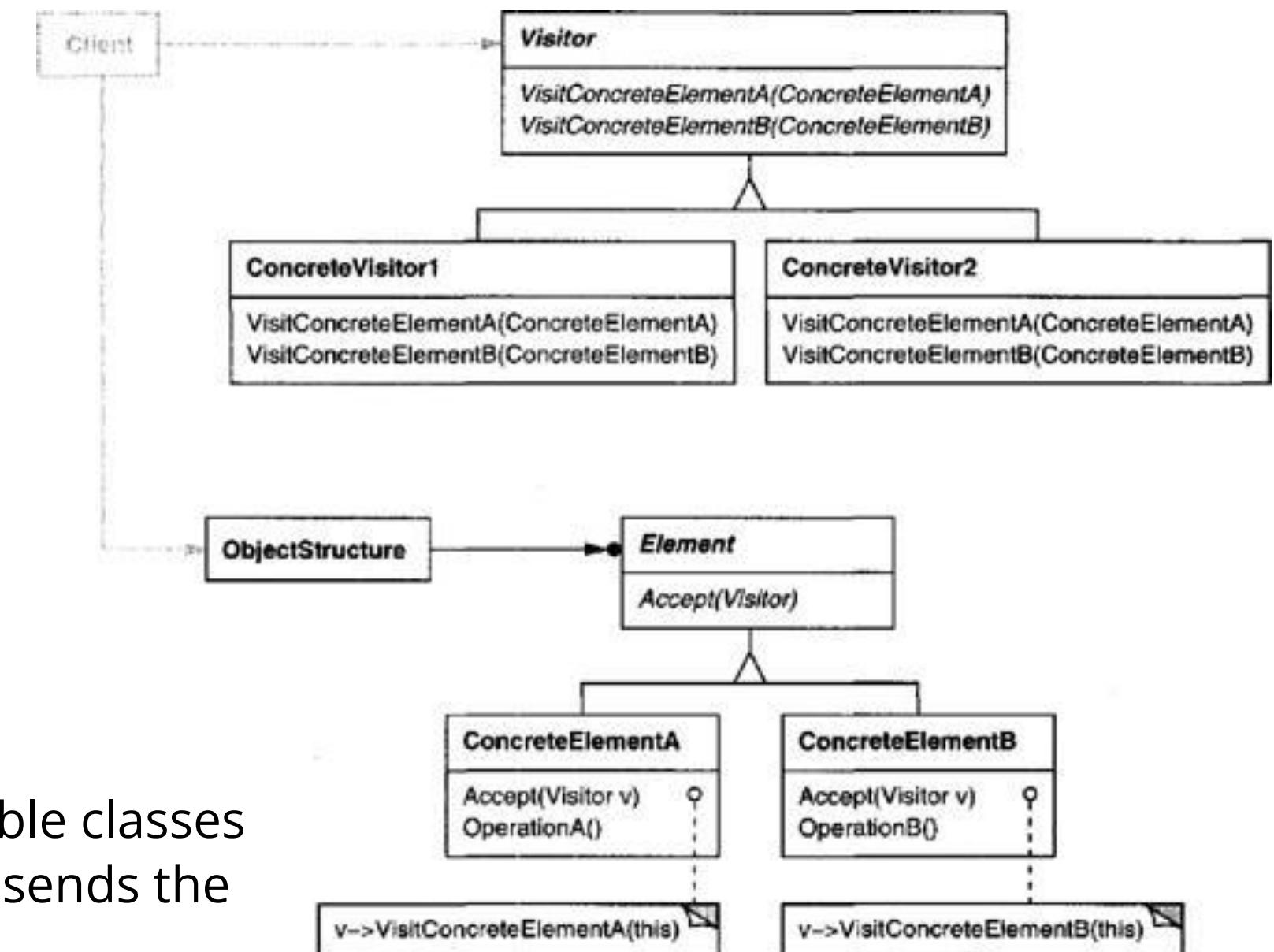
Jason Fedin

Overview



Design Patterns: Elements of Reusable Object-Oriented Software by: Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

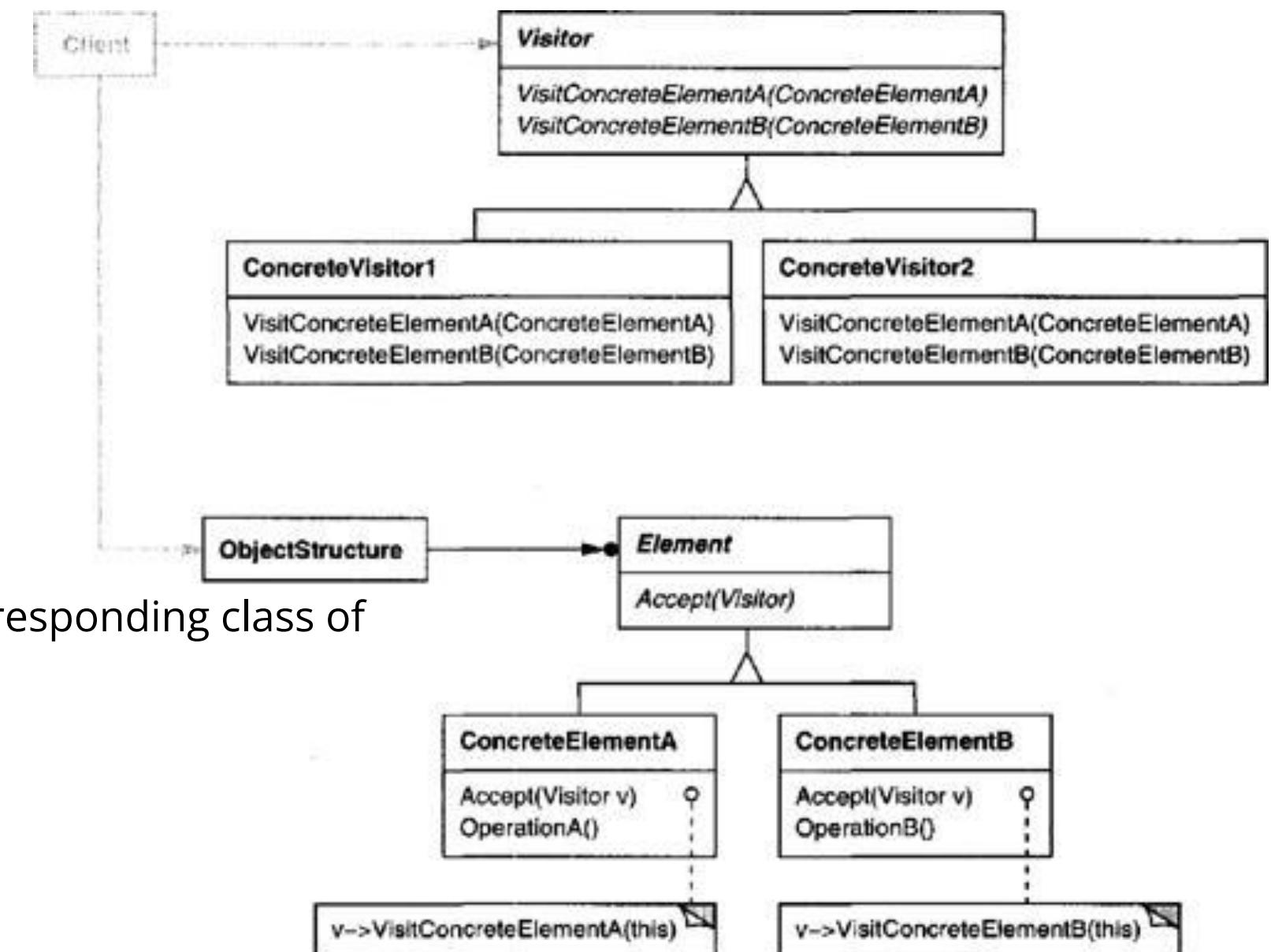
Participants



- **Visitor**
 - used to declare the visit operations for all the types of visitable classes
 - the operation's name and signature identifies the class that sends the Visit request to the visitor
 - lets the visitor determine the concrete class of the element being visited
 - the visitor can then access the element directly through its particular interface

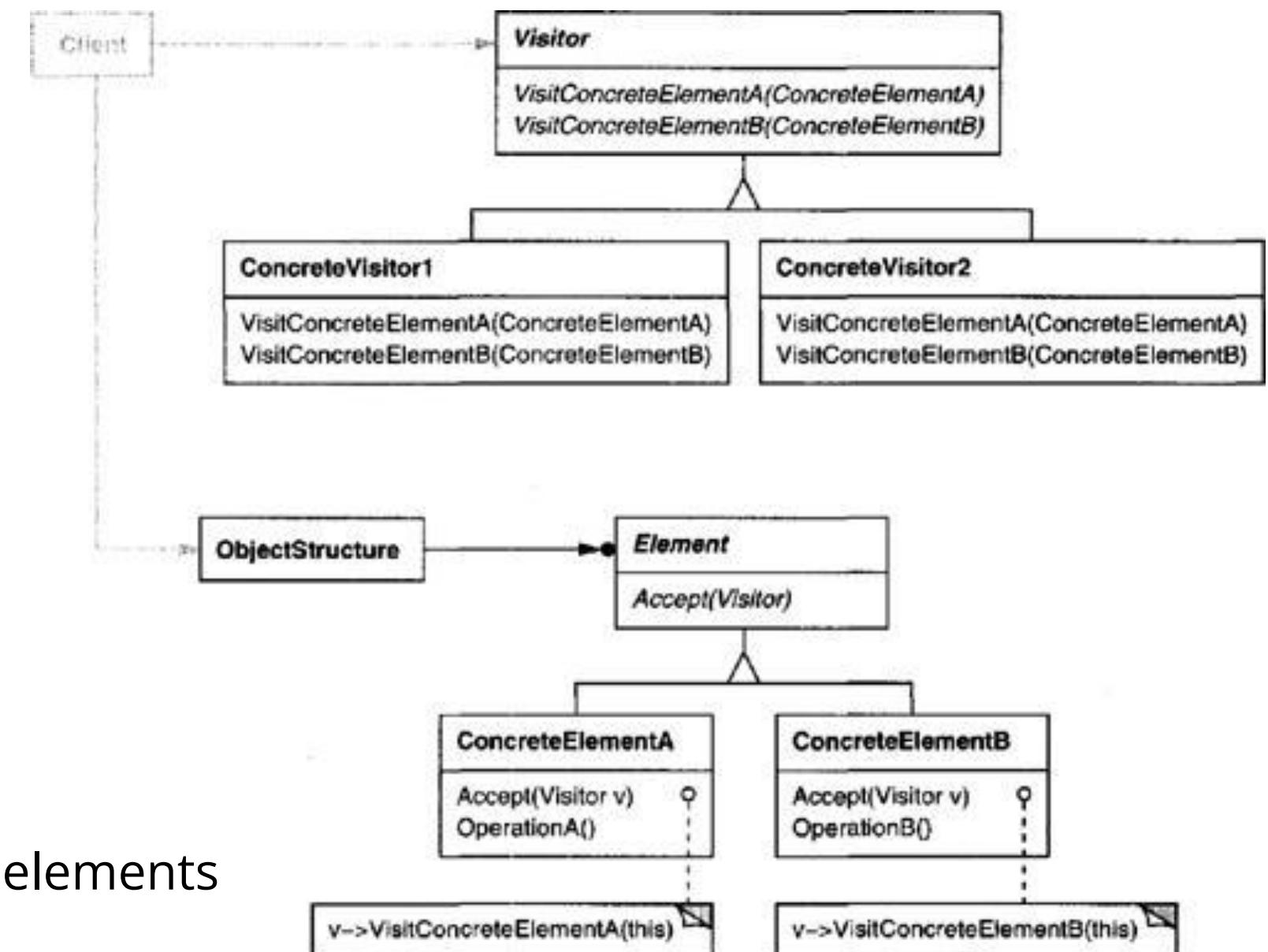
Participants

- Concrete Visitor
 - implements each method declared by Visitor
 - each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure
 - provides the context for the algorithm and stores its local state
 - state often accumulates results during the traversal of the structure
- Element
 - defines an Accept method that takes a visitor as an argument
 - the entry point which enables an object to be “visited” by the visitor object
- ConcreteElement
 - implements an Accept method that takes a visitor as an argument
 - the visitor object is passed to this object using the accept method

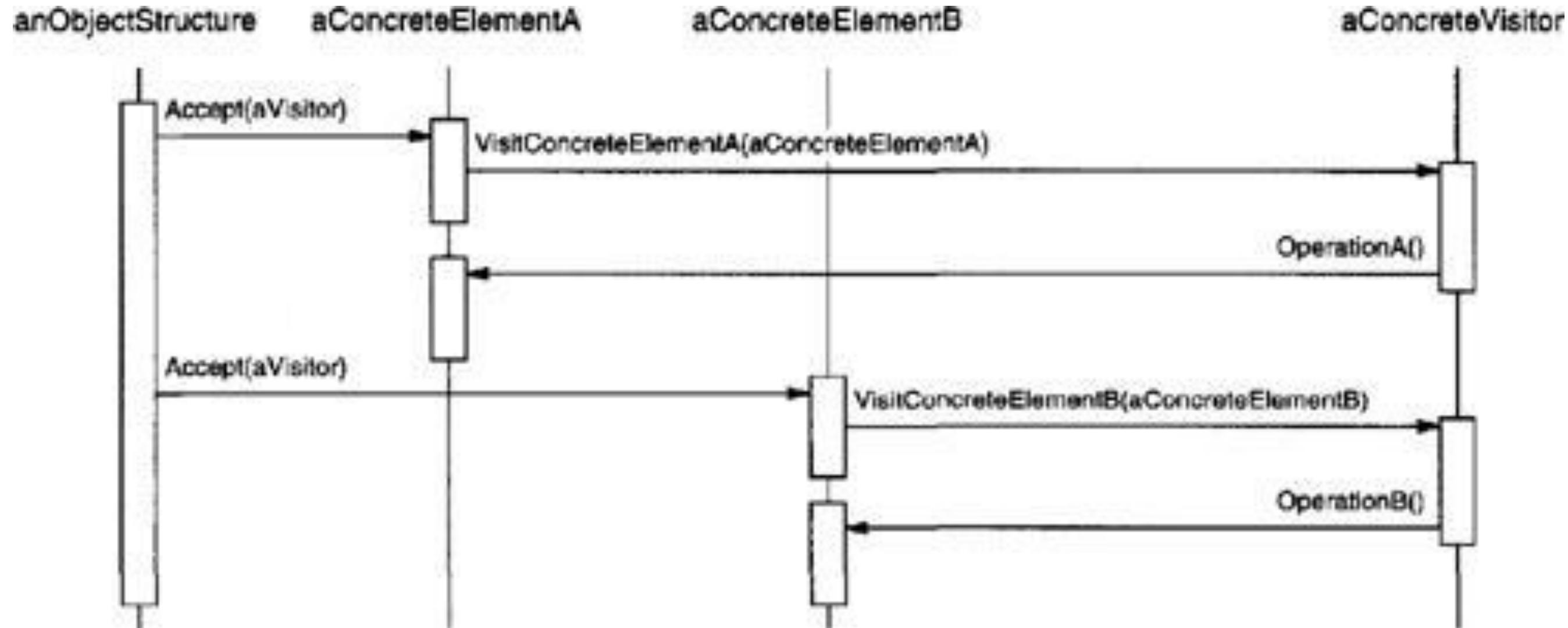


Participants

- ObjectStructure
 - can enumerate its elements
 - may provide a high-level interface to allow the visitor to visit its elements
 - may either be a composite or a collection such as a list or a set
- Client
 - a consumer of the classes of the visitor design pattern
 - has access to the data structure objects and can instruct them to accept a Visitor to perform the appropriate processing



Sequence Diagram



- a client creates a `ConcreteVisitor` object and then traverses the object structure (composite or collection), visiting each element with the visitor
- when an element is visited, it calls the `Visitor` operation that corresponds to its class
 - the element supplies itself as an argument to this operation to let the visitor access its state

Summary

- the Visitor pattern will create an external class that uses data in other classes
- the core of this pattern is the Visitor interface
 - defines a visit operation for each type of ConcreteElement in the object structure (composite or collection)
- the ConcreteVisitor implements the operations defined in the Visitor interface
 - will store local state, typically as it traverses the set of elements
- the element interface simply defines an accept method to allow the visitor to run some action over that element
 - the ConcreteElement will implement this accept method
- Visitor operations are controlled in a unified manner

Summary

- allows you to add operations to a Composite structure (or collection) without changing the structure itself
- very easy to add operations that depend on the components of complex objects
 - you can define a new operation over an object structure simply by adding a new visitor
- a visitor gathers related operations and separates unrelated ones
 - localized in a visitor
 - unrelated sets of behavior are partitioned in their own visitor subclasses
 - simplifies both the classes defining the elements and the algorithms defined in the visitors

Summary

- class encapsulation may need to be compromised when visitors are used
 - if the existing structure is really complex, the traversal mechanism becomes complex
- a drawback of visitor pattern is that we should know the return type of visit() methods at the time of designing
 - otherwise we will have to change the interface and all of its implementations
- another drawback is that if there are too many implementations of the visitor interface, it makes it hard to extend

Implementing the Visitor Pattern

Jason Fedin

Example in intellij

- lets go through an example of a Shopping cart where we can add different types of items (Elements)
- when we click on checkout button, it calculates the total amount to be paid
- we can have the calculation logic in item classes or we can move out this logic to another class using the visitor pattern (much better, less coupling)
- First, we will create different type of items (Elements) to be used in shopping cart (make up the object structure)

Create the Element interface

- defines an Accept operation that takes a visitor as an argument
 - the entry point which enables an object to be “visited” by the visitor object
- Notice that accept method takes Visitor argument
 - we can have some other methods also specific for items if we wanted to

```
public interface ItemElement {  
    public int accept(ShoppingCartVisitor visitor);  
}
```

Create the Concrete Element (Book)

- implements an Accept operation that takes a visitor as an argument
 - the visitor object is passed to this object using the accept operation

```
public class Book implements ItemElement {  
    private int price;  
    private String isbnNumber;  
  
    public Book(int cost, String isbn) {  
        this.price=cost;  
        this.isbnNumber=isbn;  
    }  
}
```

Create the Concrete Element (Book)

```
public int getPrice() {  
    return price;  
}
```

```
public String getIsbnNumber() {  
    return isbnNumber;  
}
```

```
@Override  
public int accept(ShoppingCartVisitor visitor) {  
    return visitor.visit(this);  
}
```

Create concrete element (Fruit)

```
public class Fruit implements ItemElement {
```

```
    private int pricePerKg;
```

```
    private int weight;
```

```
    private String name;
```

```
    public Fruit(int priceKg, int wt, String nm){
```

```
        this.pricePerKg=priceKg;
```

```
        this.weight=wt;
```

```
        this.name = nm;
```

```
}
```

```
    public int getPricePerKg() {
```

```
        return pricePerKg;
```

```
}
```

Create the concrete element (Fruit)

```
public int getWeight() {  
    return weight;  
}  
  
public String getName(){  
    return this.name;  
}  
  
@Override  
public int accept(ShoppingCartVisitor visitor) {  
    return visitor.visit(this);  
}  
}
```

- Notice the implementation of accept() method in concrete classes, its calling the visit() method of Visitor and passing itself as argument
- Notice that the implementation of the accept() method in all the items are same but it can be different, for example, there can be logic to check if item is free then don't call the visit() method at all
- we have the visit() method for different type of items in Visitor interface that will be implemented by concrete visitor class

Create the Visitor

- used to declare the visit operations for all the types of visitable classes
- the operation's name and signature identifies the class that sends the Visit request to the visitor
 - lets the visitor determine the concrete class of the element being visited
the visitor can then access the element directly through its particular interface

```
public interface ShoppingCartVisitor {  
    int visit(Book book);  
    int visit(Fruit fruit);  
}
```

- now we will implement visitor interface and every item will have its own logic to calculate the cost

Create the concrete visitor

- implements each operation declared by Visitor
- each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure
- provides the context for the algorithm and stores its local state
 - state often accumulates results during the traversal of the structure

```
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {  
  
    @Override  
    public int visit(Book book) {  
        int cost=0;  
        //apply 5$ discount if book price is greater than 50  
        if(book.getPrice() > 50){  
            cost = book.getPrice()-5;  
        }else cost = book.getPrice();  
        System.out.println("Book ISBN::"+book.getISBNNumber() + " cost =" +cost);  
        return cost;  
    }  
}
```

Create the concrete visitor

```
@Override  
public int visit(Fruit fruit) {  
    int cost = fruit.getPricePerKg()*fruit.getWeight();  
    System.out.println(fruit.getName() + " cost = "+cost);  
    return cost;  
}  
  
}
```

Create the client

- has access to the data structure objects and can instruct them to accept a Visitor to perform the appropriate processing
- a client creates a ConcreteVisitor object and then traverses the object structure, visiting each element with the visitor
- when an element is visited, it calls the Visitor operation that corresponds to its class
 - the element supplies itself as an argument to this operation to let the visitor access its state

```
public class ShoppingCartClient {  
  
    public static void main(String[] args) {  
        ItemElement[] items = new ItemElement[]{new Book(20, "1234"), new Book(100, "5678"),  
            new Fruit(10, 2, "Banana"), new Fruit(5, 5, "Apple")};  
  
        int total = calculatePrice(items);  
        System.out.println("Total Cost = "+total);  
    }  
}
```

Create the client

```
private static int calculatePrice(ItemElement[] items) {  
    ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();  
    int sum=0;  
    for(ItemElement item : items){  
        sum = sum + item.accept(visitor);  
    }  
    return sum;  
}  
  
}
```

Output

Book ISBN::1234 cost =20

Book ISBN::5678 cost =95

Banana cost = 20

Apple cost = 25

Total Cost = 160

Demonstration

Jason Fedin

Create the Element Interface (visitable interface)

```
import java.util.*;
```

```
interface Visitable{  
    public void accept(Visitor visitor);  
}
```

Create the concrete elements

```
class Book implements Visitable{  
    private double price;  
    private double weight;  
  
    public Book(double price, double weight) {  
        this.price = price;  
        this.weight = weight;  
    }  
  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
    public double getPrice() {  
        return price;  
    }  
    public double getWeight() {  
        return weight;  
    }  
}
```

Create other concrete elements (CD and DVD)

```
class CD implements Visitable {  
    private double price;  
    private double weight;  
  
    public CD(double price, double weight) {  
        this.price = price;  
        this.weight = weight;  
    }  
  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
    public double getPrice() {  
        return price;  
    }  
    public double getWeight() {  
        return weight;  
    }  
}
```

Create other concrete elements (CD and DVD)

```
class DVD implements Visitable{  
    private double price;  
    private double weight;  
  
    public DVD(double price, double weight) {  
        this.price = price;  
        this.weight = weight;  
    }  
  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
    public double getPrice() {  
        return price;  
    }  
    public double getWeight() {  
        return weight;  
    }  
}
```

Create the visitor interface

```
interface Visitor{  
    public void visit(Book book);  
    public void visit(CD cd);  
    public void visit(DVD dvd);  
}
```

Create the concrete visitor (US)

```
class USPostageVisitor implements Visitor {  
    private double totalPostageForCart;  
  
    //collect data about the book  
    public void visit(Book book) {  
        //assume we have a calculation here related to weight and price free postage for a book over 10  
        if(book.getPrice() < 20.0) {  
            totalPostageForCart += book.getWeight() * 2;  
        }  
    }  
    //add other visitors here  
    public void visit(CD cd) {  
        if(cd.getPrice() < 20.0) {  
            totalPostageForCart += cd.getWeight() * 2.5;  
        }  
    }  
}
```

Create the concrete visitor (US)

```
public void visit(DVD dvd) {  
    if(dvd.getPrice() < 20.0) {  
        totalPostageForCart += dvd.getWeight() * 3;  
    }  
}  
  
//return the internal state  
public double getTotalPostage() {  
    return totalPostageForCart;  
}
```

Create the concrete visitor (SouthAmerica)

```
class SouthAmericaPostageVisitor implements Visitor {  
    private double totalPostageForCart;  
  
    //collect data about the book  
    public void visit(Book book) {  
        //assume we have a calculation here related to weight and price free postage for a book over 10  
        if(book.getPrice() < 30.0) {  
            totalPostageForCart += (book.getWeight() * 2) * 2;  
        }  
    }  
    //add other visitors here  
    public void visit(CD cd) {  
        if(cd.getPrice() < 30.0) {  
            totalPostageForCart += (cd.getWeight() * 2.5) * 2;  
        }  
    }  
}
```

Create the concrete visitor (South America)

```
public void visit(DVD dvd) {  
    if(dvd.getPrice() < 30.0) {  
        totalPostageForCart += (dvd.getWeight() * 3) * 2;  
    }  
}  
  
//return the internal state  
public double getTotalPostage() {  
    return totalPostageForCart;  
}
```

Create the client

- as you can see it's a simple formula, but the point is that all the calculation for all regional postage is done in one central place
 - to drive this visitor, we'll need a way of iterating through our shopping cart, as follows:

```
public class ShoppingCart {  
    //normal shopping cart stuff  
    private static ArrayList<Visitable> items = new ArrayList<Visitable>();  
    public static double calculatePostage(Visitor visitor) {  
        //iterate through all items  
        for(Visitable item: items) {  
            item.accept(visitor);  
        }  
        double postage = visitor.getTotalPostage();  
        return postage;  
    }  
}
```

Create the client

```
public static void main(String[] args) {  
    // create a bunch of visitors (Book, CD, and DVD)  
    Visitable myBook = new Book(8.52, 1.05);  
    Visitable myCD = new CD(18.52, 3.05);  
    Visitable myDVD = new DVD(6.53, 4.02);  
  
    // add each vistor to the array list  
    items.add(myBook);  
    items.add(myCD);  
    items.add(myDVD);  
  
    // call calculatePostage  
    Visitor visitor = new USPostageVisitor();  
    double myPostage = calculatePostage(visitor);  
    System.out.println("The total postage for my items is: " + myPostage);  
    visitor = new SouthAmericaPostageVisitor();  
    myPostage = calculatePostage(visitor);  
    System.out.println("The total postage for my items is: " + myPostage);  
}  
}
```

Output Example

The total postage for my items is: 10.14

The total postage for my items is: 20.28

- if we had other types of item here, once the visitor implements a method to visit that item, we could easily calculate the total postage
- while the Visitor may seem a bit strange at first, you can see how much it helps to clean up your code
 - the whole point of this pattern - to allow you separate out certain logic from the elements themselves, keeping your data classes simple

Summary of Behavioral Patterns

Jason Fedin

Encapsulating Variation

- encapsulating variation is a theme of many behavioral patterns
- when an aspect of a program changes frequently, these patterns define an object that encapsulates that aspect
 - then other parts of the program can collaborate with the object whenever they depend on that aspect
- this theme runs through other kinds of patterns
- Abstract Factory, Builder, and Prototype all encapsulate knowledge about how objects are created
- Decorator encapsulates responsibility that can be added to an object
- Bridge separates an abstraction from its implementation, letting them vary independently

Encapsulating Variation (cont'd)

- a Strategy object encapsulates an algorithm
- a State object encapsulates a state-dependent behavior
- a Mediator object encapsulates the protocol between objects
- an Iterator object encapsulates the way you access and traverse the components of an aggregate object
- not all object behavioral patterns work like this
 - Chain of Responsibility deals with an arbitrary number of objects, all of which may already exist in the system

Objects as Arguments

- several patterns introduce an object that is always used as an argument
- a visitor object is the argument to a polymorphic accept operation on the objects it visits
 - visitor is never considered a part of those objects
- other patterns define objects that act as magic tokens to be passed around and invoked at a later time
 - in the command pattern, the token represents a request
 - in the Memento, it represents the internal state of an object at a particular time
 - in both cases, the token can have a complex internal representation, but the client is never aware of it
- polymorphism is important in the Command pattern
 - executing the Command object is a polymorphic operation
- the Memento interface is so narrow that a memento can only be passed as a value
 - it is likely to present no polymorphic operations at all to its clients

Mediator vs. Observer

- Mediator and Observer are competing patterns
 - Observer distributes communication by introducing Observer and Subject objects
 - a Mediator object encapsulates the communication between other objects
- in the Observer pattern, there is no single object that encapsulates a constraint
 - the Observer and the Subject must cooperate to maintain the constraint
 - a single subject usually has many observers
 - sometimes the observer of one subject is a subject of another observer.
- the Mediator pattern centralizes rather than distributes
 - It places the responsibility for maintaining a constraint squarely in the mediator
- easier to make reusable Observers and Subjects than to make reusable Mediators
- on the other hand, it is easier to understand the flow of communication in Mediator than in Observer
 - observers and subjects are usually connected shortly after they have been created
 - hard to see how they are connected later in the program
 - the indirection that Observer introduces will still make a system harder to understand

Decoupling Senders and Receivers

- when collaborating objects refer to each other directly, they become dependent on each other
 - can have an adverse impact on the layering and reusability of a system
- Command, Observer, Mediator, and Chain of Responsibility address how you can decouple senders and receivers, but with different trade-offs
- for example, the Command pattern supports decoupling by using a Command object to define the binding between a sender and receiver

Behavioral Patterns working together

- behavioral design patterns complement and reinforce each other
- a class in a chain of responsibility will probably include at least one application of Template Method
 - the template method can use primitive operations to determine whether the object should handle the request and to choose the object to forward to
- a class in the chain of responsibility can use the Command pattern to represent requests as objects
- an interpreter can use the State pattern to define parsing contexts
- an iterator can traverse an aggregate, and a visitor can apply an operation to each element in the aggregate

Behavioral Patterns working with other patterns

- a system that uses the Composite pattern might use a visitor to perform operations on components of the composition
- a composite could use the Chain of Responsibility to let components access global properties through their parent
 - a composite could also use Decorator to override these properties on parts of the composition
- a composite could use the Observer pattern to tie one object structure to another and the State pattern to let a component change its behavior as its state changes
- the composition itself might be created using the approach in Builder
- well-designed object-oriented systems have multiple patterns embedded in them

MVC - Model View Controller

Jason Fedin

Overview

- there are some patterns that we did not cover that I would like to talk about at a high level
- some are used specifically with J2EE (web based enterprise applications)
- Lets take a look at the following:
 - MVC – Model View Controller (more of an architectural pattern)
 - Null object
- J2EE patterns
 - Business Delegate Pattern
 - Composite Entity pattern
 - Data Access Object
 - Front Controller
 - Intercepting Filter
 - Service Locator
 - Transfer Object

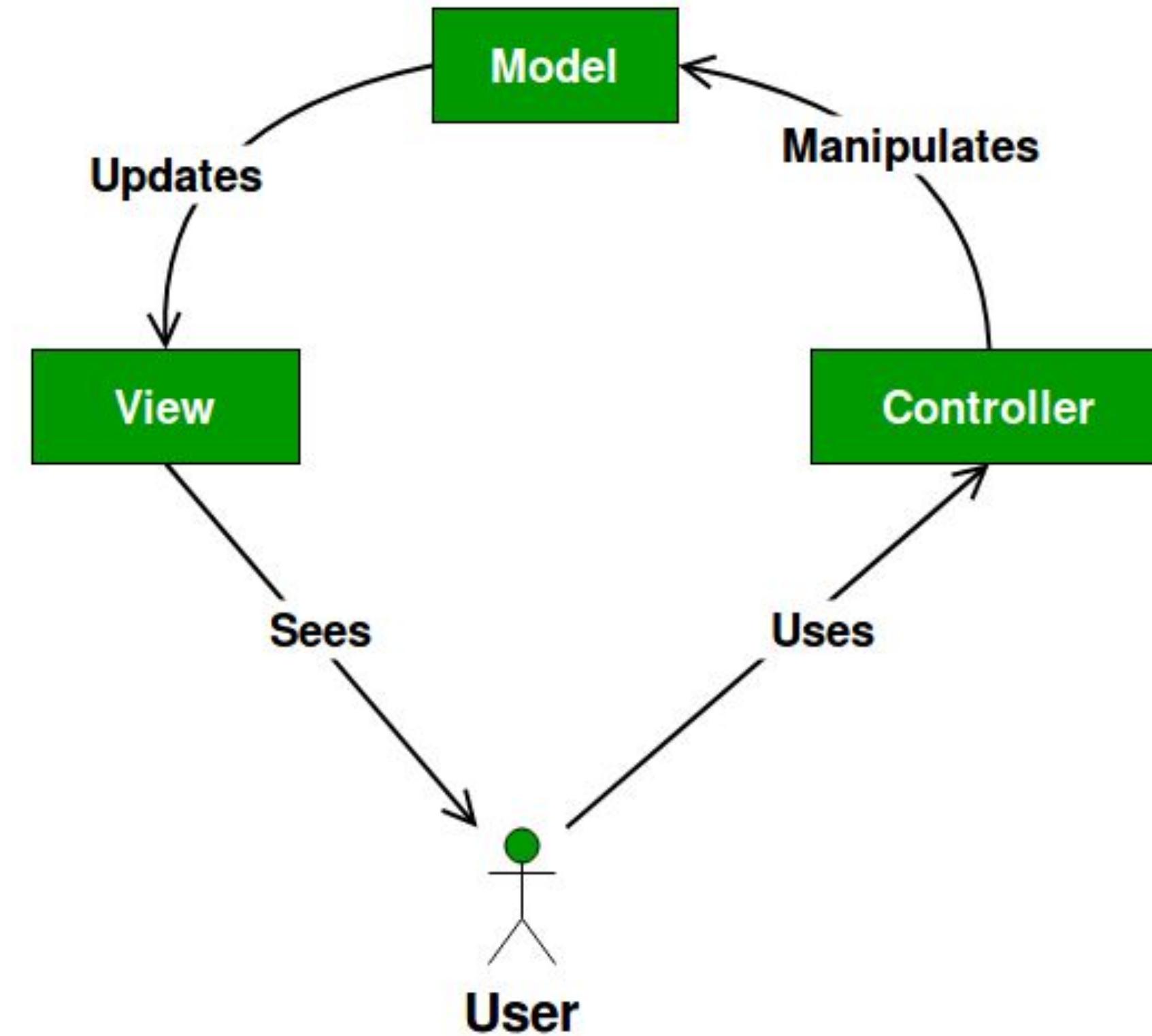
Model-View-Controller

- specifies that an application consist of a data model, presentation information, and control information
 - requires that each of these be separated into different objects
 - a popular way of organizing your code
- MVC is short for Model, View, and Controller
- MVC is more of an architectural pattern, but not for a complete application
- each section of your code's core functions has a purpose, and those purposes are different
 - some of your code holds the data of your app, some of your code makes your app look nice, and some of your code controls how your app functions
 - makes thinking about your app, revisiting your app, and sharing your app with others much easier and cleaner
 - Single responsibility principle and separation of concerns
- MVC is helpful when planning your app
 - it gives you an outline of how your ideas should be organized into actual code
- thinking about how code interacts with other code is a significant part of programming

Model-View-Controller

- the Model contains only the pure application data
 - contains no logic describing how to present the data to a user
- the View presents the model's data to the user
 - knows how to access the model's data
 - does not know what this data means or what the user can do to manipulate it
- the Controller exists between the view and the model
 - listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events
 - the reaction is usually to call a method on the model
 - the result of this action is then automatically reflected in the view

Illustration



Real-World Example

- think about what happens when you prepare a Thanksgiving dinner
- you have a fridge full of food, which represents the Model
 - the fridge (Model) contains the raw materials we will use to make dinner
- you also probably have a recipe or two
- a recipe is like the Controller of Thanksgiving dinner
 - dictates which stuff in the fridge you will take out
 - how you will put it together, and how long you need to cook it
- then, you have table-settings, silverware, etc.,
 - what your hungry friends and family use to eat dinner
- table-top items are like the View
 - let your guests interact with your Model and Controller's creation

Computer Example

- let's imagine you are creating a To-do list app
 - will let users create tasks and organize them into lists
- the Model in a todo app might define what what a "task" is and that a "list" is a collection of tasks
- the View code will define what the todos and lists looks like, visually
 - tasks could have large font, or be a certain color
- the Controller could define how a user adds a task, or marks another as complete
 - connects the View's add button to the Model
 - when you click "add task," the Model adds a new task

Advantages of MVC

- multiple developers can work simultaneously on the model, controller and views
- enables logical grouping of related actions on a controller together
 - views for a specific model are also grouped together
 - separation of concerns
- models can have multiple views
- faster development process
 - supports rapid and parallel development
- modification does not affect the entire model
 - model part does not depend on the views part
 - any changes in the Model will not affect the entire architecture

J2EE Patterns

Jason Fedin

Overview

- there are some patterns that we did not cover that I would like to talk about at a high level
- Lets take a look at the following:
 - Null object
- J2EE patterns
 - Business Delegate Pattern
 - Composite Entity pattern
 - Data Access Object
 - Front Controller
 - Intercepting Filter
 - Service Locator
 - Transfer Object

Null Object Pattern

- the intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior
 - a design where “nothing will come of nothing”
- in this pattern, we create an abstract class specifying various operations to be done, a concrete classes extending this class and a null object class providing do nothing implementation
 - used seamlessly where we need to check null value
- use the Null Object pattern when
 - an object requires a collaborator
 - makes use of a collaboration that already exists
 - some collaborator instances should do nothing
 - you want to abstract the handling of null away from the client
- the Null Object class is often implemented as a Singleton
 - a null object usually does not have any state, its state can't change, so multiple instances are identical

Business Delegate Pattern

- used to decouple the presentation tier and the business tier
 - used to reduce communication or remote lookup functionality to business tier code in presentation tier code
- acts as a client-side business abstraction
 - provides an abstraction for, and thus hides, the implementation of the business services
- Use the pattern when:
 - you want loose coupling between presentation and business tiers
 - you want to orchestrate calls to multiple business services
 - you want to encapsulate service lookups and service calls

Business Delegate Pattern

- Components include:
 - **Client**
 - presentation tier code may be JSP, servlet or UI java code
 - **Business Delegate**
 - a single entry point class for client entities to provide access to Business Service methods
 - **LookUp Service**
 - responsible to get relative business implementation and provide business object access to business delegate object
 - **Business Service**
 - concrete classes implement this business service to provide actual business implementation logic

Composite Entity Pattern

- a Composite entity is an EJB entity bean which represents a graph of objects
- when a composite entity is updated
 - internally dependent objects beans get updated automatically as being managed by EJB entity bean
- The following are the participants:
 - **Context**
 - entity beans are not intended to represent every persistent object in the object model
 - entity beans are better suited for coarse-grained persistent business objects
 - **Composite Entity**
 - primary entity bean
 - can be coarse grained or can contain a coarse grained object to be used for persistence purpose

Composite Entity Pattern

- The following are the participants:

- Coarse-Grained Object**

- contains dependent objects
 - has its own life cycle and also manages life cycle of dependent objects.

- Dependent Object**

- an object which depends on coarse grained object for its persistence lifecycle

- Strategies**

- represents how to implement a Composite Entity

Data Access Object Pattern

- many real-world J2EE applications need to use persistent data at some point
- access to data varies depending on the source of the data
- access to persistent storage varies greatly depending on the type of storage (relational databases, object-oriented databases, flat files, and so forth) and the vendor implementation
- the data may reside in mainframe systems, Lightweight Directory Access Protocol (LDAP) repositories, etc.
- this pattern is used to separate low level data accessing API or operations from high level business services
 - Data Access Object Interface
 - this interface defines the standard operations to be performed on a model object(s)
 - Data Access Object concrete class
 - implements above interface and is responsible to get data from a data source which can be database / xml or any other storage mechanism
 - Model Object or Value Object
 - simple POJO (Plain Old Java Object) containing get/set methods to store data retrieved using DAO class

Front Controller Pattern

- all requests that come for a resource in an application will be handled by a single handler and then dispatched to the appropriate handler for that type of request
- not used as widely since the MVC pattern was released
- the presentation-tier request handling mechanism must control and coordinate processing of each user across multiple requests
- when a user accesses the view directly without going through a centralized mechanism, two problems may occur
 - each view is required to provide its own system services, often resulting in duplicate code
 - view navigation is left to the views which may result in mixed view content and view navigation
- also, distributed control is more difficult to maintain, since changes will often need to be made in numerous places
- this pattern will ensure a system has a centralized access point for presentation-tier request handling to support the integration of system services, content retrieval, view management, and navigation

Front Controller Participants

- Controller
 - the initial contact point for handling all requests in the system
 - may delegate to a helper to complete authentication and authorization of a user or to initiate contact retrieval
- View
 - represents and displays information to the client
 - retrieves information from a model
 - helpers support views by encapsulating and adapting the underlying data model for use in the display
- Dispatcher
 - responsible for view management and navigation, managing the choice of the next view to present to the user, and providing the mechanism for vectoring control to this resource
- Helper
 - responsible for helping a view or controller complete its processing
 - have numerous responsibilities
 - gathering data required by the view and storing this intermediate model

Intercepting filter design pattern

- a JavaEE pattern which creates pluggable filters to process common services in a standard manner without requiring changes to core request processing code
- used when we want to do some pre-processing / post-processing with request or response of the application
 - filters are defined and applied on the request before passing the request to actual target application
 - filters can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers
- Participants include the following:
 - **Filter**
 - performs certain tasks prior or after execution of request by request handler
 - **Filter Chain**
 - carries multiple filters and help to execute them in defined order on target
 - **Target** - the request handler
 - **Filter Manager**
 - manages the filters and Filter Chain
 - **Client**
 - the object who sends request to the Target object

Service Locator Design Pattern

- the service locator pattern is a relatively old pattern that was very popular with Java EE
 - goal of this pattern is to improve the modularity of your application by removing the dependency between the client and the implementation of an interface
- used whenever we want to locate/fetch various services using JNDI (Java Naming and Directory Interface) which, typically, is a redundant and expensive lookup
- encapsulates the processes involved in obtaining a service with a strong abstraction layer
 - uses a central registry known as the “service locator” which on request returns the information necessary to perform a certain task
- the ServiceLocator is responsible for returning instances of services when they are requested for by the service consumers or the service clients
- this pattern addresses this expensive lookup by making use of caching techniques
 - the very first time a particular service is requested, the service Locator looks up in JNDI, fetches the relevant service and then finally caches this service object
 - further lookups of the same service via Service Locator is done in its cache which improves the performance of the application to a great extent

Service Locator Participants

- Service
 - actual service which will process the request
 - reference of such service is to be looked upon in JNDI server
- Context / Initial Context
 - JNDI Context carries the reference to service used for lookup purpose
- Service Locator
 - a single point of contact to get services by JNDI lookup caching the services
- Cache
 - stores references of services to reuse them
- Client
 - the object that invokes the services via ServiceLocator

Transfer Object Pattern

- a frequently used design pattern when we want to pass data with multiple attributes in one shot from client to server, to avoid multiple calls to a remote server
- a simple POJO class having getter/setter methods and is serializable so that it can be transferred over the network
 - does not have any behavior
- a server side business class normally fetches data from the database and fills the POJO and then sends it to the client
- The following are participants of this pattern:
 - Business Object
 - fills the Transfer Object with data
 - Transfer Object
 - simple POJO having methods to set/get attributes only
 - Client
 - either requests or sends the Transfer Object to Business Object

Course Summary

Jason Fedin

Topics

- overview of Design Patterns (definition, gang of four)
- advantages of Design Patterns (why use design patterns?)
- types of Design Patterns (creational, behavioral, structural)
- important Design principles and strategies related to design patterns (Java)
 - Design Smells
 - Programming to an interface
 - Composition over Inheritance
 - Delegation Principles
 - Single Responsibility (Cohesion)
 - Open-Closed Principle
 - Liskov Substitution Principle
 - Interface Segregation Principle
 - Dependency Inversion
 - Dependency Injection

Topics (cont'd)

- UML
 - class diagrams
 - object-Oriented concepts in UML (inheritance, interfaces, composition, annotation)
- model View Controller
 - Goals and Advantages
- Creational Design Patterns
 - Factory
 - Abstract Factory
 - Singleton
 - Builder
 - Prototype

Topics (cont'd)

- Structural Design Patterns

- Adapter

- Bridge

- Composite

- Decorator

- Façade

- Flyweight

- Proxy

Topics (cont'd)

- Behavioral Design Patterns
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Course Outcomes

- you now understand the fundamentals of design patterns
- you can now create Java Applications that use design patterns
- you have mastered the art of problem solving in programming by using efficient, proven methods
- you have learned how to write high quality Java code
- you have become proficient in basic design principles

CONGRATULATIONS!!!!

- do not hesitate to ask questions
- provide feedback via the ratings
- offer constructive criticism
 - always looking to improve the course
- will make course updates to improve the class

Challenge Slides.

Applying the Factory Method Design Pattern

Jason Fedin

Requirements

- we are going to create an application that utilizes different types of animals (Duck and Tiger)
- your goal for this assignment is to utilize the factory method design pattern for creation of each type of animal
- you should implement this pattern using the most common technique (using a class with a single factory method that takes a parameter to determine which object gets created)

Requirements (cont'd)

- you should create an interface (product) that contains a common method for each type of Animal (Duck and Tiger)
 - the method could be something like walk or eat
- you should create a subclass for each type of Animal (Duck and Tiger)
 - each class should implement the walk or eat method from the Animal Interface
 - the implementation could simply print out some info
- you will need to create a concrete factory class with a factory method
 - this represents the creator part of the pattern
 - you can name it whatever you want, but, I would suggest AnimalFactory for the class and getAnimalType for the factory method
 - the factory method should take a String object which determines the “correct” Animal subtype to create
 - this method should return an Animal interface reference

Requirements (cont'd)

- lastly, create a demo class that includes a main method
 - this class will utilize the AnimalFactory to create a type of animal
 - after creating the Animal subtype which points to an Animal Reference
 - invoke the walk or eat method to verify that the correct object was created
- Output would include something like:
 - “Duck says Pack-pack”
 - “Tiger says: Halum..Halum”

Abstract Factory Challenge

Jason Fedin

Requirements

- we are going to create an application that utilizes different types of movies (HollywoodMovie and BollywoodMovie, our products) which have different genre's (comedy or action)
- your goal for this assignment is to utilize the abstract factory method design pattern for creation of each type of movie with the correct Genre
 - so, our factory will create either action or comedy movies of a certain family of products (Hollywood or Bollywood)
- you should implement this pattern using any technique you want (a static method for the creator, parameter passing to determine which type to create, etc)
 - the following slides suggest only one way to implement. This is by no means, the only way.
 - the suggested solution on the following slides heavily uses interfaces and subtypes in the main class

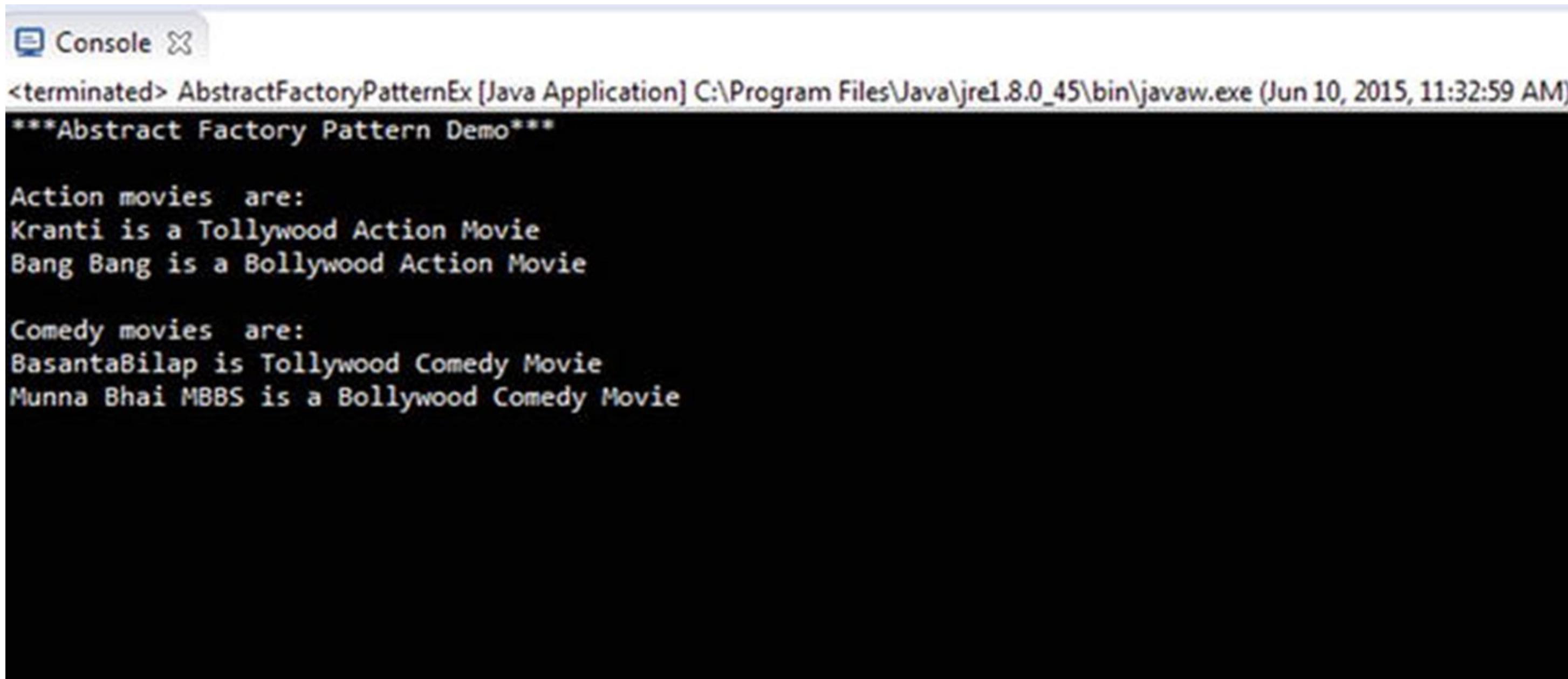
Requirements (cont'd)

- you should create two movie interfaces (Hollywood and Bollywood) that contain a common method
 - the method could be something like getting the movie name
- you should create a concrete class for each genre of movie (HollywoodComedyMovie, HollywoodActionMovie, BollywoodComedyMovie, BollywoodActionMovie) that implements the correct interface above
 - each class should implement the getMovieName method from the IHollywoodMovie or IBollywoodMovie Interface
 - the implementation could simply print out some hard-coded name
- you will need to create another interface for the Abstract factory pattern (creators)
 - this class will create a family of products (either Hollywood/Bollywood action movies or Hollywood/Bollywood comedy movies)
 - I would suggest IMovieFactory for the interface name and getHollywoodMovie and getBollywoodMovie for the factory methods (these methods will return a reference to either an IHollywoodMovie or an IBollywoodMovie)

Requirements (cont'd)

- you will also need to create classes that implement the factory methods from the IMovieFactory interface
 - ComedyMovieFactory and ActionMovieFactory are good names
 - the factory method should take a String object which determines the “correct” IMovie subtype to create
 - this method should return an IHollywoodMovie or IBollywoodMovie interface reference as mentioned in the previous slide
- lastly, create a demo class that includes a main method
 - this class will utilize the Comedy and Action movie factories to create a family of products (either Hollywood/Bollywood action movies or Hollywood/Bollywood comedy movies)
 - invoke the getMovieName method to verify that the correct object was created

Example Output



The image shows a Java console window titled "Console". The output text is as follows:

```
<terminated> AbstractFactoryPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Jun 10, 2015, 11:32:59 AM)
***Abstract Factory Pattern Demo***

Action movies are:
Kranti is a Tollywood Action Movie
Bang Bang is a Bollywood Action Movie

Comedy movies are:
BasantaBilap is Tollywood Comedy Movie
Munna Bhai MBBS is a Bollywood Comedy Movie
```

(Challenge) Singleton

Jason Fedin

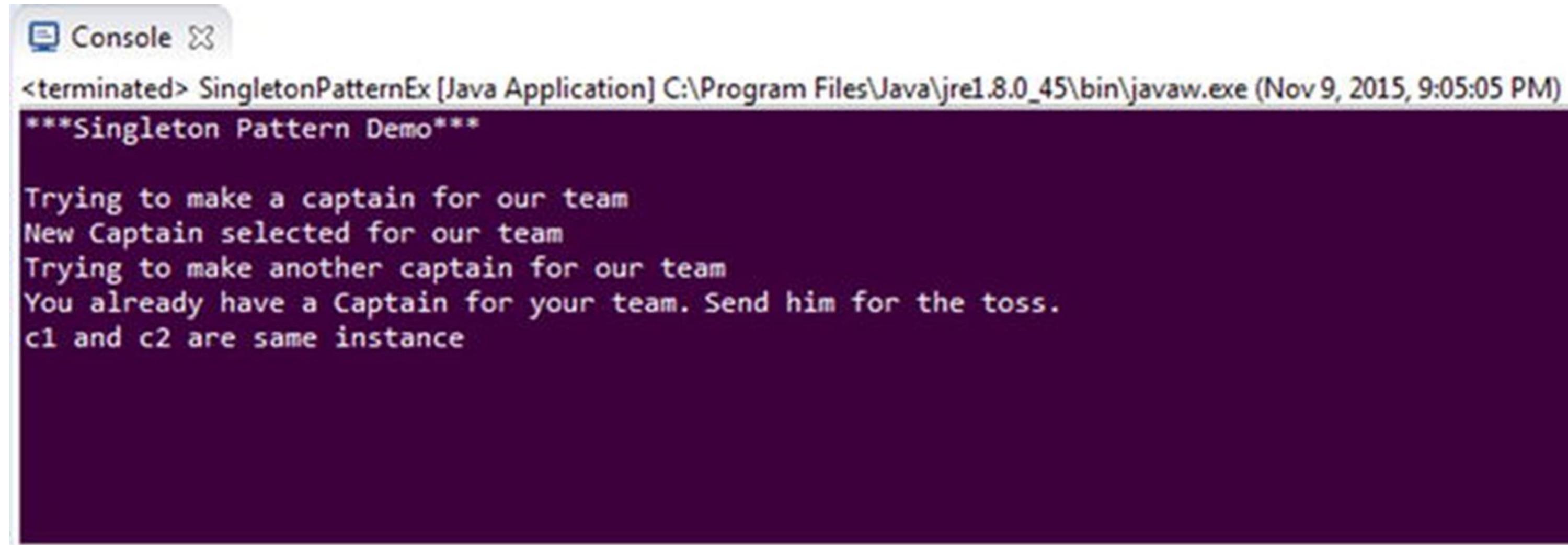
Requirements

- suppose you are a member of a cricket team
- you are in a tournament and your team is going to play against another team
- as per the rules of the game, the captain of each side must go for a toss to decide which side will bat (or bowl) first
- if your team does not have a captain, you need to elect someone as a captain first
 - at the same time, your team cannot have more than one captain
- your goal for this challenge is to use the singleton design pattern to ensure that your team only has one captain

Requirements (cont'd)

- you are required to use the Bill Pugh Singleton Implementation approach to ensure only one captain is selected on your team
 - use an inner static helper class to create your captain
- create a class that tests your implementation of the Singleton design pattern
 - call your getInstance() method multiple times to retrieve objects
 - see if the objects are the same by using the == operator in Java

Example Output



The screenshot shows a Java console window titled "Console". The title bar also displays the application name "SingletonPatternEx [Java Application]" and the path "C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe". The date and time "Nov 9, 2015, 9:05:05 PM" are also visible. The console output is as follows:

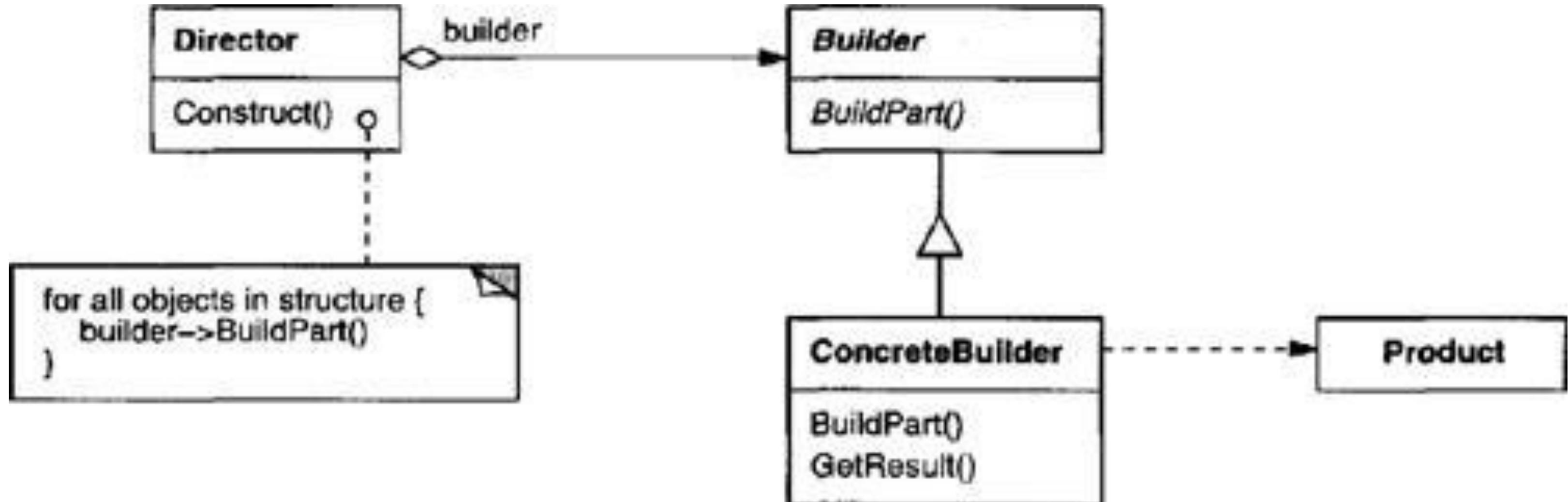
```
<terminated> SingletonPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 9, 2015, 9:05:05 PM)
***Singleton Pattern Demo***

Trying to make a captain for our team
New Captain selected for our team
Trying to make another captain for our team
You already have a Captain for your team. Send him for the toss.
c1 and c2 are same instance
```

(Challenge) Builder

Jason Fedin

Reminder (Class Diagram)



Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition By: Alan Shalloway; James R. Trott

Requirements

- suppose you are the manager of a fast-food restaurant
- typical meals consist of a burger and a cold drink
 - burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper
 - cold drink could be either a coke or a pepsi and will be packed in a bottle
- you need to utilize the Builder pattern to create objects that build different types of meals
 - vegetarian meal vs. a non-vegetarian meal
- we first need to create a hierarchy and abstract classes to set up our items (this is the representation of the object to build)
- you need to create an Item interface
 - methods could include a name and price for the item and a method that returns a Packing object in the Item Interface
 - represents food items such as burgers and cold drinks as abstract classes implementing the Item interface
 - Burger and ColdDrink abstract classes would implement the packing method but let other classes implement price and name

Requirements

- you need to create VegBurger and ChickenBurger concrete classes that extend the Burger class
 - implement the price and name methods by returning hardcoded floats and strings, respectively
- you need to create Coke and Pepsi concrete classes that extend the ColdDrink class
 - implement the price and name methods by returning hardcoded floats and strings, respectively
- you need to create a Packing interface
 - represents packaging of food items
 - concrete classes implementing the Packing interface such as Wrapper and Bottle
 - a burger would be packed in wrapper and a cold drink would be packed as a bottle
 - would include a single method such as pack() which returns a string representation of “Wrapper” or “Bottle”
- now, we need to start implementing the Builder Pattern

Requirements

- the next step is to create the class that represents the product in the Builder pattern
 - this is the meal
- create a Meal class having an ArrayList of Items
 - this class can have addItem, getCost, and showItems methods
 - getCost and showItems would iterate through each item in the ArrayList to determine the cost and show each item's name, packing type, and total price
- create a MealBuilder interface that acts as the abstract class builder
 - should include buildBurger, buildDrink, and getMeal methods

Requirements

- create concrete builder objects that implement the MealBuilder interface
 - VegMealBuilder and NonVegMealBuilder
 - contains a Meal object
 - will build different types of Meal objects by adding items to the Meal arraylist
- create the Director class from the builder pattern
 - contains an IMealBuilder member variable
 - includes a construct method that takes an IMealBuilder object as a parameter
 - includes the steps to create the object
- lastly, create a demo class which will use the Director which uses a MealBuilder to build a Meal

Example Output

Veg Meal

Item : Veg Burger, Packing : Wrapper, Price : 25.0

Item : Coke, Packing : Bottle, Price : 30.0

Total Cost: 55.0

Non-Veg Meal

Item : Chicken Burger, Packing : Wrapper, Price : 50.5

Item : Pepsi, Packing : Bottle, Price : 35.0

Total Cost: 85.5

(Challenge) Prototype

Jason Fedin

Requirements

- Suppose we want to create a car application that contains different types of cars
- we want to utilize the prototype design pattern to return duplicate cars
 - you are required to implement this pattern using the Java Cloneable interface
- the prototype for our application is named “BasicCar”
 - should be an abstract class, has some implementation
 - should include attributes for model name and price
 - should be created with some default price
 - the price could also be modified later as per the model
- the ConcretePrototypes are Nano and Ford
 - they need to implement the Clone() method defined in BasicCar

Requirements

- a class BasicCarCache should be defined
 - stores BasicCar objects in a Hashtable
 - returns their clone when requested
- you are also required to implement a driver/demo class named PrototypePatternEx
 - this is the client
 - will use BasicCarCache class to get a BasicCar object

Example Output

The screenshot shows a Java application running in an IDE's console tab. The title bar indicates it is a Java Application named 'PrototypePatternEx' (version 2) running on Jun 7, 2015, at 9:49:47 AM. The console output displays the following text:

```
<terminated> PrototypePatternEx (2) [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Jun 7, 2015, 9:49:47 AM)
***Prototype Pattern Demo***

Car is: Green Nano and it's price is Rs.189818
Car is: Ford Yellow and it's price is Rs.561925
```

(Challenge) Adapter

Jason Fedin

Requirements

- you will need to implement the Adapter pattern for an application that plays media
 - will need to implement the pattern using an Object adapter (composition)
- create a MediaPlayer interface and a concrete class named AudioPlayer
 - AudioPlayer implements the MediaPlayer interface
 - the MediaPlayer interface contains a single method named play
 - takes a string for audioType and a string for filename
 - AudioPlayer can play mp3 format audio files (audioType) by default

Requirements (cont'd)

- create another interface named AdvancedMediaPlayer and concrete classes VlcPlayer and Mp4Player
 - VlcPlayer and Mp4Player implement the AdvancedMediaPlayer interface (different interface than MediaPlayer)
 - the AdvancedMediaPlayer interface includes two methods loadFilename and listen
 - loadFilename takes a string representing the filename to listen to
 - listen will use the filename member variable to listen to audio
 - these classes can play vlc and mp4 format files
- we want to make the MediaPlayer play other formats in addition to the AudioPlayer
 - we will need to create an Adapter that adapts an AdvancedMediaPlayer to a MediaPlayer

Requirements (cont'd)

- create an adapter class named AdvancedMediaPlayerAdapter
 - should implement the MediaPlayer interface
 - should use AdvancedMediaPlayer object to listen to the required format (composition)
 - set the AdvancedMediaPlayer member object via the MediaAdapter constructor
 - the AdvancedMediaPlayerAdapter play method takes the desired audio type as input and a filename
 - does not know the actual class which can play the desired format
 - invokes the loadFilename and listen methods
- AdapterPatternDemo is our demo class
 - will use MediaPlayer class to play various formats

Participants

- Target
 - defines the domain-specific interface that Client uses
 - MediaPlayer
- Client
 - collaborates with objects conforming to the Target interface
 - mp4, vlc are the clients (implement the AdvancedMediaPlayer interface)
 - incompatible with MediaPlayer Interface
 - collaborates with AdvancedMediaPlayerAdapter which conforms to MediaPlayer interface
- Adaptee
 - defines an existing interface that needs adapting
 - AdvancedMediaPlayer
- Adapter
 - adapts the interface of Adaptee to the Target interface
 - AdvancedMediaPlayerAdapter - adopts the AdvancedMediaPlayer to the MediaPlayer Interface

Example Output

Playing mp3 file. Name: beyond the horizon.mp3

Playing mp4 file. Name: alone.mp4

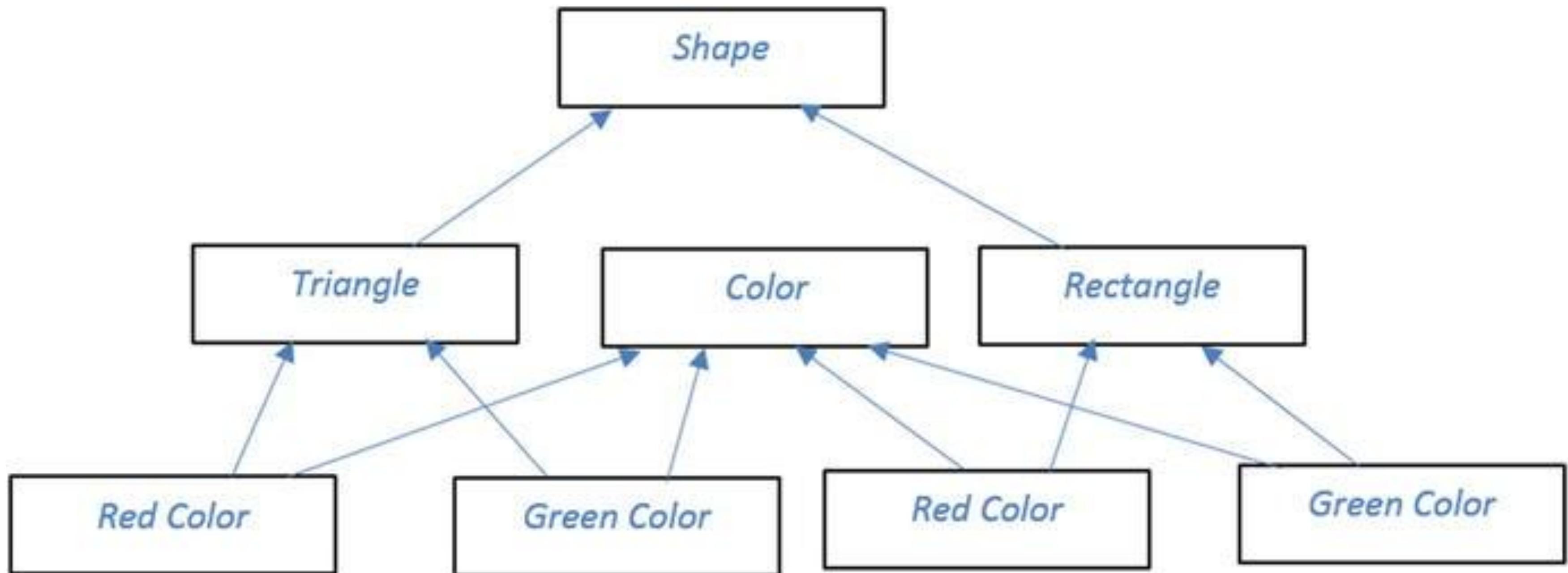
Playing vlc file. Name: far far away.vlc

Invalid media. avi format not supported

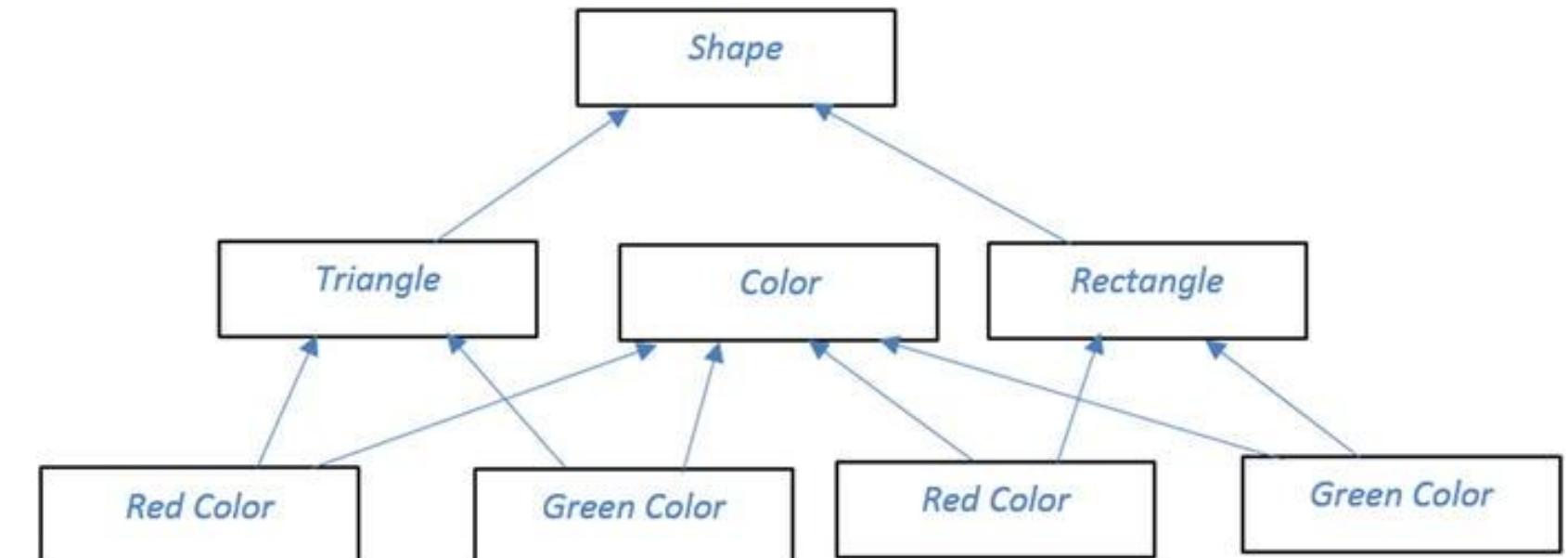
Bridge Pattern Challenge

Jason Fedin

Requirements (Current Design)



Requirements



- we want to use bridge pattern to decouple the above design
 - abstraction (Shape) and implementer (Colors)
- after this redesign and implementation it will have a cleaner look
- the resulting implementation will include an abstraction-specific and an implementer-specific method to represent the power and usefulness of this pattern

Requirements

- you are required to write a drawing program
 - will allow you to draw different shapes filled with different colors and different border sizes
- this program will consist of a drawing interface which acts as a bridge implementer
 - the interface should be named IColor
 - could include a single method: fillWithColor that takes an int border
 - concrete classes of this interface should be named RedCircle and GreenCircle
 - should implement fillWithColor by just printing out a message

Requirements

- the bridge abstraction should include an abstract class named shape which will use the drawing interface
 - should include abstract methods to drawShapes and modify borders
- you will need to create concrete classes implementing the Shape interface
 - Triangle and Rectangle
 - we can draw these shapes with a particular color with the implementer-specific method drawShape()
 - we can change the thickness of the border by the abstraction-specific method modifyBorder()
- you need to write a main class that will use the Shape class to draw different colored circles

Example Output

```
Console ✘  
<terminated> BridgePatternDemo2 [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 19, 2015, 1:29:15 PM)  
*****BRIDGE PATTERN*****  
  
Coloring Triangle:  
This Triangle colored with: Green color with 20 inch border.  
Now we are changing the border length 3 times  
This Triangle colored with: Green color with 60 inch border.  
  
Coloring Rectangle :  
This Rectangle colored with: Red color with 50 inch border  
Now we are changing the border length 2 times  
This Rectangle colored with: Red color with 100 inch border
```

Steps to complete challenge

- create the bridge implementer interface (IColor.java)
- create concrete bridge implementer classes implementing the IColor interface (GreenColor, RedColor)
- create an abstract class Shape using the IColor interface
- create concrete classes implementing the Shape interface (Triangle and Rectangle)
- use the Shape and IColor classes to draw different colored shapes

(Challenge) Composite

Jason Fedin

Requirements

- lets use the composite pattern to implement an application that stores information for a college organization
 - we want to understand the hierarchy of the faculty
- we have many professors with different titles
 - adjunct, associate, or a full professor
- we have two heads of departments (department chairs)
 - one for computer science
 - one for mathematics
- the mathematics department has two faculty members
- the computer science department has three faculty members
- we also have a dean who can be the head of multiple departments

Requirements

- the grouping that we want to achieve is to show the hierarchy of the technology department
 - who is the dean of technology
 - how many departments or chairs are under this dean
 - how many professors are in each department
- So, you will create many different objects that represent this hierarchy
 - one dean of technology (composite, since it has faculty under it)
 - two departments (math and computer science, with chairs) (composite, since it has faculty under it)
 - 2 professors under math and 3 professors under computer science (each professor is a primitive object)

Steps – Identify and create the component

- our first step is to create the component
 - declares the interface for objects in the composition
 - implements default behavior for the interface common to all classes
 - declares an interface for accessing and managing its child components (this should instead be put into the composite class)
- our component is represented by the Faculty Interface
 - contains a getDetails method that returns a string
 - lets assume that the dean, chair, and professors are all faculty members

Steps – Identify and create the leaf

- our second step is to create the leaf
 - represents leaf objects in the composition
 - a leaf has no children
 - defines behavior for primitive objects in the composition
- our leaf is a Professor
 - they all have a name, position, and office number
 - Implements the Faculty interface

Steps – Identify and create the composite

- Our third step is create the composite
 - defines behavior for components having children
 - stores child components
 - implements child-related operations in the Component interface (should also declare these methods, add, remove, etc)
- our composite is the Supervisor who has faculty under them
 - includes name and department name
 - methods include add, remove, getMyFaculty, and getDetails
- the component represents a group of objects (deans who have chairs who have professors)(chairs who have professors under them)
 - we can treat a group of objects (deans and chairs) the same as a single object (professors)

Steps – Create the client

- our last step is to create the client
 - manipulates objects in the composition through the Component interface
- remember, we have
 - one dean
 - two heads of departments (department chairs)
 - one for computer science
 - one for mathematics
 - mathematics department has two faculty members
 - computer science department has three faculty members
- all of these objects should be created and organized based on their grouping
 - two component objects
 - dean (all faculty under dean)
 - chairs (all faculty under chairs)
 - rest are primitive objects (professors)

Requirements

- client should display all faculty members in formatted output
 - dean first
 - head of departments in 2 tabs
 - each faculty member of each department in 4 tabs
- client should then delete one faculty member from the computer science department and redisplay the list of all faculty members in the above same format

Example Output

***COMPOSITE PATTERN DEMO ***

The college has the following structure

Dr. S.Som is the Dean of Technology

Mrs.S.Das is the Chair of Math Department

Mr. V.Sarcar is the Chair of Computer Science Department

Math Professor1 is the Adjunct

Math Professor2 is the Associate

CSE Professor1 is the Adjunct

CSE Professor2 is the Professor

CSE Professor3 is the Professor

Example Output (cont'd)

After CSE Professor2 leaving the organization- CSE department has following faculty:

CSE Professor1 is the Adjunct

CSE Professor3 is the Professor

Challenge (Decorator)

Jason Fedin

Requirements

- we are going to use the decorator pattern to add additional functionality to application that draws various Shapes
- you will need to create a component interface, decorator interface, concrete components and concrete decorators
- you are required to create a Shape interface (component)
 - add a simple draw method
- you are required to implement concrete classes implementing the Shape interface (concrete components)
 - Rectangle and Circle
 - implement draw by just displaying some default string

Requirements (cont'd)

- you are required to create an abstract decorator class named ShapeDecorator
 - implements the Shape interface
 - has a Shape object as its instance variable
- you are required to create some concrete decorators classes
 - RedShapeDecorator, GreenShapeDecorator??
 - add additional functionality such as setting a red or green border or changing styling
- lastly, create a main class that uses your decorator to decorate shape objects

Example Output

Circle with normal border

Shape: Circle

Circle of red border

Shape: Circle

Border Color: Red

Rectangle of red border

Shape: Rectangle

Border Color: Red

(Challenge) Facade

Jason Fedin

Requirements

- lets write an application that simulates accessing different menus from multiple restaurants in a hotel
- currently, the application is difficult to use as the interface to eat food at a restaurant is overly complicated
- for this challenge, you are required to create a façade so that it is easier to order/eat at a restaurant at a Hotel
- this hotel has a hotel keeper
- there are a lot of restaurants inside hotel
 - Veg restaurants
 - Non-Veg restaurants
 - Veg/Non, both restaurants

Requirements

- as a client you want to access all the different menus of all the restaurants
- you do not know all of the different menus that are available
- all you have is access to the hotel keeper who knows his hotel quite well
- whenever you want a menu
 - you tell the hotel keeper
 - he retrieves the correct menu from the respective restaurants
 - hands the menu over to you
- the hotel keeper acts as the façade
 - he hides the complexities of the hotel system

Example Interface (Hotel.java)

- the hotel interface only returns Menus
 - I will provide you with an intelliJ project that includes the Menus interface
 - I will provide you with an intelliJ project that includes three classes that implement this interface (NonVegMenu, VegMenu, and Both)
 - just have default constructors that output the object created

```
public interface Hotel
{
    public Menus getMenus();
}
```

Current implementation (3 available restaurants)

- there are three types of restaurants that implement the hotel interface

(NonVegRestaurant.java)

```
public class NonVegRestaurant implements Hotel
{
    public Menus getMenus()
    {
        NonVegMenu nv = new NonVegMenu();
        return nv;
    }
}
```

VegRestaurant.java

```
public class VegRestaurant implements Hotel
{
    public Menus getMenus()
    {
        VegMenu v = new VegMenu();
        return v;
    }
}
```

VegNonBothRestaurant.java

```
public class VegNonBothRestaurant implements Hotel
{
    public Menus getMenus()
    {
        Both b = new Both();
        return b;
    }
}
```

Requirements (cont'd)

- you need to create a simpler interface (façade)
 - HotelKeeper
 - should contain 3 methods to get each type of menu
- create a client that uses the façade
 - asks the façade for each type of menu
- it should be clear that the complex implementation will be done by HotelKeeper
 - the client will just access the HotelKeeper and ask for either Veg, NonVeg or VegNon/Both restaurant menus

Sample output

Created a VegMenu

Created a NonVegMenu

Created both menus

(Challenge) Flyweight

Jason Fedin

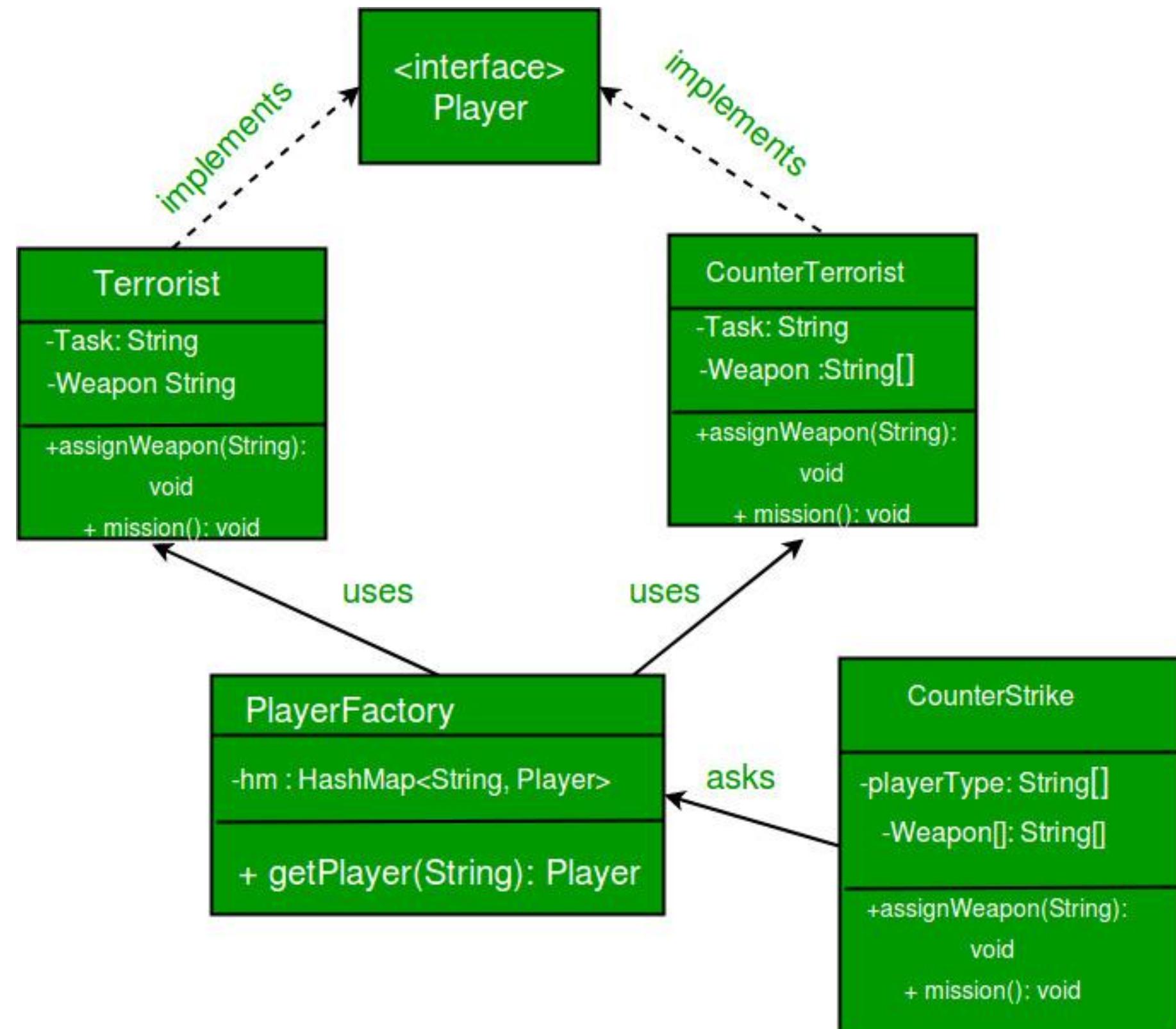
Requirements

- lets write an application that simulates the popular counter strike game
 - we will implement the creation of terrorists and counter terrorists objects using the flyweight pattern
- you will need to 2 classes
 - Terrorist(T)
 - Counter Terrorist(CT)
- whenever a player asks for a weapon we assign him the asked weapon
- in the mission, the terrorist's task is to plant a bomb while the counter terrorists have to diffuse the bomb
- we want to reduce the object count for each player and will use the flyweight pattern to achieve this
 - we have n number of players playing this game
 - if we do not follow the FlyWeight Design Pattern then we will have to create n number of objects, one for each player
- using the flyweight pattern, we will only have to create 2 objects
 - one for terrorists and another for counter terrorists
 - will reuse them again and again whenever required

Requirements (cont'd)

- it is important to identify the intrinsic and extrinsic state when using the flyweight pattern
- for our application, 'task' is an intrinsic state for both types of players
 - always the same for terrorists and counter-terrorists
 - we can have some other states like their color or any other properties which are similar for all the Terrorists/Counter Terrorists
- for our application, 'weapon' is an extrinsic state
 - each player can carry any weapon of his/her choice
 - weapon needs to be passed as a parameter by the client itself

Possible design



Possible Output

Counter Terrorist Created

Counter Terrorist with weapon Gut Knife | Task is DIFFUSE BOMB

Counter Terrorist with weapon Desert Eagle | Task is DIFFUSE BOMB

Terrorist Created

Terrorist with weapon AK-47 | Task is PLANT A BOMB

Terrorist with weapon Gut Knife | Task is PLANT A BOMB

Terrorist with weapon Gut Knife | Task is PLANT A BOMB

Terrorist with weapon Desert Eagle | Task is PLANT A BOMB

Terrorist with weapon AK-47 | Task is PLANT A BOMB

Counter Terrorist with weapon Desert Eagle | Task is DIFFUSE BOMB

Counter Terrorist with weapon Gut Knife | Task is DIFFUSE BOMB

Counter Terrorist with weapon Desert Eagle | Task is DIFFUSE BOMB

(Challenge) Proxy

Jason Fedin

Requirements

- a very simple real life scenario is accessing the internet from the library
 - restricts access to some sites
- you need to implement the proxy design pattern to restrict this access
- the proxy will first check the host you are connecting to
 - if it is not part of restricted site list, then it connects to the real internet
- create a subject class contains a method to connect to the internet
 - takes a string representing the hostname
 - throws an exception if the hostname is restricted

Requirements (cont'd)

- create the real subject class which actually connects to the internet
 - can just hard-code some output that says “Connected to the internet...”
- create the proxy class
 - has a reference to the real subject class
 - contains a list of restricted sites (to search)
 - If host you are trying to connect to is a banned site then throw an exception
 - otherwise, connect to internet
- create the client
 - uses the proxy to try to connect to legitimate sites

Sample Output

Connecting to jasonfedin.org

Access Denied

(Challenge) Chain of Responsibility

Jason Fedin

Requirements

- let's write an error handler application that uses the chain of responsibility pattern
- consider an application which is handling email or fax errors
 - we need to take care of the issues reported in each of these communications
- we will need two error handler classes (concrete handlers)
 - EmailErrorHandler and FaxErrorHandler
 - EmailErrorHandler will handle e-mail errors only and is not responsible for fax errors
 - will parse a message for text that contains the word "Email"
 - FaxErrorHandler will handle fax errors and does not care about e-mail errors
 - will parse a message for text that contains the word "Fax"

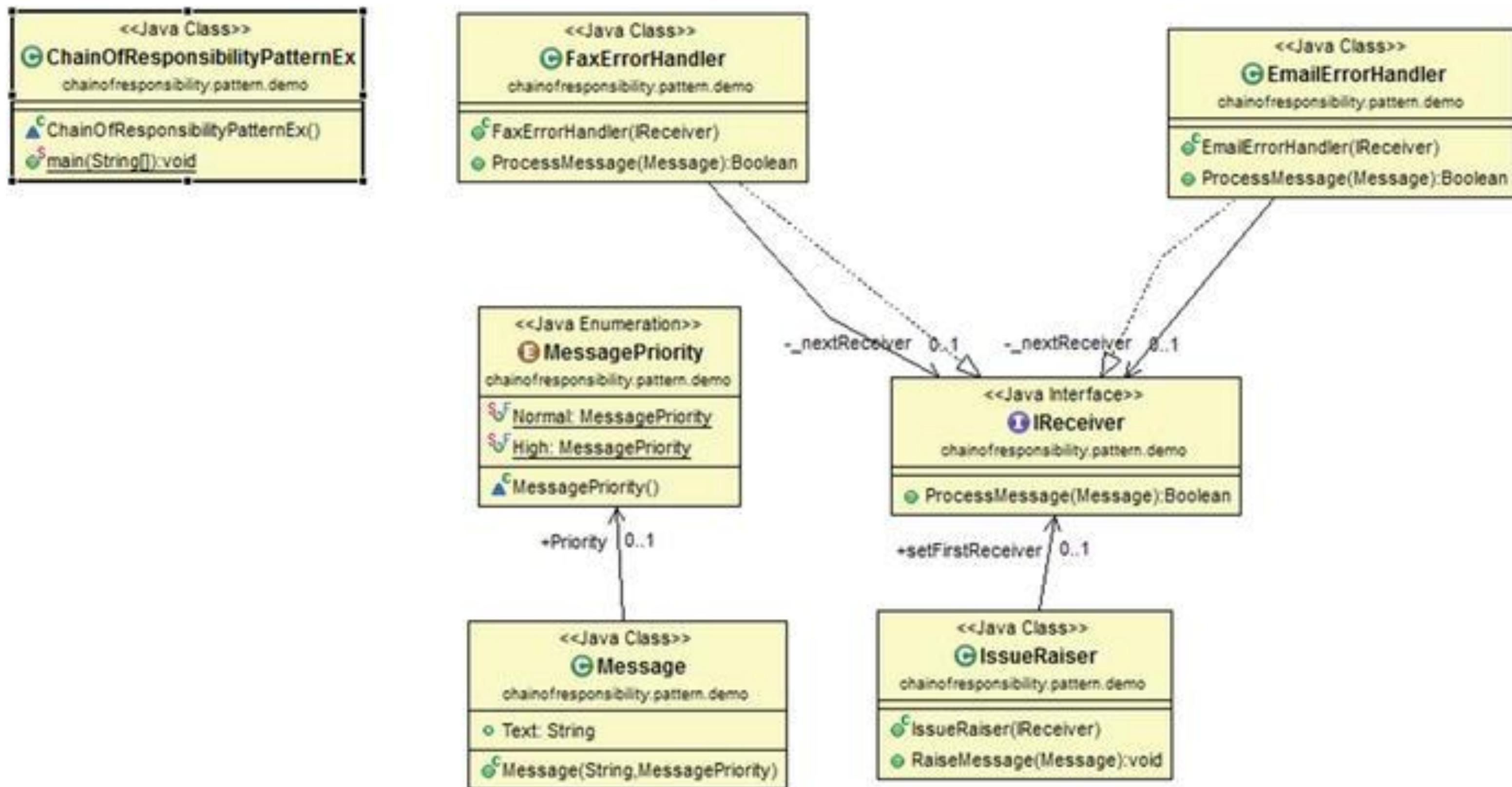
Requirements

- we can make a chain as follows
 - whenever our main application finds an error
 - it will just raise this and forward the error with the hope that one of those handlers will handle it
- the request will first come to FaxErrorHandler (first in chain)
 - if it finds that it is a fax issue, it will handle the request
 - otherwise, it will forward the issue to EmailErrorHandler (second in chain)
- our chain ends with EmailErrorHandler

Requirements (cont'd)

- your application should also be able to process both normal and high-priority issues from e-mail and fax communications
- you could create helper classes for MessagePriority (enum) and Message (test, MessagePriority member variables)
- another helper class may be useful for initiating the chain
 - IssueRaiser class could be used by the client to start the chain
- your handler interface can have methods to processMessage(Message) and setNextChain

Example Class Diagram



Output

```
Console ×
<terminated> ChainOfResponsibilityPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 18, 2015, 2:46:10 PM)
***Chain of Responsibility Pattern Demo***

FaxErrorHandler processed Normalpriority issue: Fax is reaching late to the destination
EmailErrorHandler processed Highpriority issue: Email is not going
EmailErrorHandler processed Normalpriority issue: In Email, BCC field is disabled occationally
FaxErrorHandler processed Highpriority issue: Fax is not reaching destination
```

Other considerations

- if we need to handle another type of issue (e.g., Authentication), we can make an AuthenticationErrorHandler and put it after EmailErrorHandler
- whenever the issue cannot be fixed by EmailErrorHandler, the issue can be forwarded to AuthenticationErrorHandler and the chain will end there
- the bottom line is as follows
 - the chain will end if the issue is being processed by some handler or there are no more handlers to process it (i.e., we have reached the end of the chain)

(Challenge) Command

Jason Fedin

Requirements

- let's write an application that simulates the buying and selling of stocks via a broker
- you will use the command pattern to implement this buying and selling of stocks
- you need to create an interface Order which acts as a command
 - only contains the execute method
- create concrete command classes for buying and selling stocks
 - should contain a receiver object
 - can pass the receiver object via the constructor
 - an execute method will invoke a "buy" or "sell" method on the receiver

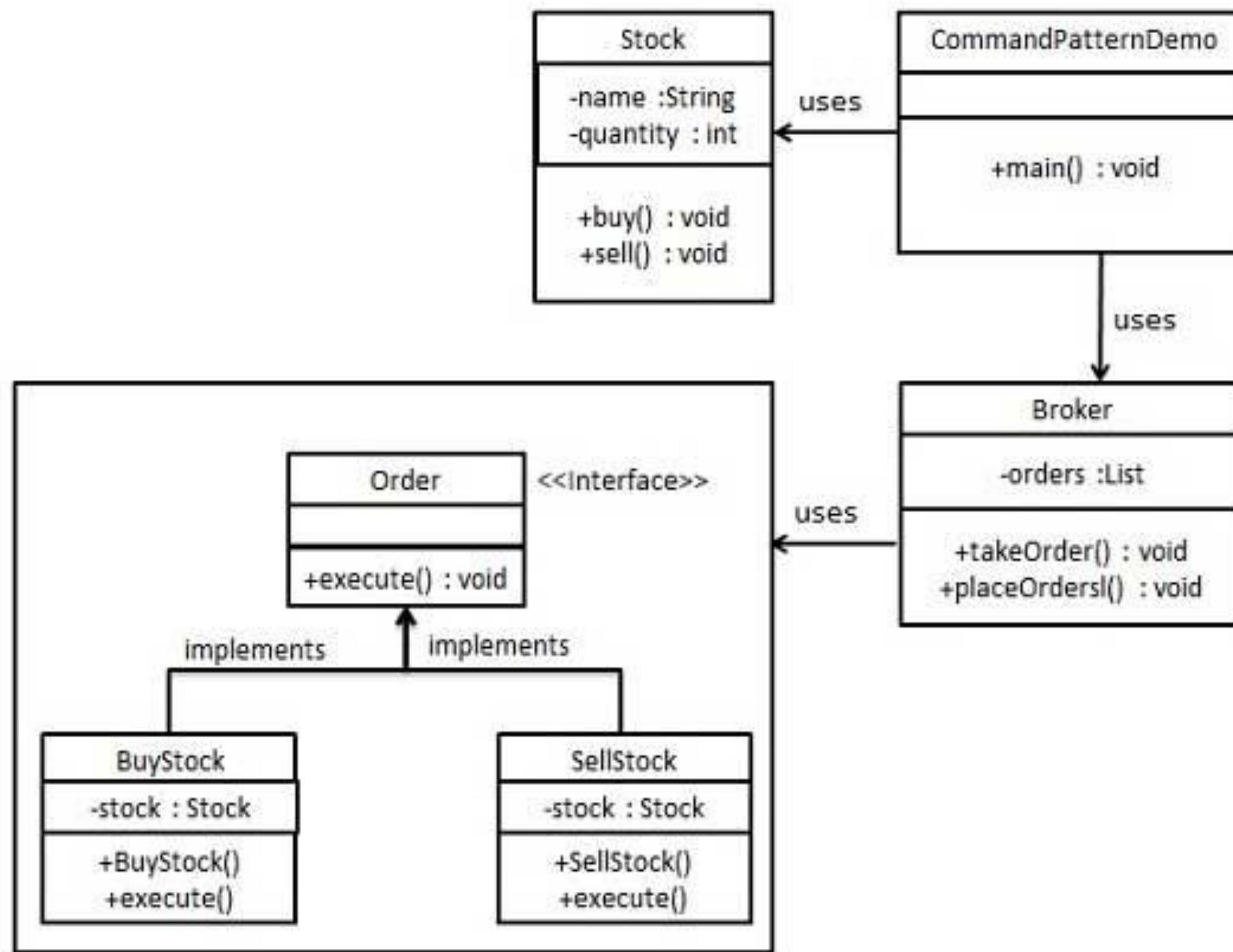
Requirements

- a Stock class could act as the receiver
 - does the actual buying and selling (actual work)
 - can just output hard-coded strings for operations
 - would contain a name and quantity for the stock as member variable and include them in the output
- a class Broker could act as an invoker object
 - takes and places orders (a list of orders)
 - takeOrder could be a method to add orders to a list
 - placeOrders would go through the list and invoke each execute method on the specific command
- the broker object uses the command pattern to identify which object will execute which command based on the type of command

Requirements

- the client should create the receiver and associate it with a command (via the command constructor)
 - should have buy and sell orders
- the client should create the invoker and associate it with each command by calling the takeOrder method
 - pass command to this method as a parameter
- lastly, the client will execute all orders by telling the broker to place orders

Class Diagram



Example Output

Stock [Name: Google, Quantity: 100] bought

Stock [Name: Google, Quantity: 100] sold

(Challenge) Interpreter

Jason Fedin

Requirements

- let's create a simple application that utilizes the interpreter design pattern
- this application will interpret sentences that contain "AND" and "OR"
 - with an expression on each side (exp AND exp) (exp OR exp)
 - will return a boolean result based on AND or OR logic (both have to be true for AND and only one has to be TRUE for OR)
- our first step will be to write the Interpreter context class that will do the actual interpretation (determine when to return true or false)
 - this class will contain a string that represents the input
 - you should create a getResult function that returns true if the input contains the data of the expression (just use the contains string method)
- the next step is to create the expression interface
 - this should include a single interpret method that takes a context and returns a boolean

Requirements

- there is only one terminal class required for this language
 - will contain a constructor that sets the data for the expression
 - will implement the interpret method by calling the context getResult method and passing in the expression data
- there should be two non-terminal expression classes
 - one for AND and one for OR
 - each class should contain two expression references (set in constructor)
 - interpret method should AND or OR the two expression references
 - will use recursion

Requirements

- the last step is to create a client
- you need to create at least two Terminal expressions that contain strings
 - use each one to pass into your non-terminal expression (AND, OR)
- need to create a context (with an input) that will return true or false based on which non-terminal expression was created (AND or OR)
- for example, you could create two terminal expressions that represent a person and marriage
 - one is "Jason" and another is "Married"
- you then create a non-terminal expression (AND) representing if two people are married using your two terminal expressions
 - I would pass these expressions into my non-terminal AND expression constructor
 - if I create a context that represents is Jason Married? ("Married Jason") the AND expression should interpret this as TRUE
 - if I create a context that represents is John Married ("Married John") the AND expression should interpret this as FALSE
- you will notice that this client class represents the grammar
 - contains an instance for each terminal and each non-terminal (Jason and John are terminals and AND is a non-terminal)

Requirements

- another example would be to create the following two terminal expressions that represent a person
 - one is “Lucy” and another is “Myra”
- you then create a non-terminal expression (OR) representing a female by using the two terminal expressions you created
 - I would pass these expressions into my non-terminal OR expression constructor
 - if I create a context is Lucy Female (“Lucy”) the OR expression should interpret this as TRUE
 - if I create a context is Myra Female (“Myra”) the OR expression should interpret this as TRUE
 - If I create a context is Jason Female (“Jason”) the OR expression should interpret this as FALSE

Grammar

- the grammars for my language (using the examples on previous slides) would be
 - AND_EXPR -> exp AND exp
 - OR_EXPR -> exp OR exp
 - exp -> “Jason”, “Married”, “Lucy”, “Myra”
- for your application, you can create any terminals that you want

Output

Lucy is female? True

Myra is female? True

Jason is female? False

Jason is married? True

John is married? False

(Challenge) Iterator

Jason Fedin

Requirements

- we are going to create an application that is used in a college environment
 - will access specific data (different subjects) of certain departments
- different departments store information in different data structures
- the arts department uses an array data structure to store different subjects
- the science department uses a linked list data structure to store different subjects
- you are required to use the iterator pattern so that you can access this data through common methods
 - it does not care which data structure is used by individual departments
 - we are displaying the subjects using iterators

Design

- you are required to create an Iterator interface
 - will contain common methods for both types of iterators (Science and Arts)
 - includes methods:
 - first: void (reset to first element)
 - next(): String (get next element)
 - isDone(): Boolean (end of collection check)
 - currentItem(): String (retrieve current item)
- we are displaying subjects through the common methods isDone() and next()
 - you can use the other two methods also (first() and currentItem())
- concreteIterators named ArtsIterator and ScienceIterator need to be created
 - should be inner classes of your concreteAggregates (only can use iterators to traverse)
 - constructors will take an array for Arts and a linked list for Science

Requirements

- create your aggregate interface (ISubject)
 - includes the createIterator method which returns an iterator
- create your concrete aggregates
 - Arts should use an array to store subjects (constructor or main can populate array with default strings)
 - Science should use a LinkedList to store subjects (constructor or main can populate list with default strings)
- lastly, create your client
 - should display a bunch of subjects using both types of iterators (arts or science)

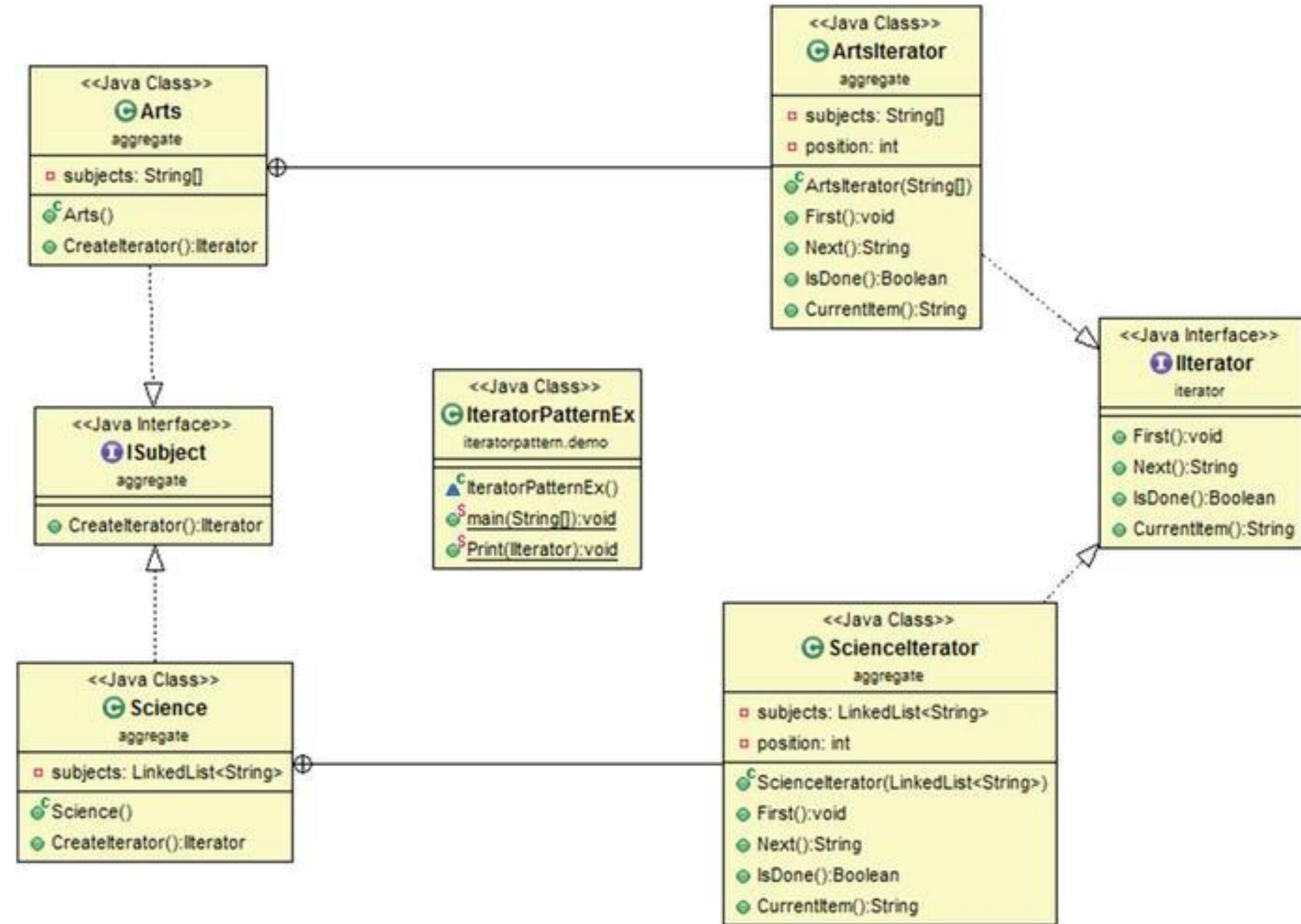
Example Output

```
Console X
<terminated> IteratorPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 14, 2015, 12:15:08 PM)
***Iterator Pattern Demo***

Science subjects :
Maths
Comp. Sc.
Physics

Arts subjects :
Bengali
English
```

Example Class Diagram



Summary

- this implementation allows us to support different variations for the traversal of an aggregate (the interface to create an Iterator object), and above all, it simplifies the interface
- we must be careful while traversing and any kind of modification during that traversal period can cause problems

(Challenge) Mediator

Jason Fedin

Requirements

- lets create an application that simulates an auction
- we will use the mediator pattern for communication of all bids done by buyers
- the Auction Mediator is responsible
 - for adding the buyers
 - determining which buyer won each auction
 - letting each buyer know that the auction has ended (after finding the winner)
- so, your first step is to create the mediator interface
 - can add buyers and find the highest bidder

Requirements

- the next step is to create concrete mediators
 - should maintain a list of all buyers
 - find highest bidder will traverse list to see who has the highest bid
 - send message to buyer that the auction is over
- create the colleague abstract class (Buyer)
 - contains methods for bidding, canceling a bid, and notification of when an auction has ended
 - state for name and price
- create the concrete colleagues (AuctionBuyer)
 - can display default message when auction has ended

Requirements

- create the clients
 - should simulate an auction
 - create some buyers
 - add buyers to mediator
 - have buyers place bids, cancel bids, etc.

Example Output

Tal Baum was added to the buyers list.

Elad Shamilov was added to the buyers list.

John Smith was added to the buyers list.

Welcome to the auction. Tonight we are selling a vacation to Vegas. please make some offers.

Waiting for the buyer's offers...

The auction winner is Elad Shamilov.

He paid 2000\$ for the item.

Elad Shamilov Has canceled his bid!, In that case The auction winner is Tal Baum.

He paid 1800\$ for the item.

Tal Baum has been notified that the auction is over

Elad Shamilov has been notified that the auction is over

John Smith has been notified that the auction is over

Goals

- you should have a clear idea that this pattern is very useful when we observe complex communications in a system
 - Communication (among objects) is much simpler with this pattern
- this pattern reduces the number of subclasses in the system and it also enhances the loose coupling in the system
- here the “many-to-many” relationship is replaced with the “one-to-many” relationship—which is much easier to read and understand
- we can provide a centralized control with this pattern
- sometimes the encapsulation process becomes tricky and we find it difficult to maintain or implement

(Challenge) Memento

Jason Fedin

Requirements

- lets create an application that uses the Memento pattern!
- this application will save some “dummy” states
- you must create the three main classes to implement this pattern
 - Memento
 - stores the internal state of the Originator object
 - Originator
 - creates a memento containing a snapshot of its current internal state
 - uses the memento to restore its internal state
 - Caretaker
 - responsible for the memento’s safekeeping
 - never operates on or examines the contents of a memento

Requirements

- the originator class should contain some state data and get/set methods for each state
 - should also contain a save or createMemento method
 - will return a new memento with the saved state
 - should also contain an restore or getStateFromMemento method
 - restores the originator's state from the memento state (saved state)
- your memento class should contain the same state data as the originator (does not need to be an inner class)
 - a constructor that sets the state and a get method
- your caretaker class must be able to save multiple states using the memento objects
 - maintain a list of memento objects
- the caretaker can include add and get methods which add memento's to the list and retrieve a memento from the list (based on index)

Requirements

- your client should use the Originator and CareTaker objects
- create an originator object and set multiple states
- save multiple states (save points) using the originator's memento object
- restore multiple save points to demonstrate that the caretaker worked (look up different mementos in the caretaker's list)

Example Output

Current State: State #6

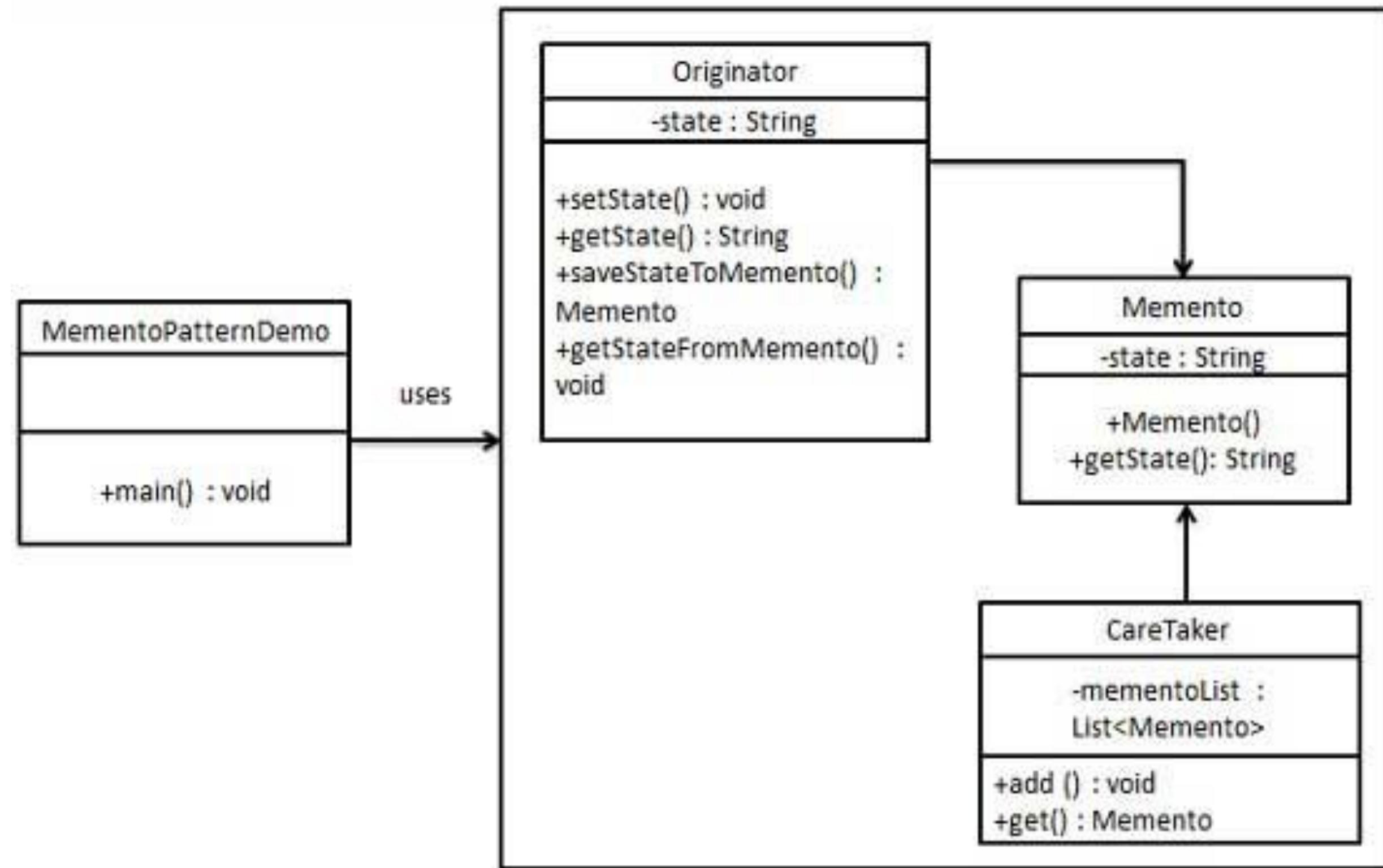
First saved State: State #2

Second saved State: State #4

Reminders

- the memento object should be treated as an opaque object
 - caretakers should not be allowed to change them
- we should pay special attention so that other objects are not affected by the change made in the originator to the memento
- sometimes the use of this pattern can cost more
 - if we want to store and restore large amount of data frequently
- also, from a caretaker point of view, the caretaker has no idea about how much state is kept in the memento that it wants to delete

Example class diagram



(Challenge) Observer

Jason Fedin

Requirements

- lets build a cricket application that uses the observer pattern to notify viewers about the latest updates in a match
- this application will let viewers know the following information
 - runs, wickets, and overs
 - will use the above information to determine the average score and current score
- the viewers will be the Observers
 - two classes that display the average score and the current score
 - uses data from the subject (current score, runs, wickets, and overs)
- the subject will be the CricketData and update the observers whenever the runs, wickets, or overs changes

Steps

- create the subject interface
 - contains methods for register, unregister, and notify
- create the concreteSubject (CricketData)
 - contains state data for run, wickets, and overs (hard-code values)
 - keeps a list of all observers (arrayList)
 - passes state data to update in observer in notifyAll
 - includes a method to trigger notifications (dataChanged)
- create the Observer interface
 - should only include update method (with state data as parameters)
 - no need for any observers to talk to subject

Steps

- create the concrete observers (data changes passed to update method)
 - AverageScore and CurrentScore
- AverageScore observer should have data to compute the run rate and predicted score
 - run rate is calculated by dividing runs/over
 - predicted score is calculated by multiplying the runrate * 50
- CurrentScore observer should have data for runs, wickets, and overs
 - will just display this data when passed to the update method

Steps

- create the client to simulate some cricket data
- create the observers
- create the subject
 - register the observers with the subject
- trigger changes in the subject which will cause all observers to be notified
- remove an observer and then re-trigger to see if that observer is no longer notified

Example Output

Average Score Display:

Run Rate: 8.823529

PredictedScore: 441

Current Score Display:

Runs: 90

Wickets:2

Overs: 10.2

Current Score Display:

Runs: 90

Wickets:2

Overs: 10.2

(Challenge) State

Jason Fedin

Requirements

- let's create a very simple application that demonstrates the state design pattern
- suppose we want to simulate the alert states of a mobile phone
 - the phone can be in either a vibration state or a silent state
- based on this alert state, behavior of the mobile device changes when an alert happens

Steps

- create the state interface
 - should contain an alert method
- create the context class
 - should have a reference to the state interface
 - should have a setState method and an alert method
 - alert will use the state reference to invoke the correct instance method (polymorphism)
- create two concrete state classes
 - one for vibration and one for silent
 - each alert method can simply display a default message
- create a client class that creates a context object and sets its various states while also issuing alerts

Example Output

vibration...

vibration...

silent...

silent...

silent...

(Challenge) Strategy

Jason Fedin

Requirements

- lets create an application that performs mathematical operations using the strategy design pattern
- create a Strategy interface defining an action and concrete strategy classes implementing the Strategy interface
- the context class will use the strategy
- lastly, create a client that will use Context and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses

Requirements

- your strategy interface can include just one simple method
 - performOperation that takes two numbers as parameters
- the concrete strategies should perform mathematical operations
 - one for addition
 - one for subtraction
 - one for multiplication
- the context class will execute the correct strategy based on what the client wants
 - clients can pass strategy to a method or the context can contain a reference to the strategy (set in the constructor by the client)

Requirements

- create a client class that creates one or more contexts (depends on how you are telling the context what strategy to use)
- create multiple strategies that the context will use to perform mathematical operations

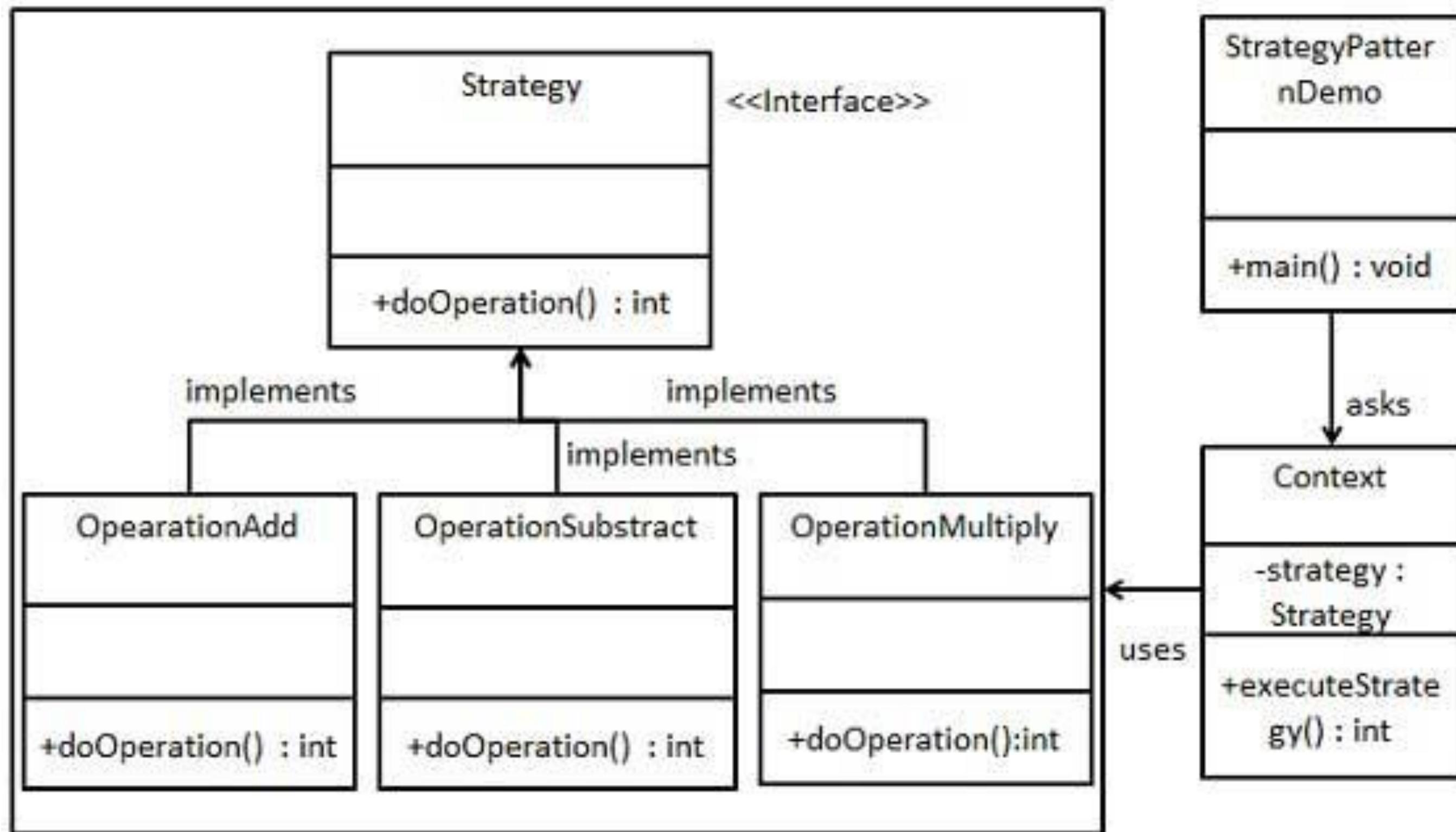
Example output

$10 + 5 = 15$

$10 - 5 = 5$

$10 * 5 = 50$

Example Class Diagram



(Challenge) Template method

Jason Fedin

Requirements

- this application has to do with order processing flow
- create an application that allows clients to order items and process them by paying either online or in the store
- use the template method to determine payment
 - online or in-store
- the overall algorithm used to process an order is defined in the base class and used by the subclasses
 - but the way individual operations are performed vary depending on the subclass

Requirements

- create a abstract class containing the algorithm skeleton (OrderProcessTemplate)
 - the processOrder() is the method that contains the process steps
 - doSelect, doPayment, and delivery are the abstract methods
 - determine how the item is selected (added to an online cart or picked from a shelf)
 - determine how the item is paid for (using a credit card, paypal, or cash (if picked from a shelf))
 - determine how the item is delivered (postage or picked up at counter)
 - other steps in the algorithm could include giftWrapping
- create two subclasses
 - NetOrder and StoreOrder
 - has the same order processing steps
 - implements abstract methods (can just print behavior)

Create the client

- item creation is not necessary for implementation because we are demonstrating the use of the template method by processing the order
 - you can assume the items have been created
- create some Orders (netOrder or storeOrder)
- process each order to demonstrate the template method pattern

Example output

Item added to online shopping cart

Get gift wrap preference

Get delivery address.

Online Payment through Netbanking, card or Pay pal

Gift wrap successful

Ship the item through post office to delivery address

Customer chooses the item from shelf.

Pays at counter through cash/POS

Gift wrap successful

Item delivered to in delivery counter.

(Challenge) Visitor

Jason Fedin

Requirements

- Lets create another type of shopping cart application that demonstrates the visitor pattern
- this application will calculate the cost of postage for shipping items
- our set of elements will be the items in our shopping cart
- postage will be determined using the type and the weight of each item, and of course depending on where the item is being shipped to (what region)
- you need to create a separate visitor for each postal region
 - allows us to separate the logic of calculating the total postage cost from the items themselves
 - means that our individual elements do not need to know anything about the postal cost policy
 - nicely decoupled from the object structure and element logic

Steps

- create the Element interface
 - should have an accept method that takes a Visitor
- create three concrete element types that each contain a price and weight as member variables and implement the accept method
 - Book, CD, and DVD
 - accept method just calls visit method, passing in itself
 - does not need to know anything about the postal cost policy
 - nicely decoupled from visitor logic

Steps

- create the visitor interface
 - three visit methods for each type of element
- create two types of visitors (one for inside the US and one for South America)
 - USPostageVisitor and SouthAmericaPostageVisitor
 - each class should contain state data for the total postage for a cart of items
 - each visit method should contain the algorithm for calculating the postage for a item based on the weight of that item
 - for a book, algorithm should be weight * 2
 - for a cd, algorithm should be weight * 2.5
 - for a dvd, algorithm should be weight * 3
 - if outside the us, double the postage cost
 - for all items, if the price is greater than 20 dollars (inside US) or price is greater than 30 dollars (South America), allow for free shipping

Steps

- create a client
- create many visitable items (a couple books, cds, and dvds with different weights and prices)
- create a list and add the above items to this list
- iterate through the list to calculate total regional postal costs for two visitors
 - create two visitors (US and South America)
- display the regional cost for shipping all items in the list to a US region and to a South America Region

Output Example

The total postage for shipping my items to the US is: 10.14

The total postage for shipping my items to South America is: 20.28

- if we had other types of items here, once the visitor implements a method to visit that item, we could easily calculate the total postage for each region
- while the Visitor may seem a bit strange at first, you can see how much it helps to clean up your code
 - the whole point of this pattern
 - to allow you to separate out certain logic from the elements themselves, keeping your data classes simple