Introduction to Design Patterns

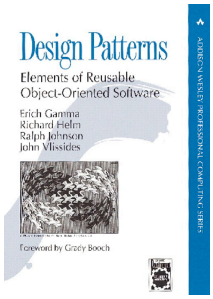
- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What are Design Patterns?

- Software patterns are re-useable designs of how objects can work together in some fashion
- Using patterns helps save you from re-inventing the wheel
- The pre-eminent book on patterns is "Design Patterns, Elements of Reusable Object-Oriented Software" by Gamma, et. al.



In this chapter, we will study some of the more popular patterns from the Gang of Four book. However, you should be aware that there are many other patterns. For example, in the Java server-side arena, there are several very useful patterns described in the "Core J2EE Patterns: Best Practices and Design Strategies" by Alur, et. al.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Pattern Categories

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

These are the categories of patterns from the Gang of Four book.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Creational Patterns

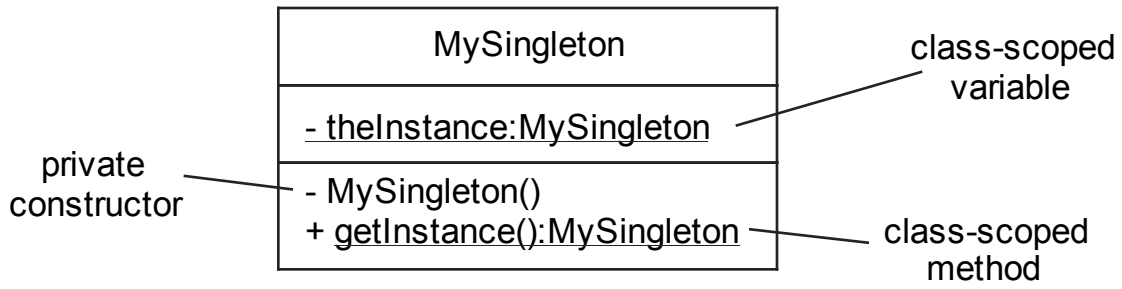
Pattern	Description
Abstract Factory *	Decouples the client from the code that instantiates objects
Builder *	Separates the construction of a complex object from its representation
Factory Method	Defers instantiation to concrete subclasses
Prototype	Creates new objects given an example object
Singleton *	Ensures that a class allows only a single instance

* - Covered in this chapter

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Singleton Pattern

- The **Singleton** pattern is useful in scenarios where you want to have a single instance of a class



Singletons pop up in many situations where you need "global" data or methods.

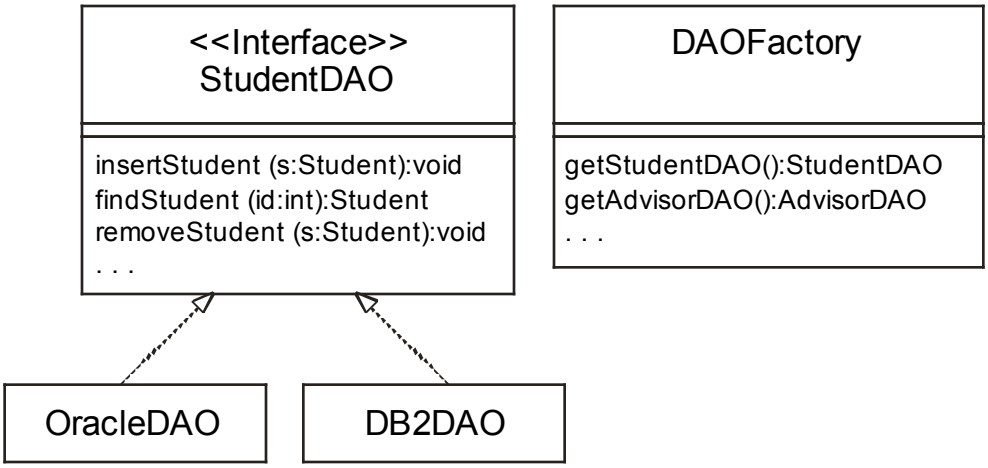
The implementation of Singleton in Java uses features known as "static fields" and "static methods", which reside in the class rather than in the objects that are instances of the class (this implies that the class exists while the program is running!).

Note also that we define the "constructor" as private, which in languages like Java prevents clients from manually creating instances. In other words, the only way to get an instance of the class is to call the "getInstance()" method, which always returns the single instance.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Abstract Factory

- An **Abstract Factory** creates sets of related or dependent objects without requiring "clients" to know the concrete class of the objects

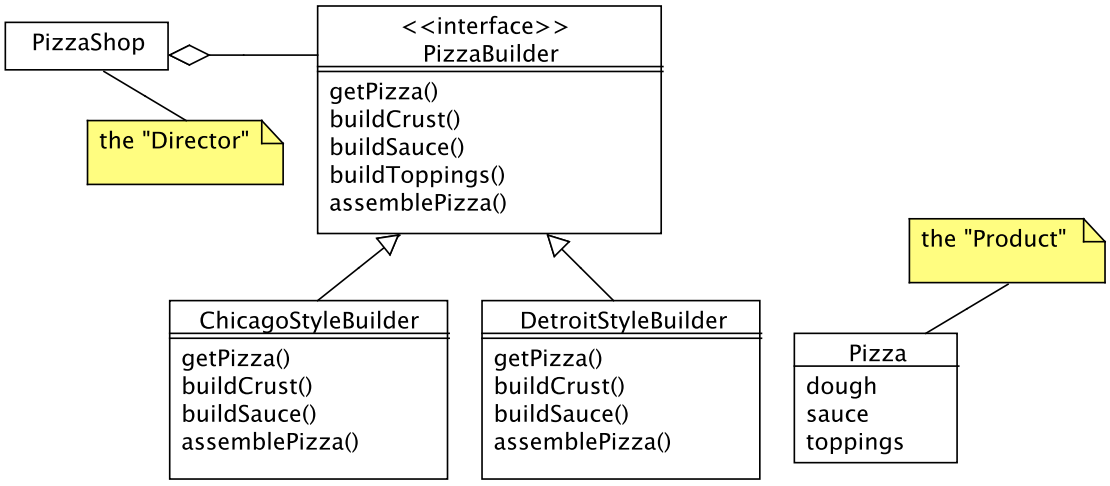


Here we show an example of using the Abstract Factory pattern. Clients that need a "Data Access Object" class instance don't need to know whether the application is using the DB2 or Oracle database. The act of creating the instance is hidden within the DAOFactory class, so if we decide to change databases, the client program doesn't need to be updated.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Builder

- **Builder** is useful if creating a complex object takes several steps AND the object's parts can take different representations
- Builder hides the construction behind an interface and concrete classes for the representations



Builder often is used to create objects that follow the Composite design pattern

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Structural Patterns

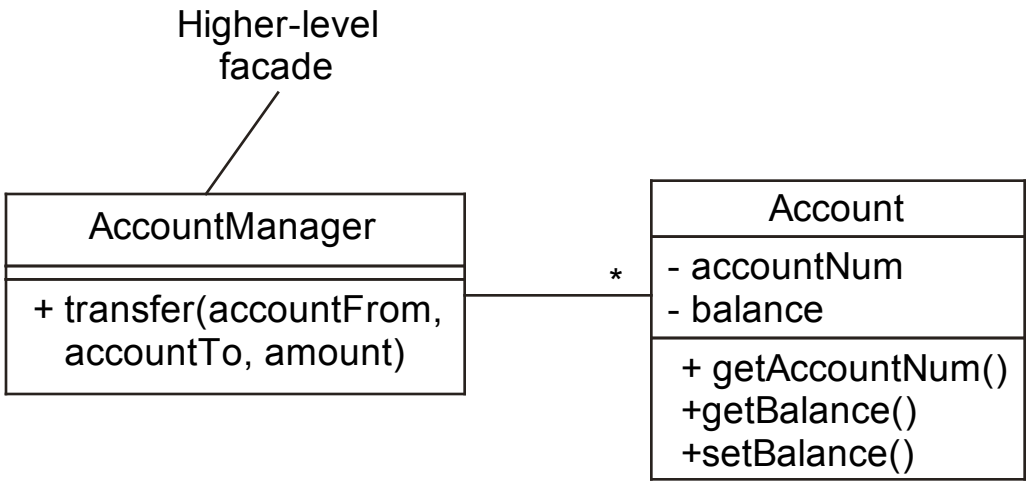
Pattern	Description
Adapter *	Converts the interface of a class into something that clients expect
Bridge	Splits an interface hierarchy from an implementation hierarchy
Composite	Treats aggregates objects as an individual object
Decorator	Lets you add responsibilities to individual objects rather than classes
Facade *	Defines a higher-level interface for a subsystem
Flyweight	Creates lightweight fine-grained objects
Proxy *	Provides a stand-in or representative for an object

* - Covered in this chapter

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Facade Pattern

- In many designs, especially those using remote invocation, it's advantageous to expose a high-level, coarse-grained interface rather than many fine-grained methods



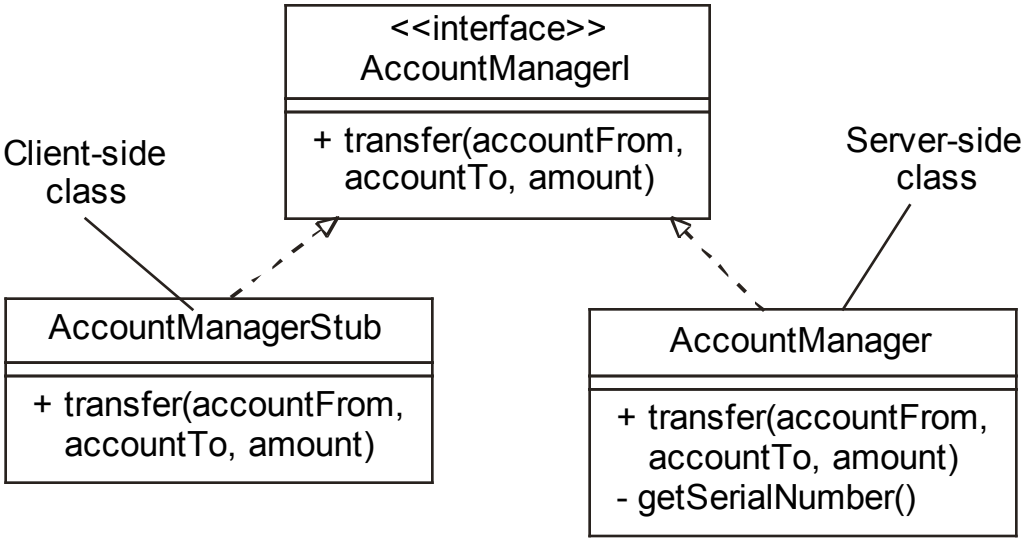
This class diagram shows a fairly trivial but illustrative use of the Facade pattern. Instead of having clients do a balance transfer by calling the primitive get/set methods on individual account objects, instead, the clients call the higher-level facade method named "transfer()".

Not only does this simplify the client's interface to the application, but if the client is running on a different computer than where the AccountManager and Account objects reside, we avoid costly network traffic (the client calls only a single remote method to transfer a balance).

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Proxy Pattern

- In many designs, especially those using remote invocations, it's advantageous to have a "stand-in" object that forwards requests to the actual object



There are serveral cases where using a proxy, or "stand-in" object is useful, including:

1. In distributed computing where a client-side proxy object makes a remote object appear to be local by forwarding local requests over the network.

2. So-called "virtual proxies", where a stand-in object acts as a minimal object until a method on the object is actually called. Useful for resource-intensive objects where you don't want to completely instantiate the object until it's really needed.

3. Access proxies, that can perform authorization to control access to a sensitive object's methods.

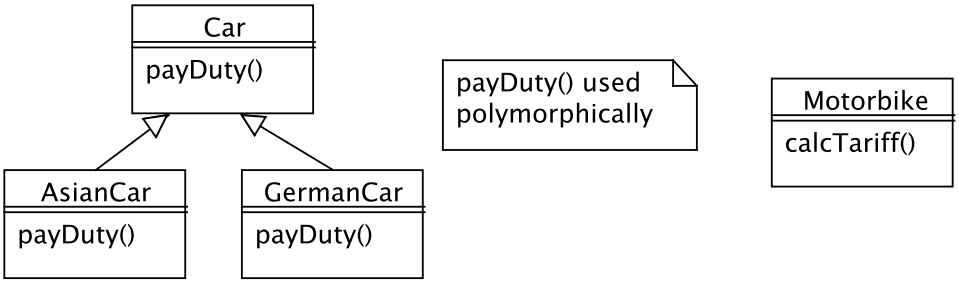
4. Aspect-oriented programming (AOP), where proxies intercept calls to objects and perform additional processing such as transactions, logging and so forth.

The example shown here is for a remote proxy. Here we define an interface that both client and server-side classes implement. The server implementation is the "real" code, while the client-side implementation simply forwards requests to the remote object.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Adapter Pattern

- The **Adapter** pattern lets you adapt existing code to a different interface
- Example: Suppose we have this model:



- Assume we need to add an existing Motorbike class to be used by Car's polymporphic algorithms
- Assume we cannot modify Motorbike and it does **not** extend Car

---

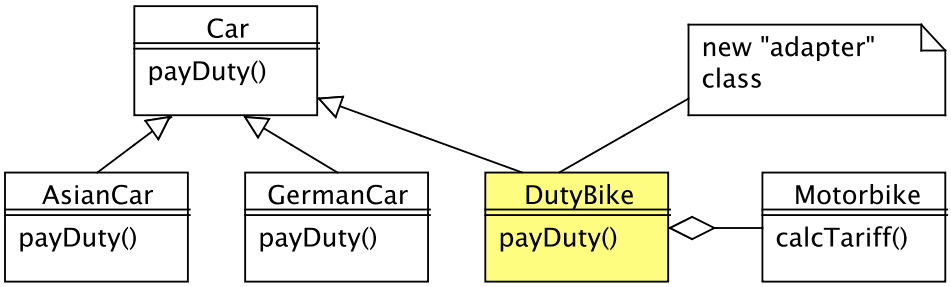
The key point here is that Motorbike is an existing class that we wish to "shoe-horn" into our hierarchy, but it doesn't derive from Car, nor does it provide a payDuty() method.

This might be the case if the Motorbike class was provided by a vendor or by a different department in our organization.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Adapter Pattern, cont'd

- Applying **Adapter**:



- The new **DutyBike's** payDuty() delegates to **Motorbike's** calcTariff()
- Now we can use DutyBike in polymorphic algorithms
- This is not a pleasing OO design, since we cannot say that **DutyBike** "is a" Car, but it's better than rewriting the whole hierarchy

---

The key is the new DutyBike class that adapts the immutable Motorbike class to our existing hierarchy.

This is a common trend of many Design Patterns -- using delegation.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Behavioral Patterns, Part 1

Pattern	Description
Chain of Responsibility	Gives each object in a list the opportunity to handle a request
Command *	Creates an object to represent a request
Interpreter	Allows you to define a command language and grammar and the code that executes it
Iterator *	Defines an object that can traverse through a data collection
Mediator	Encourage loose coupling by defining an object that controls the interaction of other objects
Memento *	Save and restore an object's state without breaking encapsulation

* - Covered in this chapter

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Behavioral Patterns, Part 2

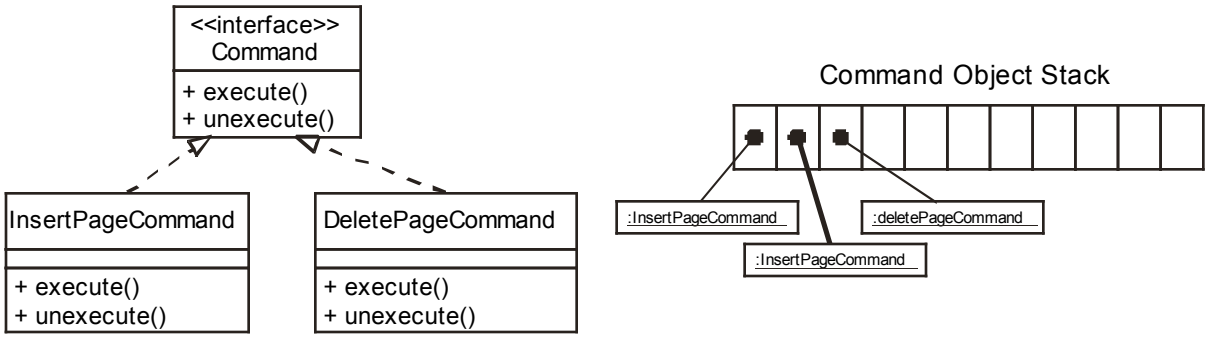
Pattern	Description
Observer *	Allows multiple objects to monitor a single object's state
State	Lets you define objects whose behavior changes depending on the object's mode
Strategy *	Encapsulates algorithms so that clients are insensitive to changes in the algorithm
Template Method *	Defines algorithms that are partially implemented by subclasses
Visitor	Separates the logic that operates on elements in a collection from the collection itself

* - Covered in this chapter

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Command Pattern

- The **Command** pattern encapsulates requests as objects
- This pattern also allows for a straightforward implementation of an "undo" facility



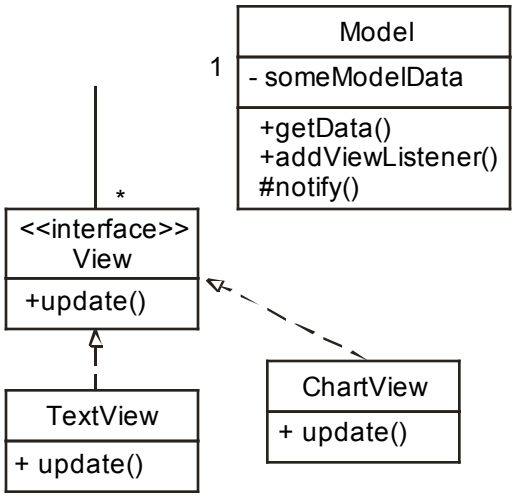
Here we show using the Command pattern to implement a stack of un-doable requests. As the user makes a request (e.g. via a GUI menu), we create the appropriate Command object and put it at the end of the command-object stack and then call the execute() method. If the user wants to undo the last command, we "pop" the last command off of the stack and then call its unexecute() method.

In J2EE, the Command pattern is often enhanced into the "Front Controller" pattern, where a single object routes all requests to the objects that actually execute the requests, which are encapsulated as objects).

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Observer Pattern

- The **Observer** pattern defines dependencies between a subject object and other objects, so that when the subject object's state changes, the other objects are notified



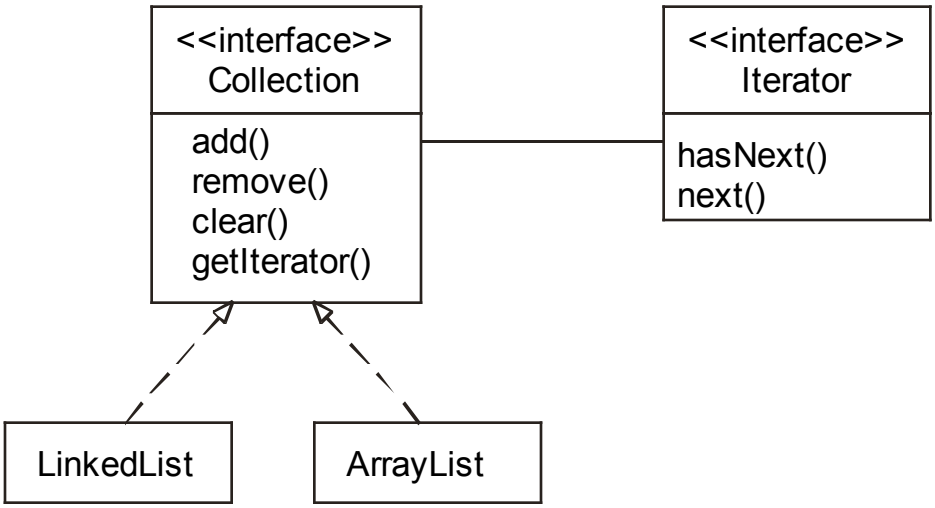
Here we show a use of the Observer pattern. We have a class named Model that contains the data of an application. It also maintains a list of View objects that are interested in changes in the Model's state. The View objects display the Model's data in some fashion. In this case, we have two types of Views -- one that presents data in textual form, the other that charts the data.

The View interface defines a method named "update" -- when the Model detects a state change, it traverses through its list of registered observers (Views) and calls the update() method. Thus, both views update their display when the Model changes.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Iterator Pattern

- The **Iterator** pattern decouples the act of traversing a collection from the collection itself
- This standardizes access across different types of collections and avoids exposing the collection's internal structure



Many class libraries include the notion of Collection classes that maintain groups of objects. There are several ways to code such collections using classic data structures such as arrays and linked lists, each of which has different memory and performance characteristics.

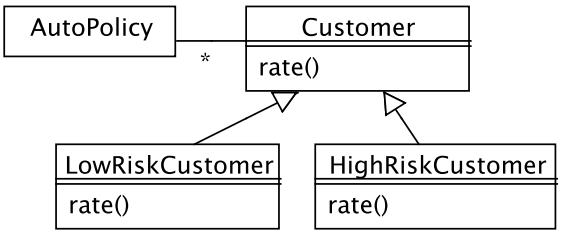
Using a separate Iterator object allows users of collections to access all collections in the same fashion, and also doesn't expose the actual data structure -- that way, you can change the collection type (e.g. from ArrayList to LinkedList), without having to modify any code that processes the objects in the collection.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Strategy Pattern

- The **Strategy** pattern lets you define families of algorithms and insulate the classes that use the algorithms from changes
- Strategy also lets you dynamically change how algorithms are applied at runtime

- Example: This model uses polymorphism, but subclasses don't do anything but calculate a rate (cohesive?)
- How could we change the risk level at runtime?



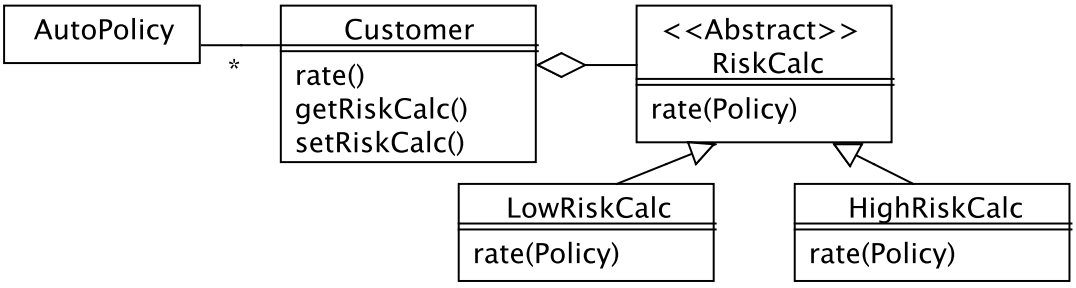
Sometimes you may find yourself designing systems that use inheritance solely so you can create polymorphic algorithms. While such designs are good conceptually, they become difficult to manage if you end up with a large number of subclasses.

But perhaps the largest shortcoming is that we cannot easily change a given Customer's risk value "on the fly" -- we have to somehow change its class, which is difficult in most OO languages.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Strategy Pattern, cont'd

- Applying **Strategy**:



- Now **Customer** is cohesive and insulated from future changes in risk calculation
- The **RiskCalc** subclasses are cohesive, simple and easy to write, test and maintain
- Plus, we can now change how a **Customer** calculates risk "on the fly"

Like in other patterns, Strategy uses delegation to create a more maintainable design. When AutoPolicy calls rate() on a Customer, it defers the calculation to its companion RiskCalc object by calling its rate() method.

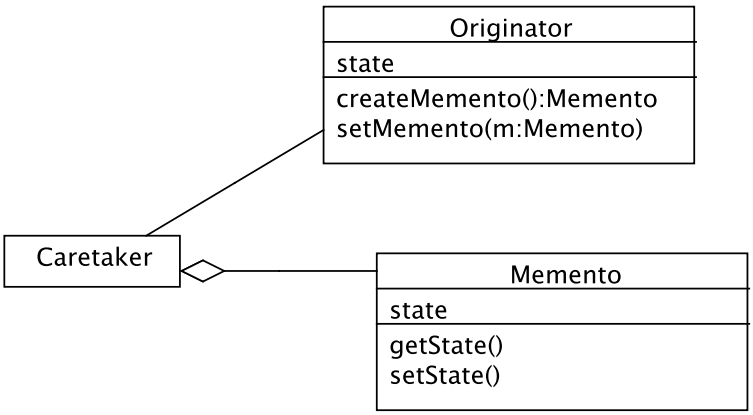
To change RiskCalc on the fly, create an appropriate RiskCalc subclass object and then call setRiskCalc() on the Customer.

Note that if the RiskCalc class itself has no concrete state or behavior, you could define it as an interface instead.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Memento Pattern

- The **Memento** pattern lets the outside world (known as a *Caretaker*) capture and externalize an object's state so it it can revert the object in the future
- The state capture and restore must not violate encapsulation



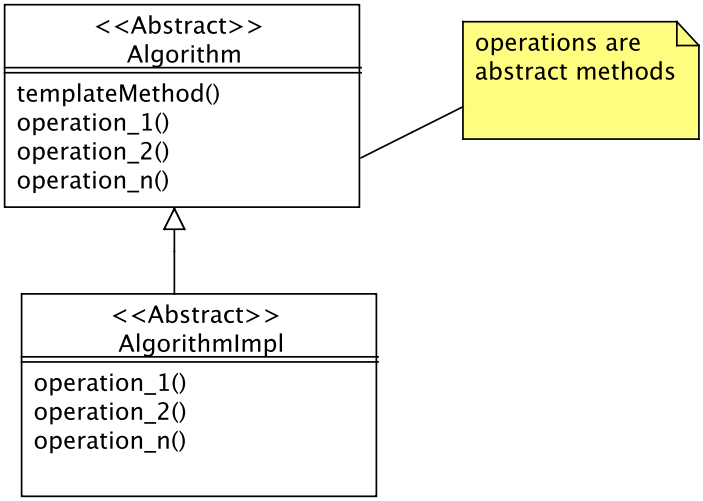
Memento is useful for applications with stateful objects where you want to support an "undo" facility. Before performing an operation, the application could use Memento to capture the object's state, and then if undo is required, use the Memento to restore the object back to its state before the operation.

To avoid violating encapsulation, the Caretaker should NOT have access to the Originator's private state. Note, however, that the Memento needs that access. For that reason, in Java, Memento classes are often implemented as "inner classes" since they have unfettered access to the "outer" class's private state.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Template Method Pattern

- In the **Template Method** pattern, you define the skeleton of a multi-step process in a method, but defer the implementation of the steps to subclasses
- This lets you redefine how the process actually works by replacing the subclasses



If you have a process, or algorithm that takes several well-defined steps, and the execution of the steps themselves may vary, this pattern lets you encapsulate the variation in subclasses that provide the operations.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Review Questions

- Why is the Facade pattern useful in distributed systems?
- Why is the Observer pattern useful in a GUI-based application?

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About how design patterns save you from re-inventing the wheel
- The basics of several popular and useful patterns

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum