

Object-Oriented Design

Marek Rychlý

`rychly@fit.vutbr.cz`

Brno University of Technology
Faculty of Information Technology
Department of Information Systems

Information Systems Analysis and Design (AIS)
21 November 2019



- 1 Object-Oriented Analysis and Design
 - Object-Oriented Design Patterns and Principles
 - Responsibility-Driven Design (RDD)
 - Object-Oriented Design Activities
- 2 Gang of Four (GoF) Design Patterns
 - Creational Patterns
 - Structural Patterns
 - Behavioural Patterns
- 3 General Responsibility Assignment Software Patterns (GRASP)
 - Information Expert, Creator, Controller
 - Low Coupling, High Cohesion, Polymorphism
 - Pure Fabrication, Indirection, Protected Variations; Law of Demeter

Object-Oriented Analysis and Design

- Object-oriented analysis to analyse requirements.
(UP Inception/Elaboration phases; a use-case model, domain model)
 - It involves studying the problem to be solved
 - and identifies **what** the problem is.
(without addressing how it will be solved)
- Object-oriented design to design architecture and implementation.
(UP Elaboration/Construction; an executable architecture baseline, design model)
 - It transforms a problem into a particular solution,
 - and identifies **how** the problem will be solved.
(a conceptual solution rather than an implementation, “think first, code later”)
- It is object-oriented, we identify/describe objects, their operations.
(contrary to the structured analysis/data-driven design which deals with data, data-flows, and their processing as described in ER and DFD models)

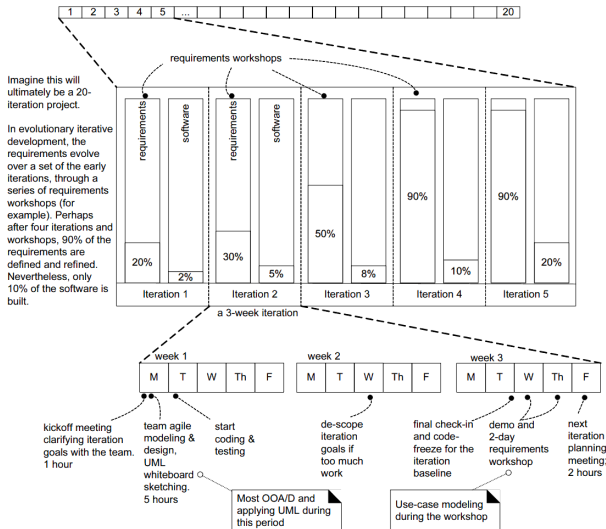
Object-Oriented Design

- OO design is about assigning responsibilities to objects.
(what should they do and where, how should they interact, etc.)
- In OO design, we need to
 - identify classes of the objects,
(described in class diagrams and others)
 - describe interactions of the objects,
(in interaction diagrams and others)
- ... in accordance with responsibilities of the objects.
(from behaviour and limitations given by functional&non-functional requirements)
- It is critical to correctly apply design principles and patterns.
(to follow best-practices in object-oriented software design)

How to Design

- ❶ Design during implementation/coding with refactoring, etc.
(UML models can be generated from the source code by reverse engineering tools)
 - ❷ Design/think first, implement/code later.
(in agile development, these two steps are iterated quickly for small increments)
 - ❸ Just design, without the implementation/coding.
(Model-Driven Development/MDD; still, there must be some coding to fill in gaps)
- Design phase should be fast, models should be “just barely good enough”.
(just few hours/one day design in a three week iteration, according to Larman)
 - It is important to design critical parts, to understand, not to document.
(object operations and responsibilities; see Agile Modelling practices)
 - Design models are an inspiration for following implementation/coding.

Design and Code in Iteration



(adopted from "Applying UML and Patterns" by Craig Larman)

Responsibilities in Object-Oriented Design

- Responsibility-Driven Design (RDD)

- 1 Model requirements as use-cases, etc.
(a system sequence diagram, operation contracts, etc.)
- 2 Create a domain model.
(the subject domain without describing the software implementation)
- 3 Identify responsibilities for each step in use-case scenarios.
(the responsibilities in terms of the system's operation)
- 4 Assign the responsibilities to domain/design objects.
(domain objects as participants of the operations and new design objects)

doing to itself, initiating actions of others, controlling/coordinating others
(from interaction diagrams – operations/messages in object communication)

knowing about private data, related objects, things to derive/calculate
(from a domain model – software domain objects represent a knowledge domain)

- Low Representational Gap (LRG), Domain Driven Design (DDD)
(an application's design should have just very few differences with the domain logic it represents; domain&design models should be similar, in domain/design objects)

Class Responsibility Collaborator (CRC) Cards

- A design model proposed by Beck&Cunningham in 1989, recommended for agile development by Scott W. Ambler.
- To determine classes/objects that are needed & how they interact.
(each card contains a class name, super/sub-classes, responsibilities in doing/knowing, and objects/classes collaborating to fulfil the responsibilities)

Class: <i>Book</i>		Class: <i>Librarian</i>	
Responsibilities	Collaborators	Responsibilities	Collaborators
<i>knows whether on loan</i>		<i>check in book</i>	<i>Book</i>
<i>knows due date</i>		<i>check out book</i>	<i>Book, Borrower</i>
<i>knows its title</i>		<i>search for book</i>	<i>Book</i>
<i>knows its author(s)</i>		<i>knows all books</i>	
<i>knows its registration code</i>		<i>search for borrower</i>	<i>Borrower</i>
<i>knows if late</i>	<i>Date</i>	<i>knows all borrowers</i>	
<i>check out</i>			
Class: <i>Borrower</i>		Class: <i>Date</i>	
Responsibilities	Collaborators	Responsibilities	Collaborators
<i>knows its name</i>		<i>knows current date</i>	
<i>keeps track of borrowed items</i>		<i>can compare two dates</i>	
<i>keeps track of overdue fines</i>		<i>can compute new dates</i>	

(adopted from "CSC/ECE 517 Fall 2007/wiki2 5 kq" by NC State University)

Doing and Knowing

in terms of responsibilities, roles, and collaborations

Doing Responsibilities

- Doing something to itself.
(creating an object or doing a calculation)
- Initiating actions in other objects.
(sending messages, calling operations)
- Controlling and coordinating activities in other objects.
(an orchestration)

Knowing Responsibilities

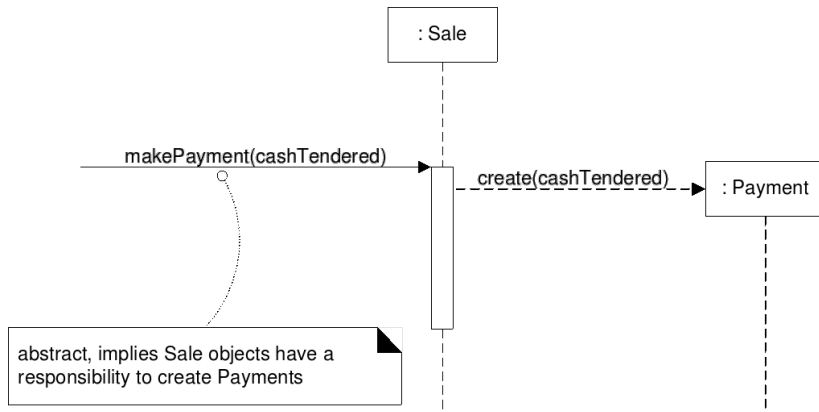
- Knowing about private encapsulated data.
(in an encapsulated variable, private attribute, a method parameter)
- Knowing about related objects.
(from association with the objects, from a collection of the objects)
- Knowing about things it can derive or calculate.
(by a composition of information above, also by cooperation with other objects)

Responsibilities are implemented by operations in objects.

(modelled statically in a class diagram and as collaborations in interaction diagrams)

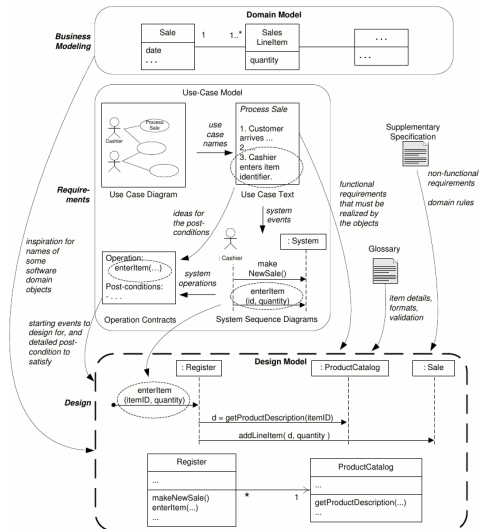


An Example of the Doing Responsibility



(adopted from “Applying UML and Patterns” by Craig Larman)

Sample UP Artefacts in Object-Oriented Design



(adopted from "Applying UML and Patterns" by Craig Larman)

Object-Oriented Design Inputs (1)

The first <i>two-day requirements workshop</i> is finished.	The chief architect and business agree to implement and test some <i>scenarios of Process Sale in the first three-week</i> timeboxed iteration.
<i>Three of the twenty use cases</i> —those that are the most architecturally significant and of high business value—have been analyzed in detail, including, of course, the <i>Process Sale</i> use case. (The UP recommends, as typical with iterative methods, analyzing only <i>10%-20% of the requirements</i> in detail before starting to program.)	<i>Other artifacts</i> have been started: Supplementary Specification, Glossary, and Domain Model.
<i>Programming experiments</i> have resolved the show-stopper technical questions, such as whether a Java Swing UI will work on a touch screen.	The chief architect has drawn some ideas for the <i>large-scale logical architecture</i> , using UML package diagrams. This is part of the UP Design Model.

(adopted from “Applying UML and Patterns” by Craig Larman)

Object-Oriented Design Inputs (2)

<p>The use case text defines the visible behavior that the software objects must ultimately support—objects are designed to “realize” (implement) the use cases. In the UP, this OO design is called, not surprisingly, the use case realization.</p>	<p>The Supplementary Specification defines the non-functional goals, such as internalization, our objects must satisfy.</p>
<p>The system sequence diagrams identify the system operation messages, which are the starting messages on our interaction diagrams of collaborating objects.</p>	<p>The Glossary clarifies details of parameters or data coming in from the UI layer, data being passed to the database, and detailed item-specific logic or validation requirements, such as the legal formats and validation for product UPCs (universal product codes).</p>
<p>The operation contracts may complement the use case text to clarify what the software objects must achieve in a system operation. The post-conditions define detailed achievements.</p>	<p>The Domain Model suggests some names and attributes of software domain objects in the domain layer of the software architecture.</p>

(adopted from “Applying UML and Patterns” by Craig Larman)

Object-Oriented Design Activities

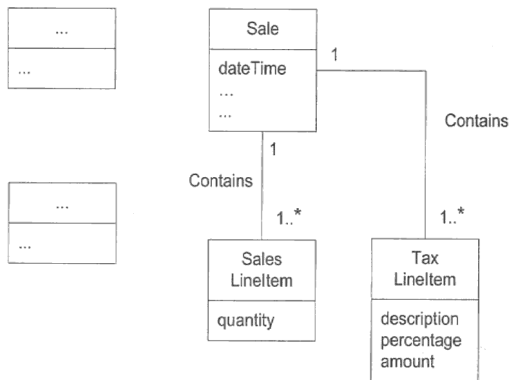
- Coding/implementation, test-driven development (TDD).
(start coding as soon as possible, test first; to maintain a constant “cost of change”)
- Outlining design models as UML diagrams.
(a static model by class diagrams, a dynamic model by interaction diagrams)
- Creating supplementary models (such as CRC cards).

Application of design principles and patterns.

(TDD, RDD, GRASP, dependency inversion principles; GoF patterns)


Emerging Domain Objects during Design

- It may happen – keep the new objects if they will be utilised in design or in communication with stakeholders.
- For example, there is new domain object “TaxLineItem”.



(adopted from “Applying UML and Patterns” by Craig Larman)

Patterns on Object-Oriented Design

- General reusable solutions to commonly occurring SW design problems. (organisation of objects into layers, object interactions, responsibilities, etc.)
- A pattern is usually described by
 - name of the pattern,
 - problem addressed,
 - generalised solution (an abstraction, example),
 - consequences of its application.
- Several sets of the patterns for different phases of the design process. (architectural pattern such as MVC, MVP, PCEMF, etc.; design pattern such as GRASP or from GoF; anti-patterns; etc.)
- Understand the patterns → identify opportunities for their applications.
 - 1 study all the patterns both theoretically and in practical examples; (understand their goals, preconditions, context, and consequences)
 - 2 try to apply them as much as possible, look for every opportunity; (get “a sense of smell” for when the patterns should or should not be applied)
 - 3 learn to recognise when the patterns are necessary and when they would be just an over-engineering (keep it simple).
(a pattern application should simplify something, e.g., future development)

Gang of Four (GoF) Design Patterns

“Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides

- To utilise the capabilities and beware pitfalls of OO programming.

Creational:

- Abstract Factory
- Builder
- Factory method
- Prototype
- Singleton

Structural:

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

Behavioural:

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

How to Select the Right GoF Design Patterns

“Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides

- 1 Consider how design patterns solve design problems.
(they help you find appropriate objects, determine granularity, specify interfaces)
- 2 Read through each pattern's intent.
(to find one or more that sound relevant to your problem)
- 3 Study how patterns interrelate.
(knowledge of relationships helps direct you to the right pattern/group of patterns)
- 4 Study patterns of like purpose.
(creational, structural patterns, behavioural patterns have different purposes)
- 5 Examine a cause of redesign.
(look at the patterns that help you with redesign, analyse the causes of redesign)
- 6 Consider what should be variable in your design.
(consider what you want to be able to change without redesign, focus on encapsulating the concept that varies in the patterns)

How to Use a Particular GoF Design Pattern

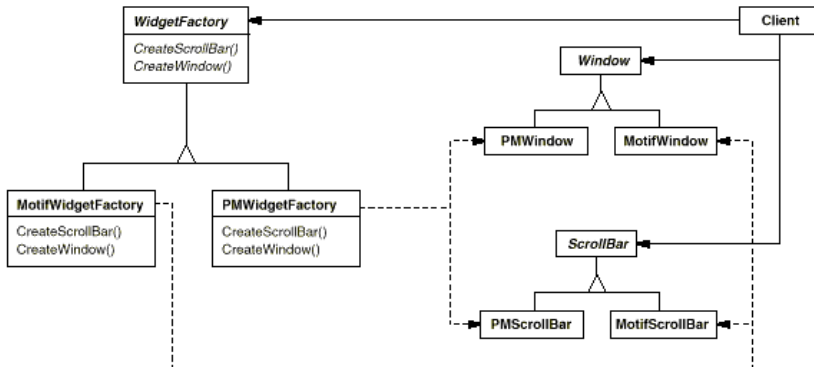
“Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides

- 1 Study the pattern, its structure, participants, and collaborations.
(study its applicability and consequences to ensure it is right for your problem; understand the classes&objects in the pattern and how they relate to one another)
- 2 Look at the sample code to see a concrete example of the pattern.
(it helps you learn how to implement the pattern)
- 3 Name pattern participants in your application context.
(names for participants in design patterns are usually too abstract; keep naming conventions, e.g., use “-strategy” suffix for classes in the Strategy pattern)
- 4 Define the classes of participating objects.
(declare interfaces, inheritance, attributes for data and object references; identify existing classes in your application that the pattern will affect, and adapt them)
- 5 Define application-specific names for operations in the pattern.
(use the responsibilities and collaborations associated with each operation as a guide; keep naming conventions, e.g., use “create-” prefix for factory methods)
- 6 Implement the operations to carry out the responsibilities and collaborations in the pattern.



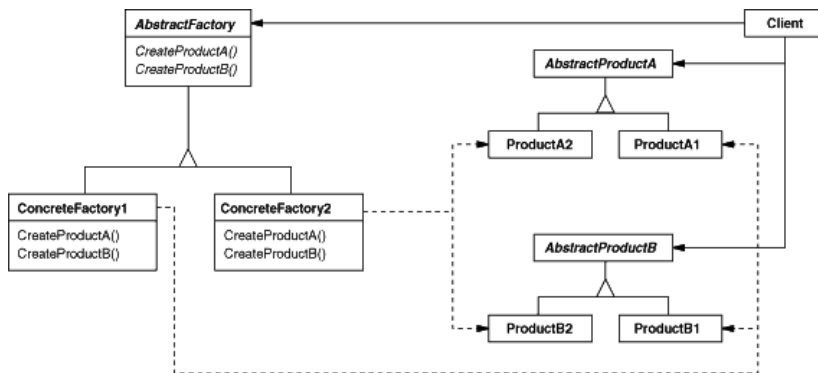
Creational GoF: Abstract Factory Example

To provide an interface for creating families of related or dependent objects without specifying their concrete classes.



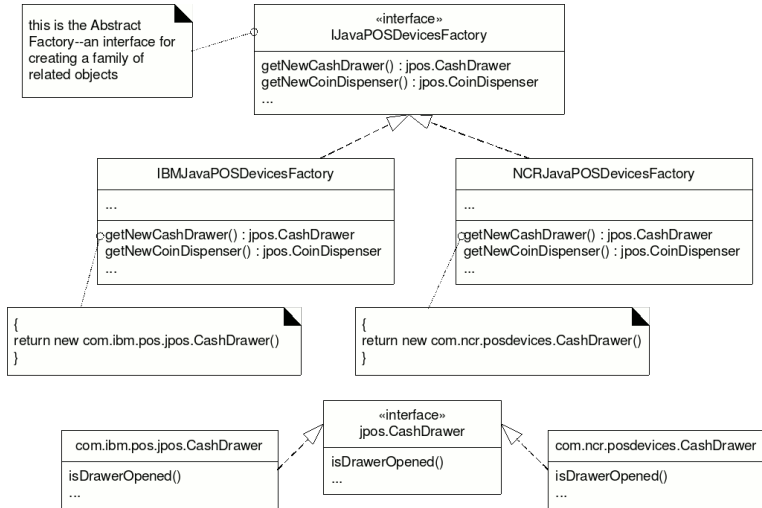
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Creational GoF: Abstract Factory Pattern



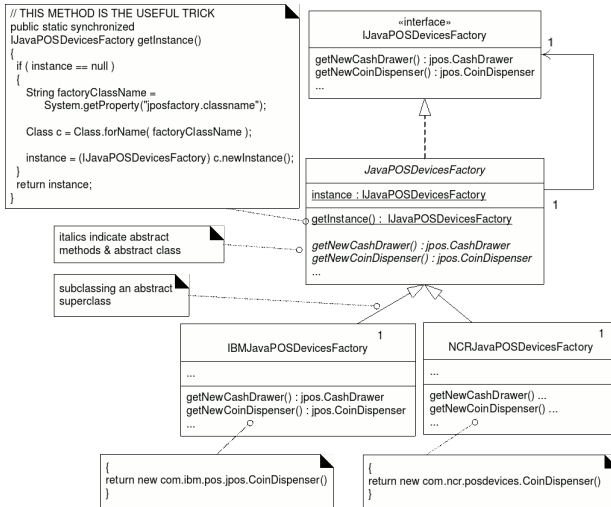
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

An Example: An Abstract Factory in NextGen POS (1)



(adopted from "Applying UML and Patterns" by Craig Larman)

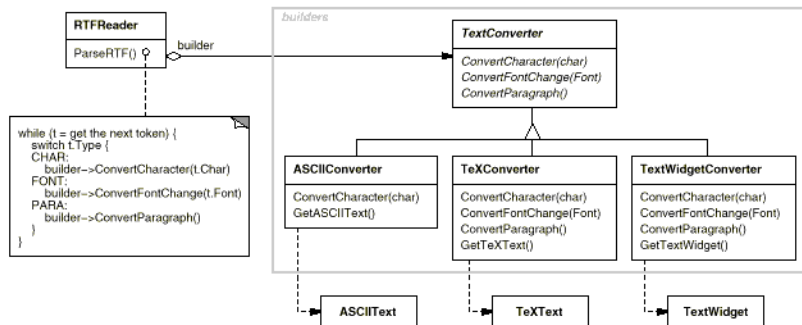
An Example: An Abstract Factory in NextGen POS (2)



(adopted from "Applying UML and Patterns" by Craig Larman)

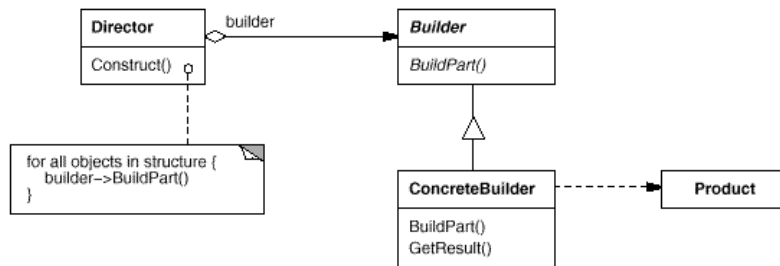
Creational GoF: Builder Example

To separate the step-by-step construction of a complex object from its representation so that the same construction process (an algorithm) can create different representations.



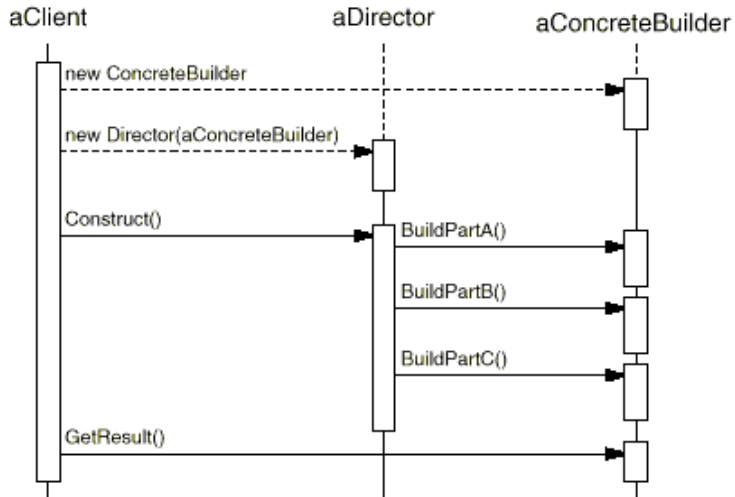
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Creational GoF: Builder Pattern Classes



(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

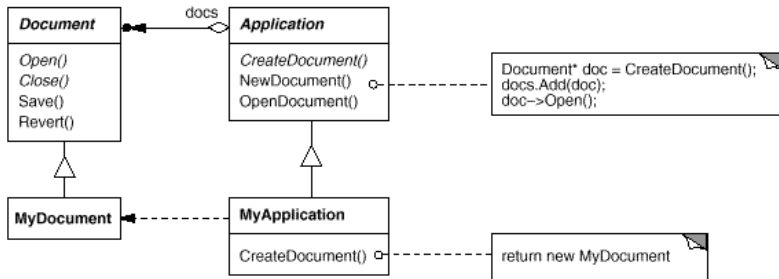
Creational GoF: Builder Pattern Interactions



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

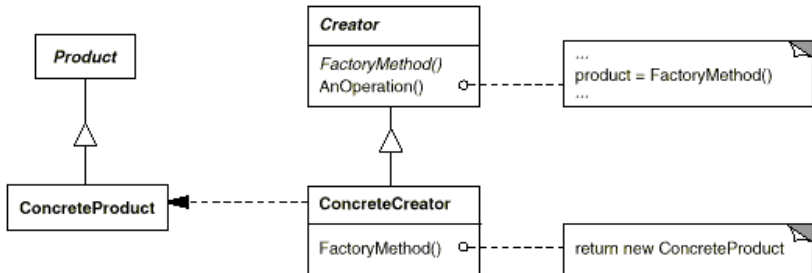
Creational GoF: Factory Method Example

To define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



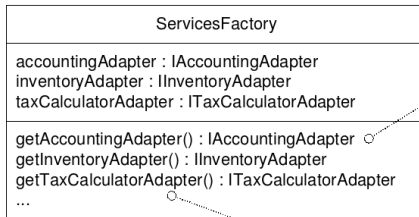
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Creational GoF: Factory Method Pattern



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

An Example: Factories in NextGen POS



note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

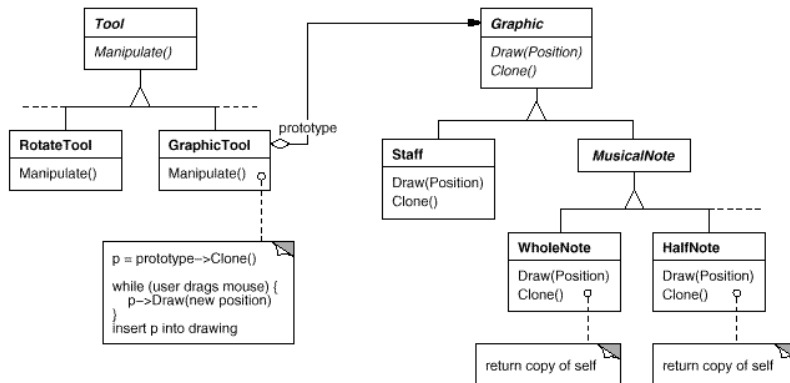
```
if ( taxCalculatorAdapter == null )
{
    // a reflective or data-driven approach to finding the right class: read it from an
    // external property

    String className = System.getProperty( "taxcalculator.class.name" );
    taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName( className ).newInstance();
}
return taxCalculatorAdapter;
```

(adopted from "Applying UML and Patterns" by Craig Larman)

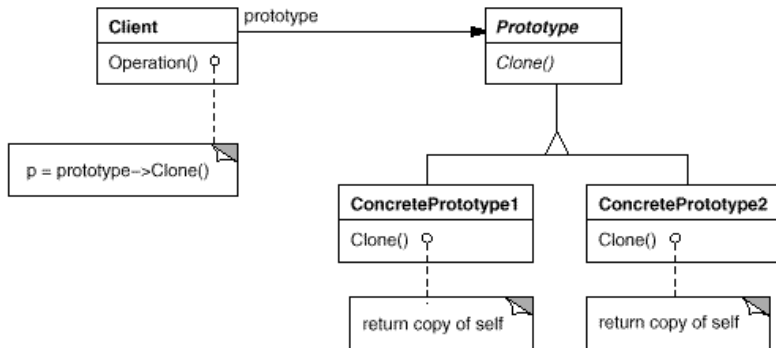
Creational GoF: Prototype Example

To specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

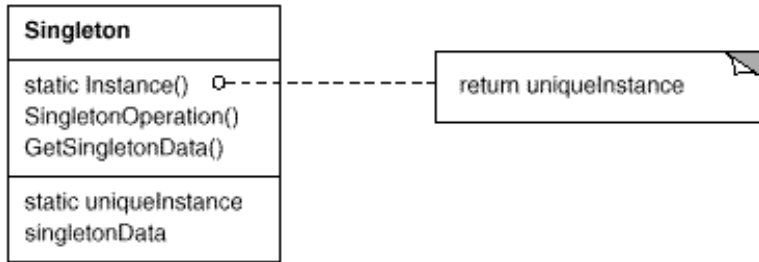
Creational GoF: Prototype Pattern



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

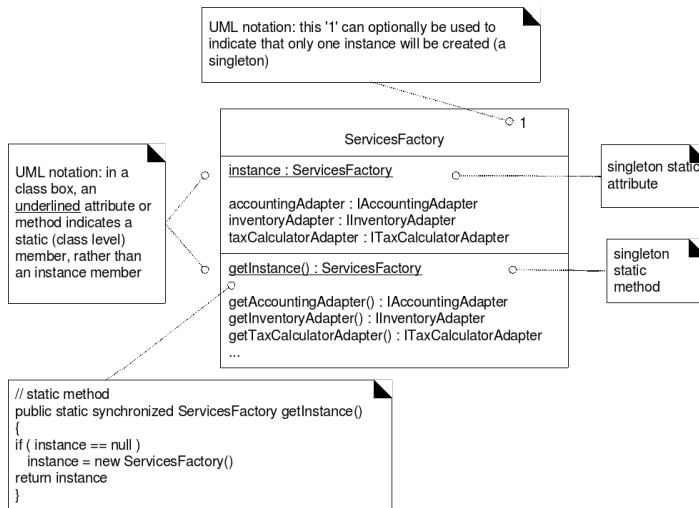
Creational GoF: Singleton Pattern

To ensure a class only has one instance, and provide a global point of access to it.



(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

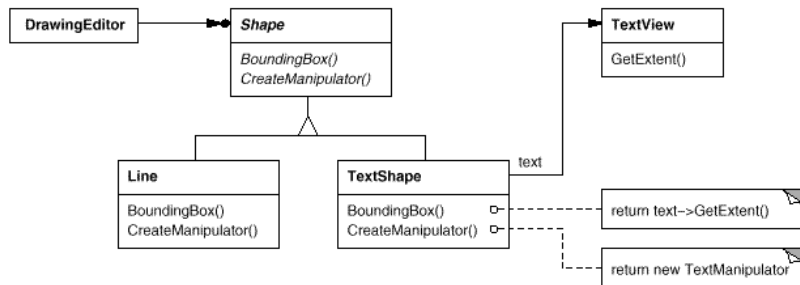
An Example: A Singleton in NextGen POS



(adopted from “Applying UML and Patterns” by Craig Larman)

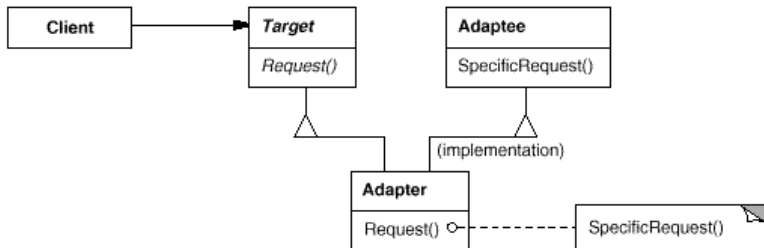
Structural GoF: Adapter Example

To convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.



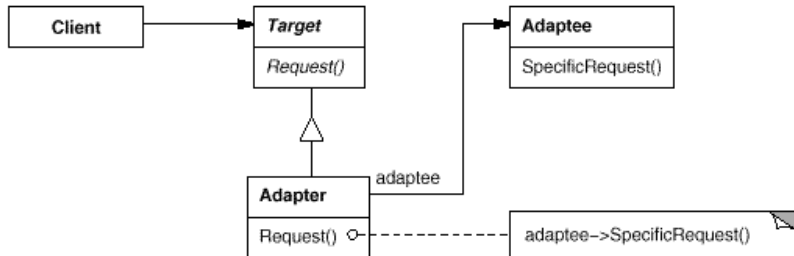
(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

Structural GoF: Adapter Pattern with Inheritance



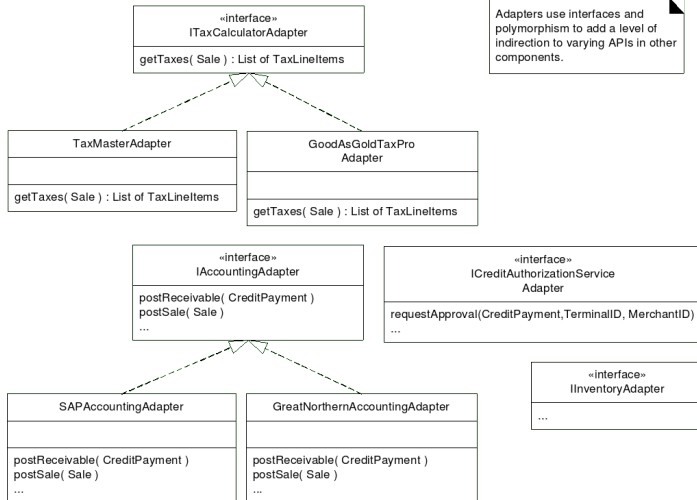
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Structural GoF: Adapter Pattern with Composition



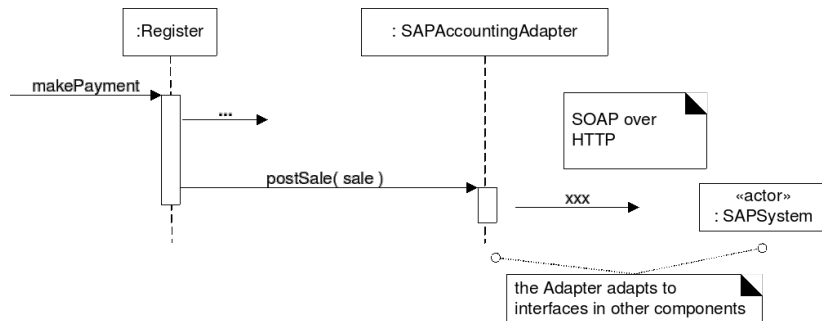
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

An Example: Adapters in NextGen POS



(adopted from "Applying UML and Patterns" by Craig Larman)

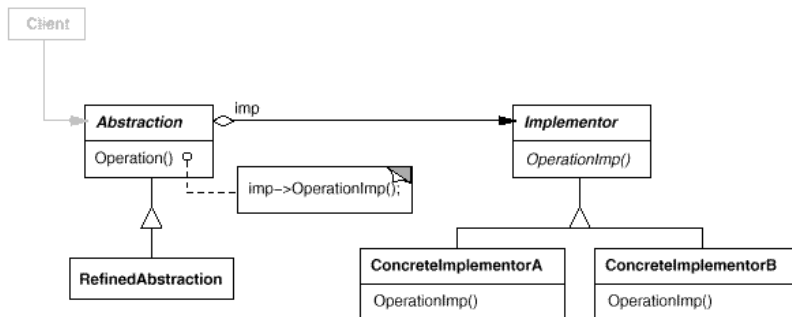
An Example: Interactions with NextGen POS Adapters



(adopted from “Applying UML and Patterns” by Craig Larman)

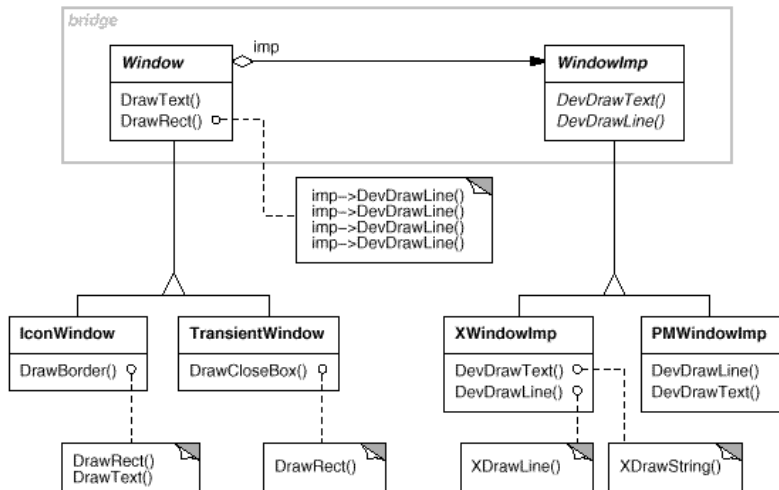
Structural GoF: Bridge Pattern

To decouple an abstraction from its implementation so that the two can vary independently. It is used up-front in a design to let abstractions and implementations vary independently, not to make unrelated classes work together (that is the Adapter pattern).



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

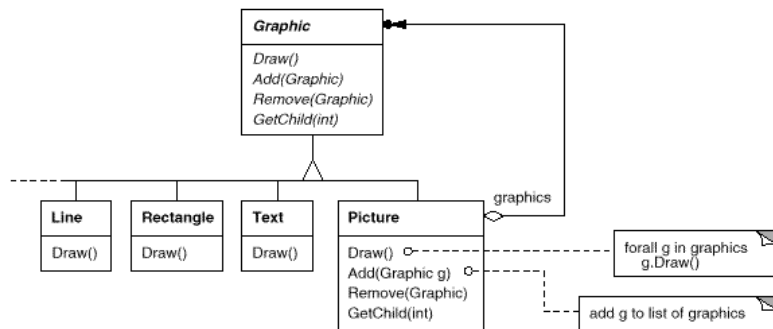
Structural GoF: Bridge Example



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

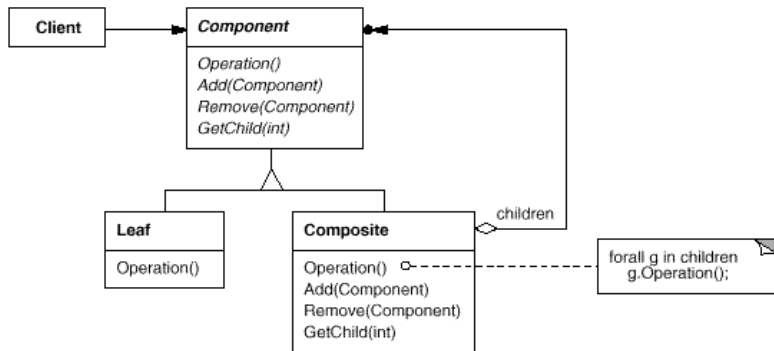
Structural GoF: Composite Example

To compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



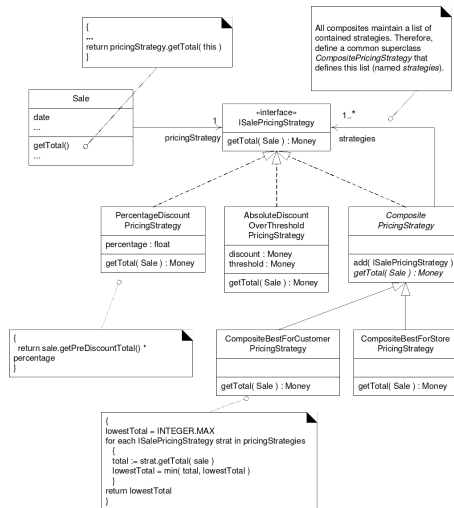
(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

Structural GoF: Composite Pattern



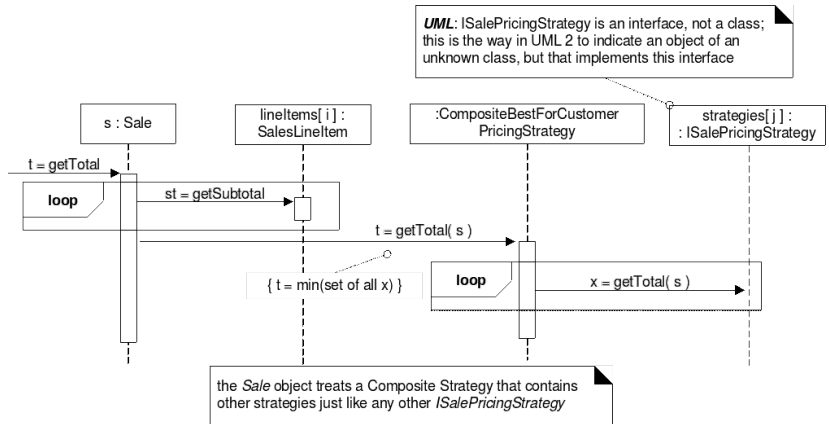
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

An Example: Composite Strategy in NextGen POS (1)



(adopted from "Applying UML and Patterns" by Craig Larman)

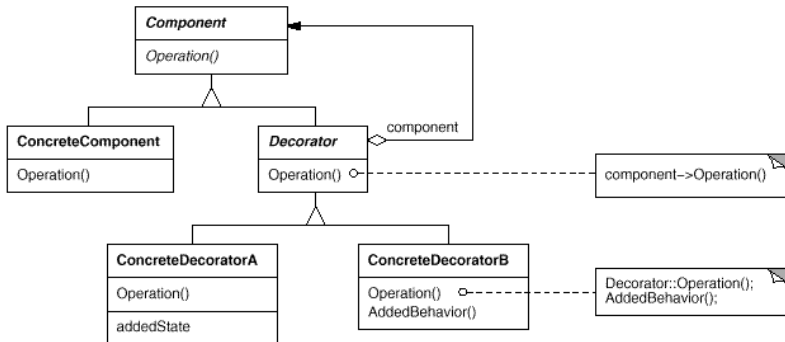
An Example: Composite Strategy in NextGen POS (2)



(adopted from “Applying UML and Patterns” by Craig Larman)

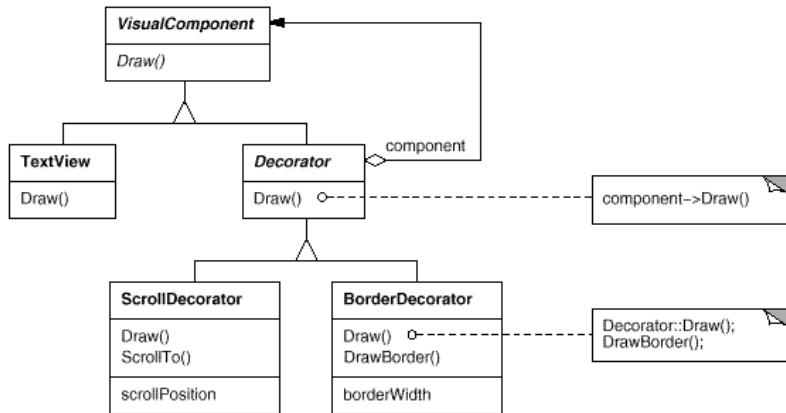
Structural GoF: Decorator Pattern

To attach additional responsibilities to an object dynamically (not to unify the interface as in the case of the Composite pattern). Decorators provide a flexible alternative to sub-classing for extending functionality.



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

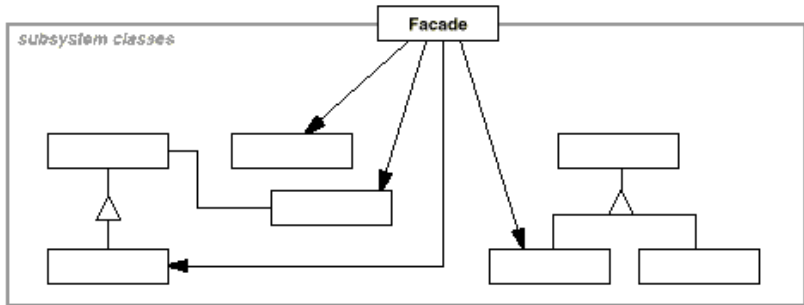
Structural GoF: Decorator Example



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

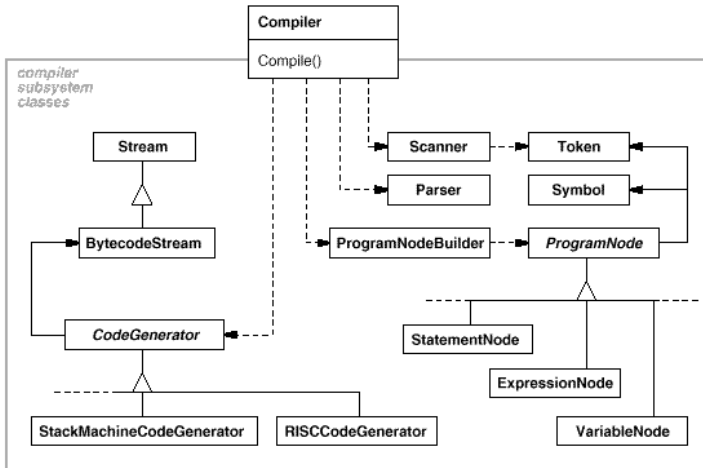
Structural GoF: Facade Pattern

To provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



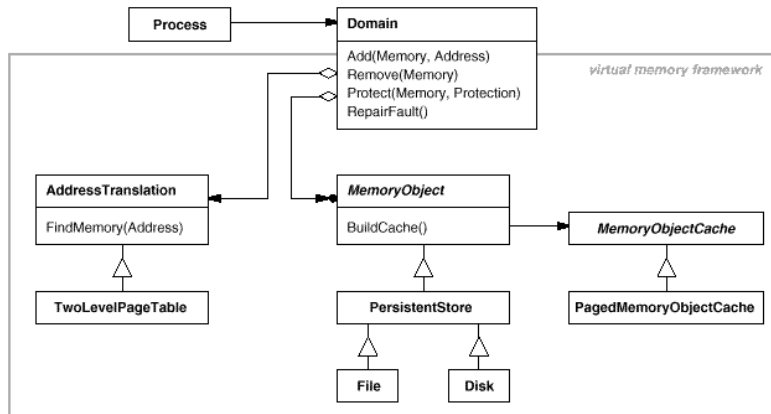
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Structural GoF: Facade Example (1)



(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

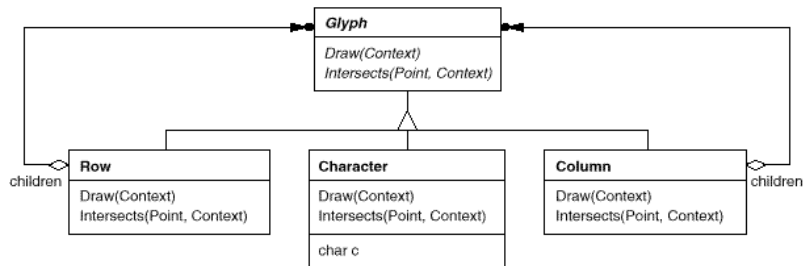
Structural GoF: Facade Example (2)



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

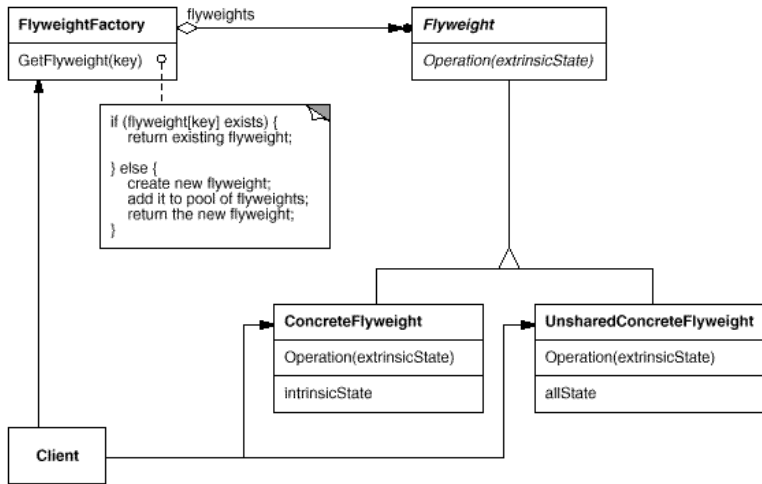
Structural GoF: Flyweight Example

To use sharing to support large numbers of fine-grained objects efficiently (it minimizes memory usage by sharing as much data as possible with other similar objects). The flyweight stores intrinsic (invariant) state that can be shared and provides an interface through which extrinsic (variant) state can be passed in.



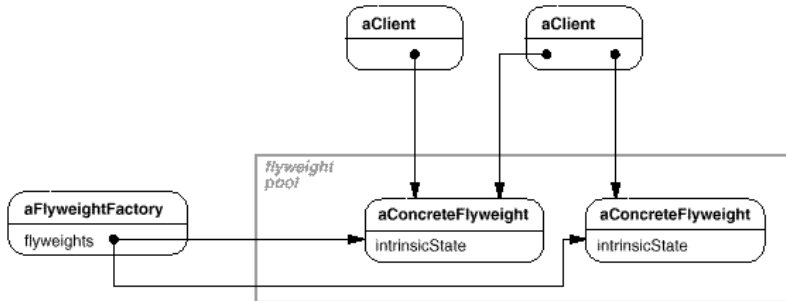
(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

Structural GoF: Flyweight Pattern Classes



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

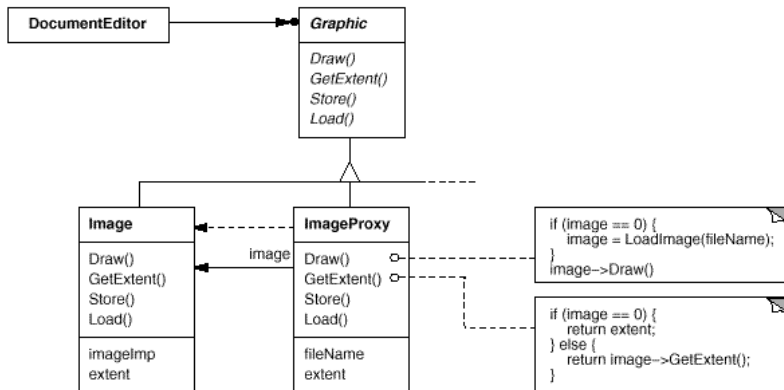
Structural GoF: Flyweight Pattern Objects



(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

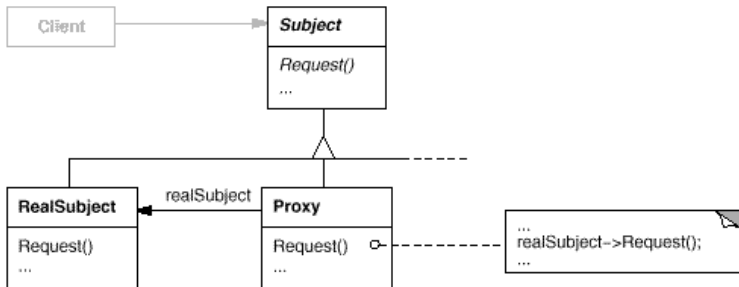
Structural GoF: Proxy Example

To provide a surrogate or placeholder for another object to control access to it.



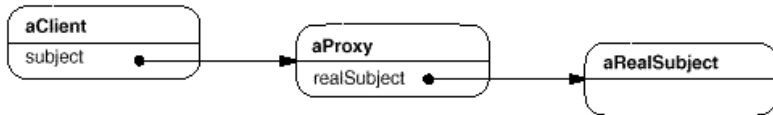
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Structural GoF: Proxy Pattern Classes



(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

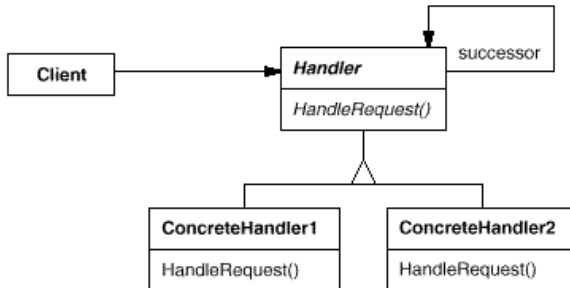
Structural GoF: Proxy Pattern Objects



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

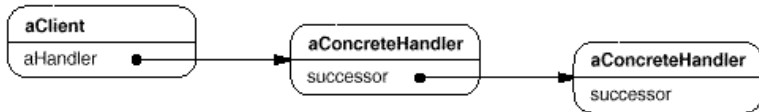
Behavioural GoF: Chain of Resp. Pattern Classes

To avoid 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.



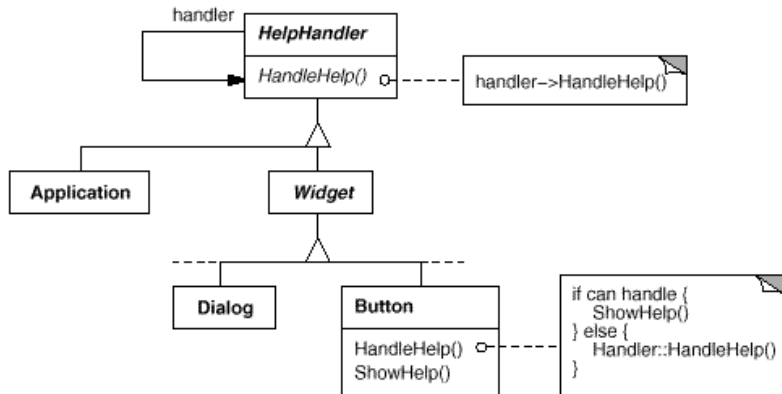
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Behavioural GoF: Chain of Resp. Pattern Objects



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

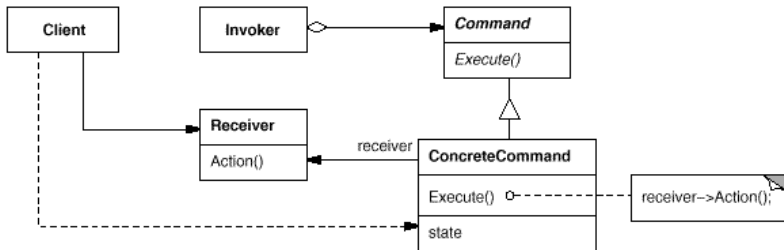
Behavioural GoF: Chain of Responsibility Example



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

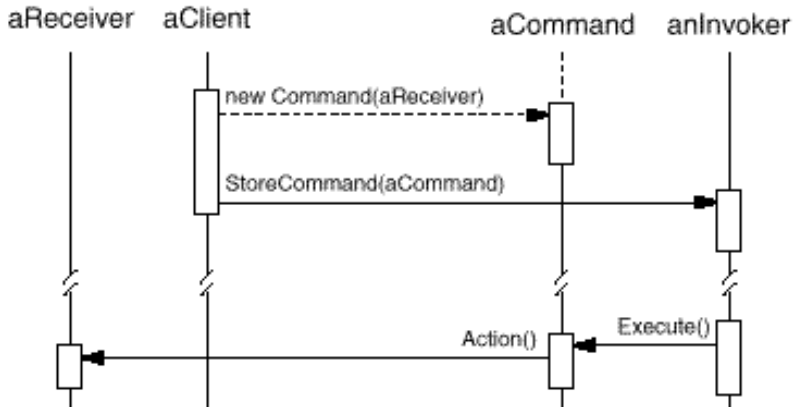
Behavioural GoF: Command Pattern Classes

To encapsulate a request as an object, thereby letting you parametrise clients with different requests, queue or log requests, and support undoable operations.



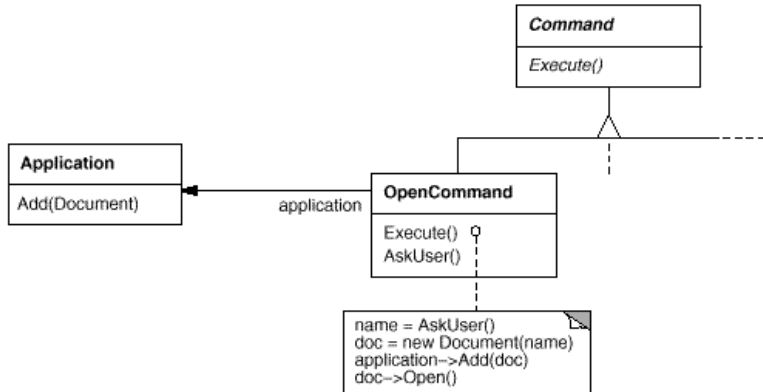
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Behavioural GoF: Command Pattern Interaction



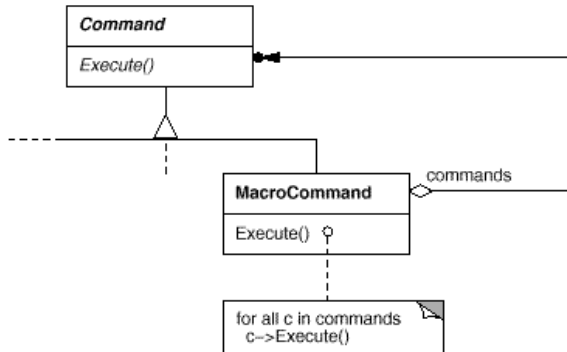
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Behavioural GoF: Command Example



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

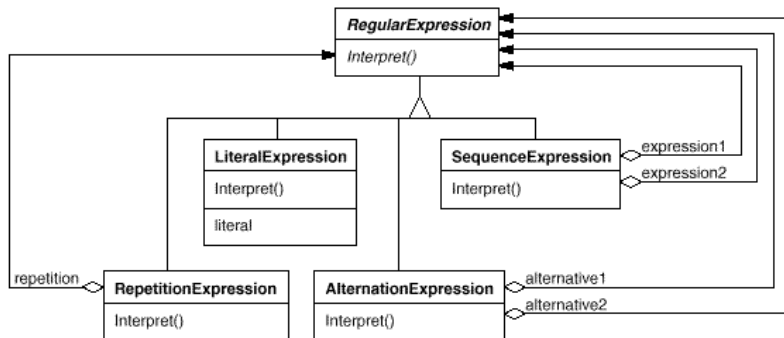
Behavioural GoF: Command Example with Composite



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

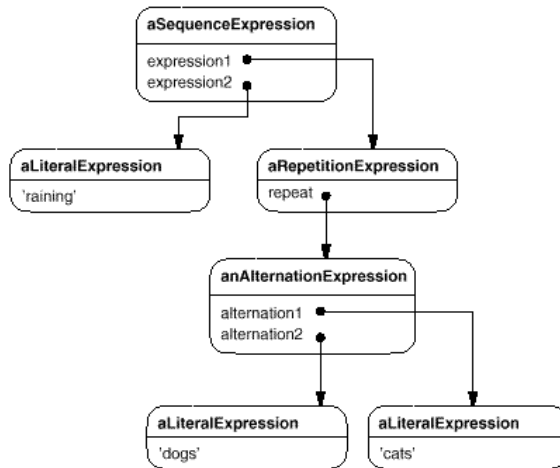
Behavioural GoF: Interpreter Example Classes

To define a representation for a grammar of a given language along with an interpreter that uses the representation to interpret sentences in the language.



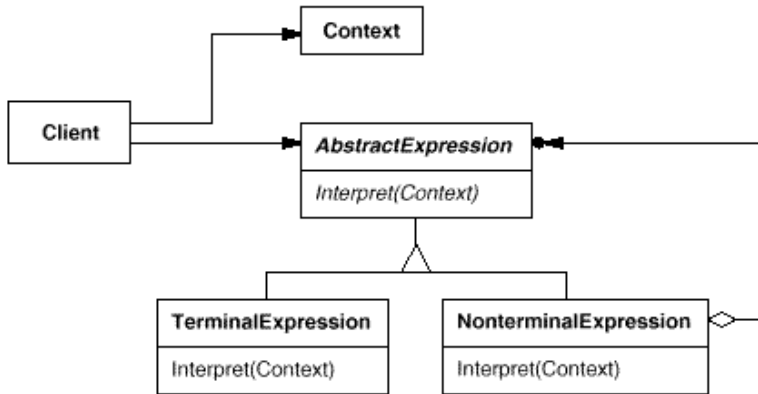
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Behavioural GoF: Interpreter Example Objects



(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

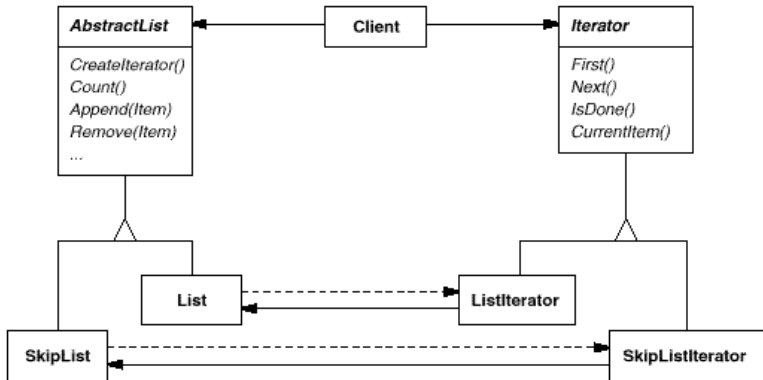
Behavioural GoF: Interpreter Pattern



(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

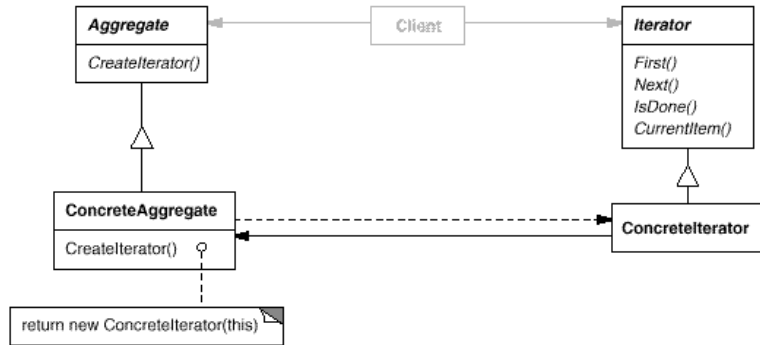
Behavioural GoF: Iterator Example

To provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

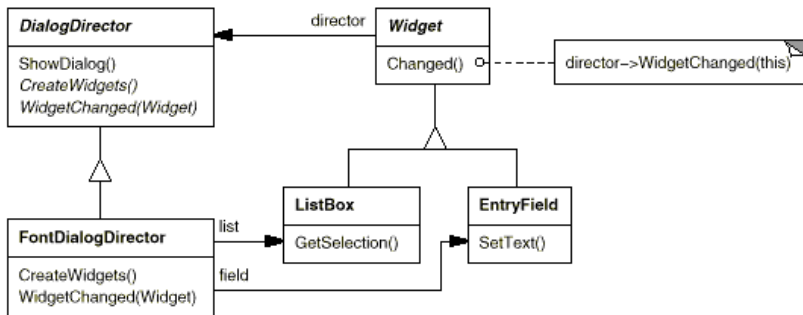
Behavioural GoF: Iterator Pattern



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

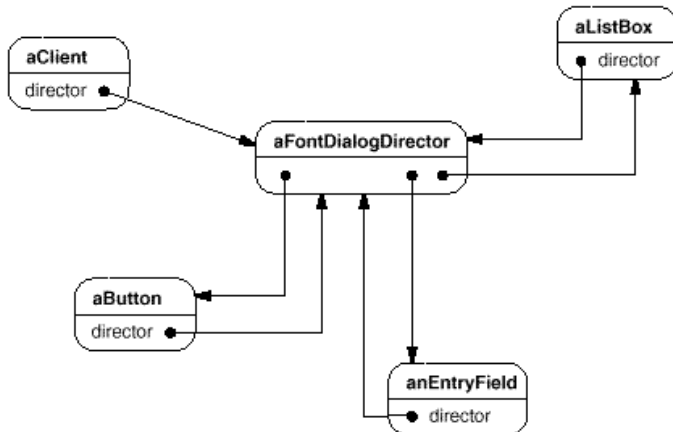
Behavioural GoF: Mediator Example Classes

To define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.



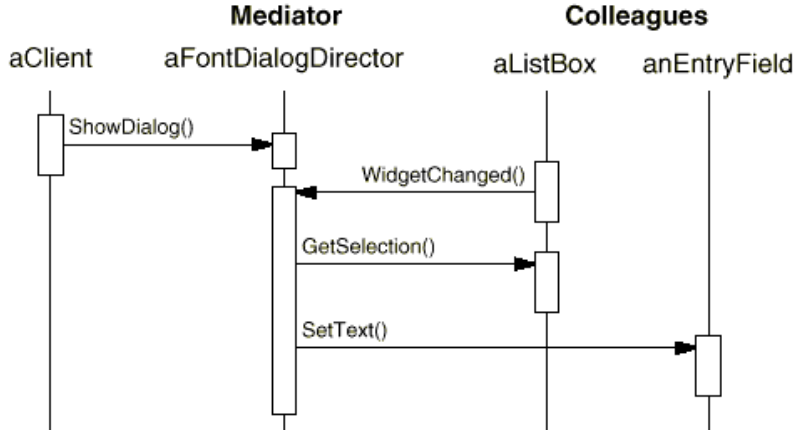
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Behavioural GoF: Mediator Example Objects



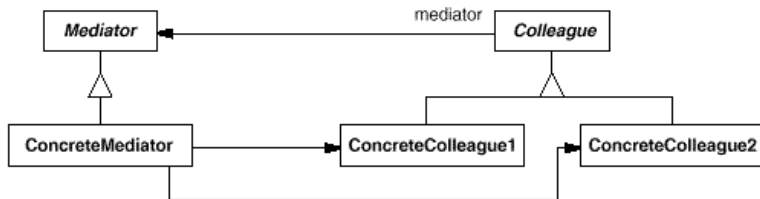
(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

Behavioural GoF: Mediator Example Interaction



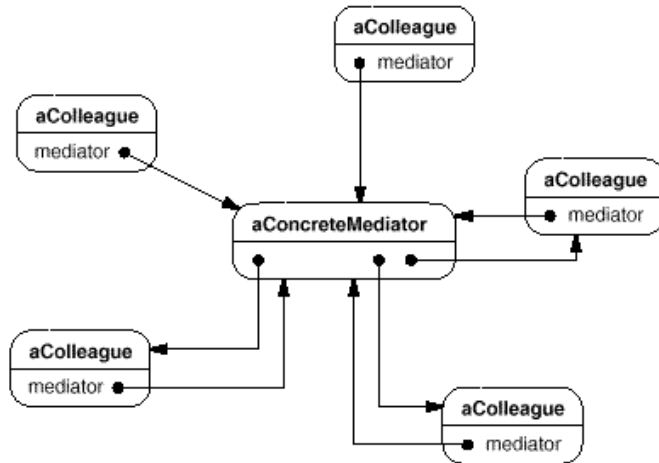
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Behavioural GoF: Mediator Pattern Classes



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

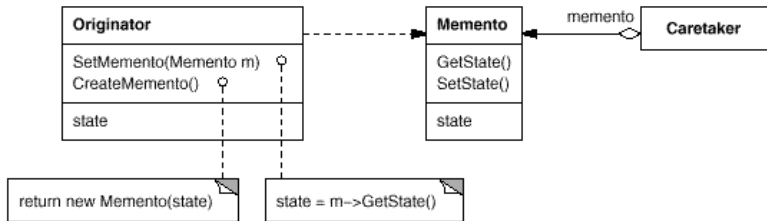
Behavioural GoF: Mediator Pattern Objects



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

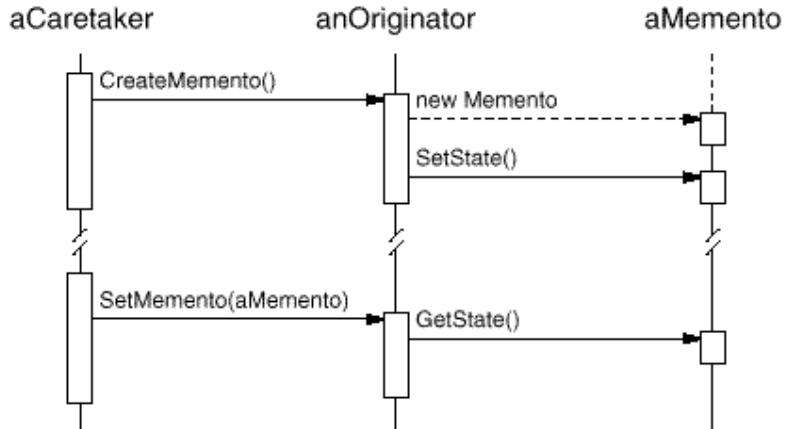
Behavioural GoF: Memento Pattern Classes

To capture and externalize an object's internal state, without violating encapsulation, so that the object can be restored to this state later.



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

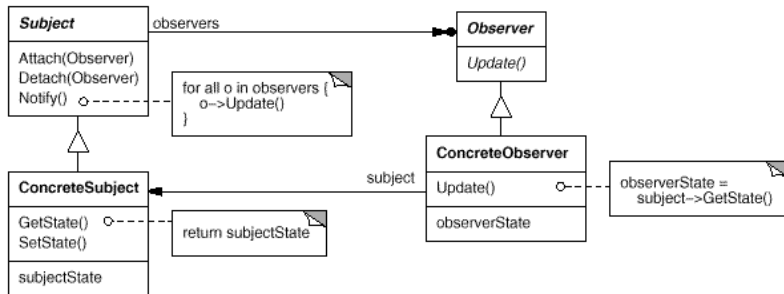
Behavioural GoF: Memento Pattern Interactions



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

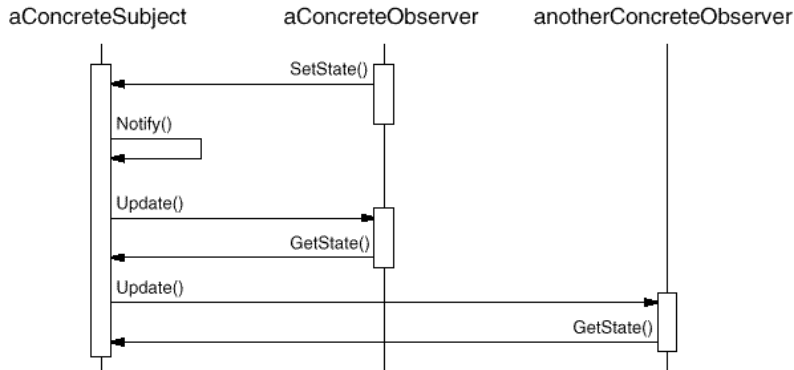
Behavioural GoF: Observer Pattern Classes

To define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



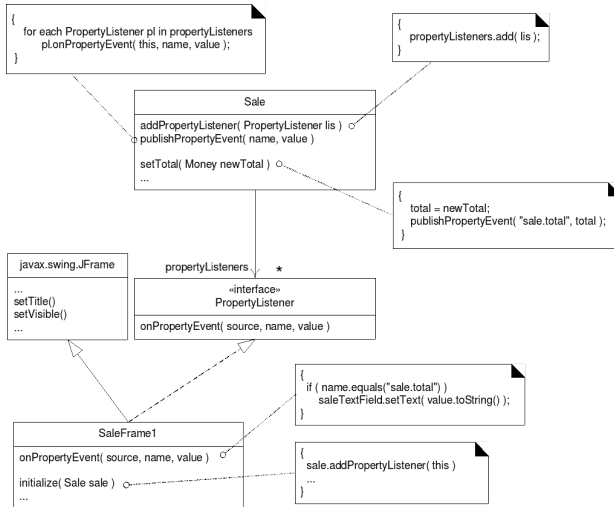
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Behavioural GoF: Observer Pattern Interactions



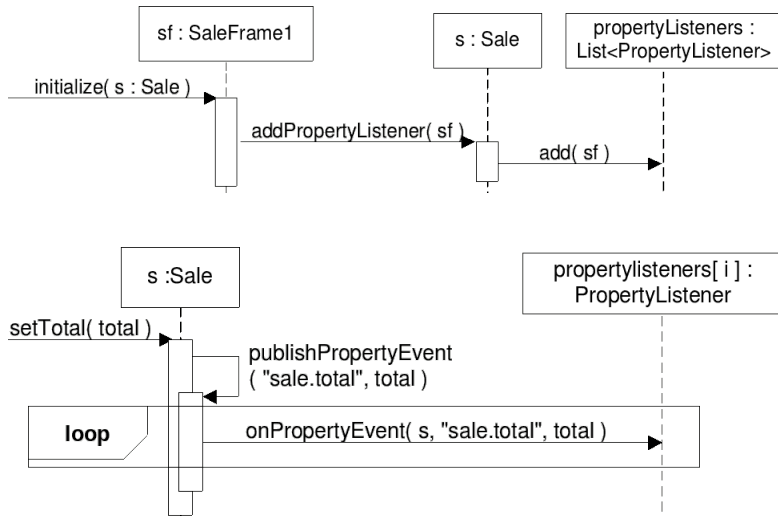
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

An Example: Observer in NextGen POS



(adopted from "Applying UML and Patterns" by Craig Larman)

An Example: Observer Sequence in NextGen POS (1)

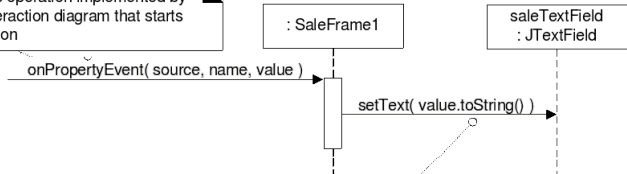


(adopted from "Applying UML and Patterns" by Craig Larman)

An Example: Observer Sequence in NextGen POS

23)

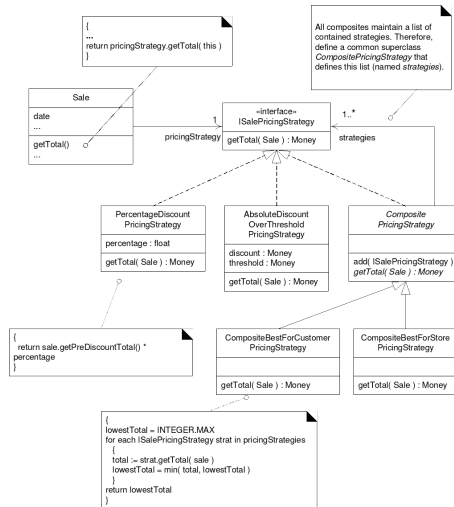
Since this is a polymorphic operation implemented by this class, show a new interaction diagram that starts with this polymorphic version



UML notation: Note this little expression within the parameter. This is legal and concise.

(adopted from “Applying UML and Patterns” by Craig Larman)

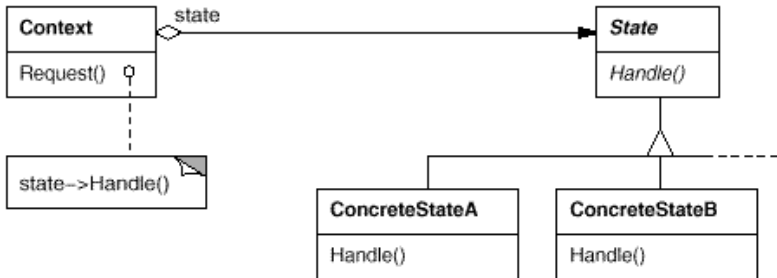
An Example: Composite Strategy in NextGen POS (1)



(adopted from "Applying UML and Patterns" by Craig Larman)

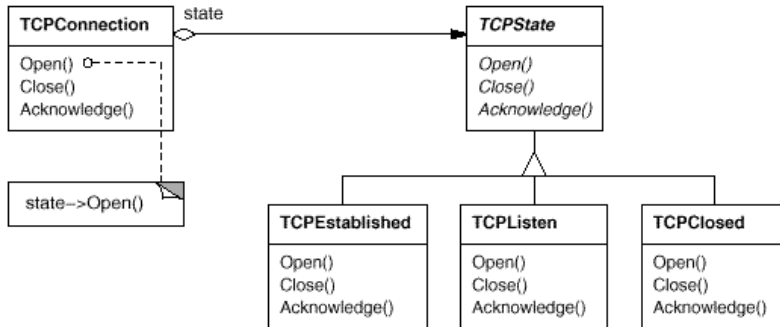
Behavioural GoF: State Pattern

To allow an object to alter its behaviour when its internal state changes. The object will appear to change its class on a state transition, however, it just switches an association to another state object providing particular implementation of operations in the target state.



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

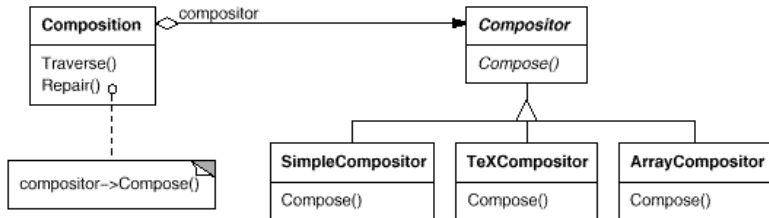
Behavioural GoF: State Example



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

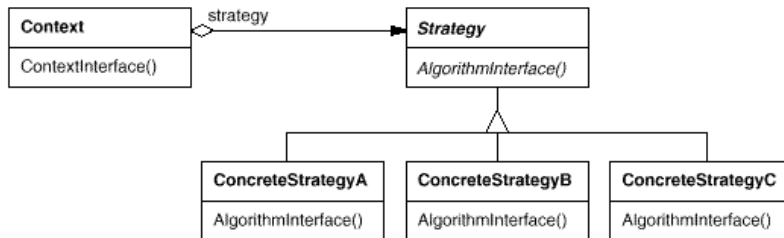
Behavioural GoF: Strategy Example

To define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



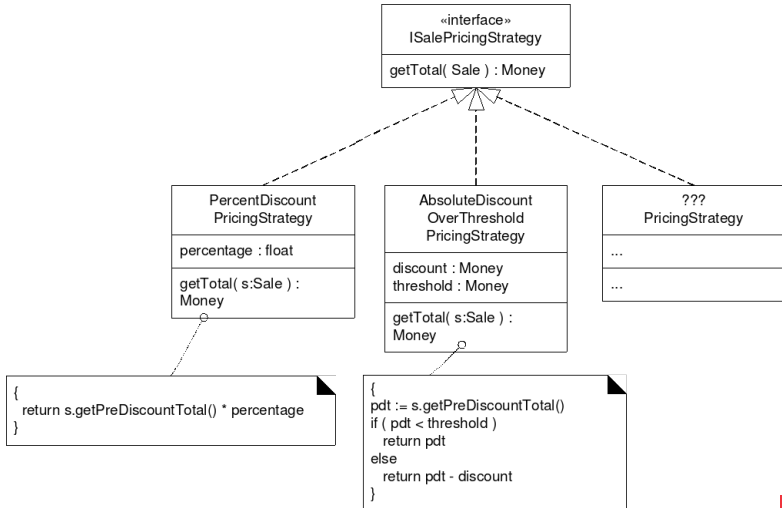
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Behavioural GoF: Strategy Pattern



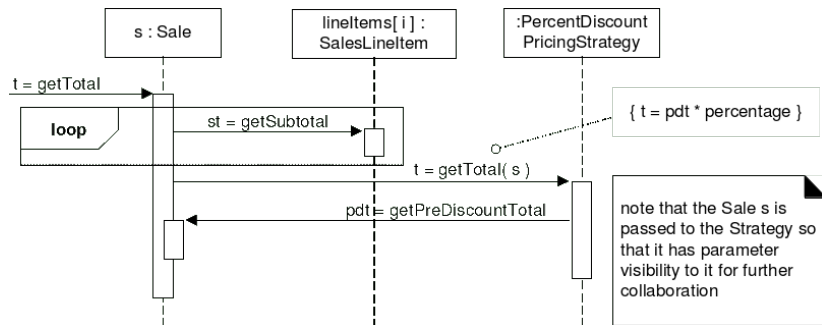
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

An Example: Strategy in NextGen POS



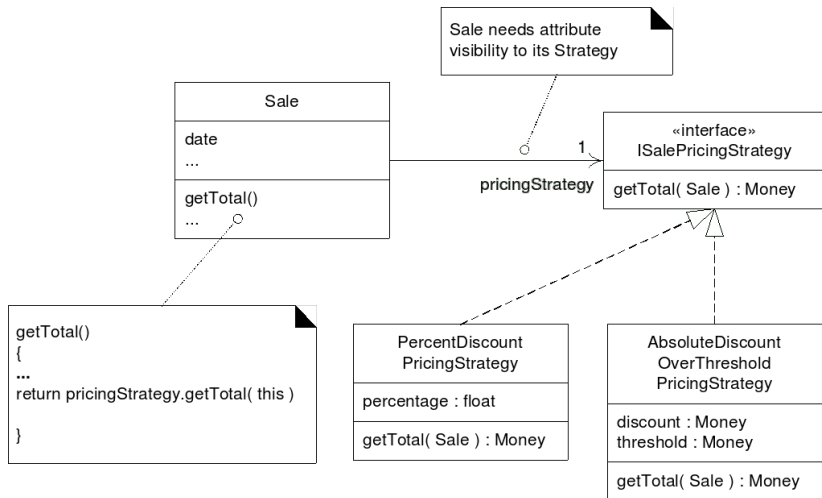
(adopted from "Applying UML and Patterns" by Craig Larman)

An Example: Strategy Interactions in NextGen POS



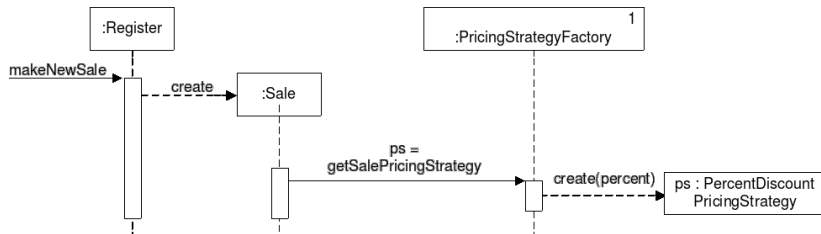
(adopted from "Applying UML and Patterns" by Craig Larman)

An Example: Strategy Usage in NextGen POS



(adopted from "Applying UML and Patterns" by Craig Larman)

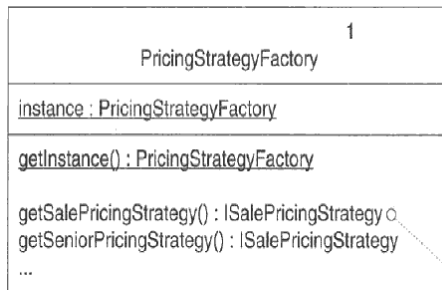
An Example: Strategy Creation in NextGen POS (1)



(adopted from "Applying UML and Patterns" by Craig Larman)

- Why do not create the strategy objects in ServicesFactory?
- Why to create it in `makeNewSale()` operation?
- How to read a parameter value of the current Strategy?
(e.g., the percents of discount)

An Example: Strategy Creation in NextGen POS (2)

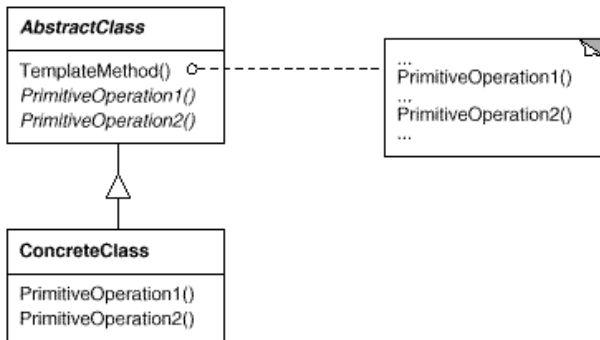


```
{  
    String className = System.getProperty( "salepricingstrategy.class.name" );  
    strategy = (ISalePricingStrategy) Class.forName( className ).newInstance();  
    return strategy;  
}
```

(adopted from "Applying UML and Patterns" by Craig Larman)

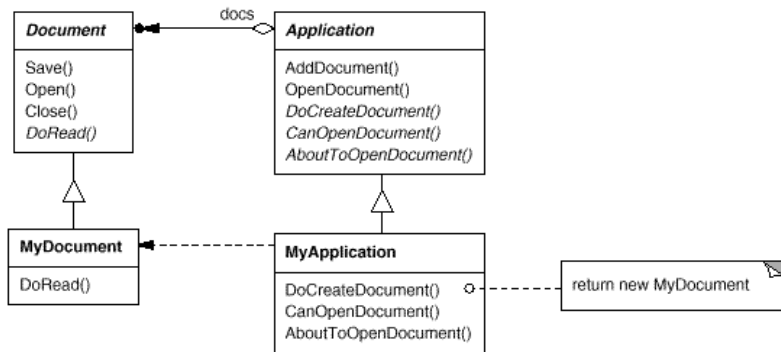
Behavioural GoF: Template Pattern

To define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

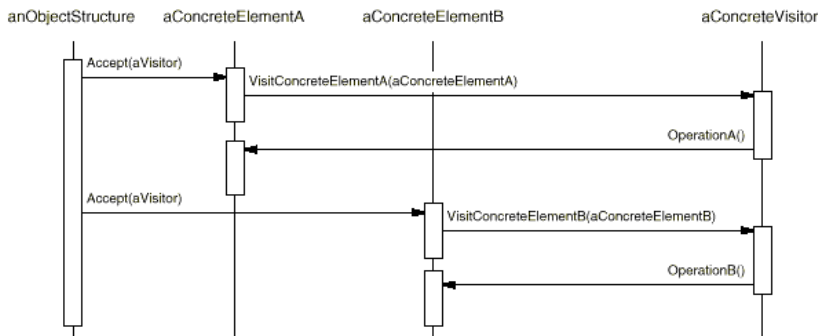
Behavioural GoF: Template Example



(adopted from “Design Patterns: Elements of Reusable OO Software” by Gamma, Helm, Johnson, and Vlissides)

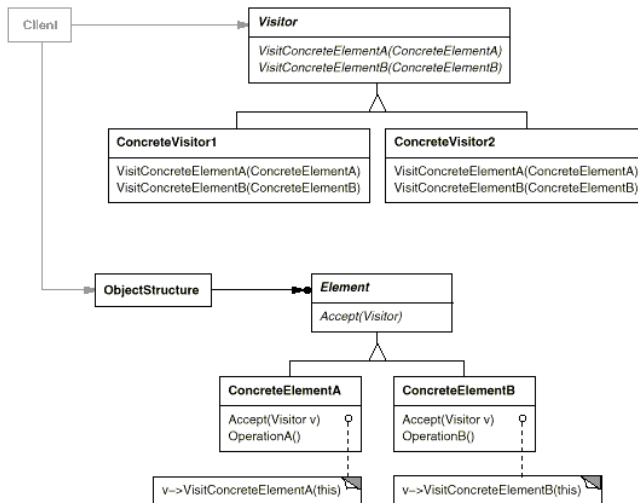
Behavioural GoF: Visitor Pattern Interactions

To represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.



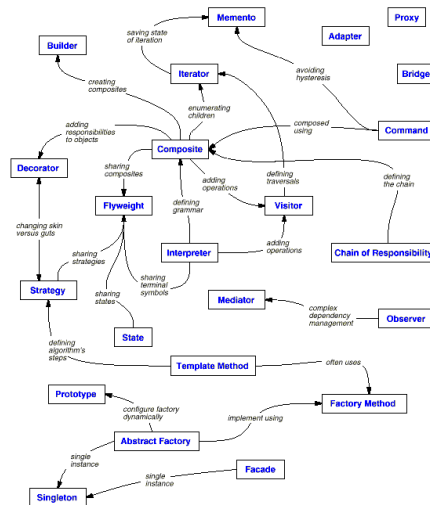
(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

Behavioural GoF: Visitor Pattern Classes



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

GoF Design Pattern Relationships



(adopted from "Design Patterns: Elements of Reusable OO Software" by Gamma, Helm, Johnson, and Vlissides)

General Responsibility Assignment Software Patterns

GRASP in “Applying UML and Patterns” by Craig Larman

- Guidelines for assigning responsibility to classes and objects.
(based on Responsibility-Driven Design/RDD and Object Orientation/OO)
- Explain the reasons for and benefits of most GoF design patterns.
(GRASP is more about principles, more fundamental than the GoF patterns)
- Should be applied collectively instead of one at a time or a case.

-
- (Information) Expert
 - Creator
 - Controller
 - Low Coupling
 - High Cohesion

- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

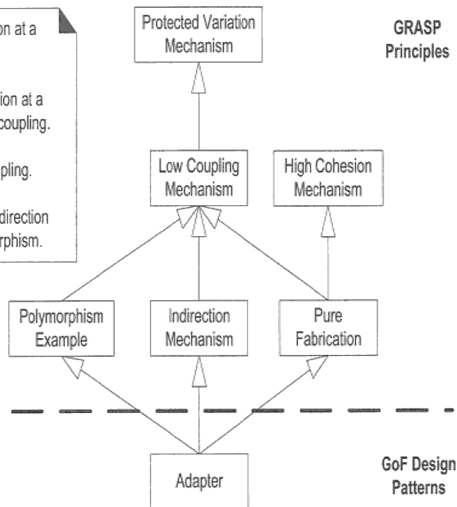
GRASP and GoF Adapter Design Pattern

Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.



(adopted from "Applying UML and Patterns" by Craig Larman)

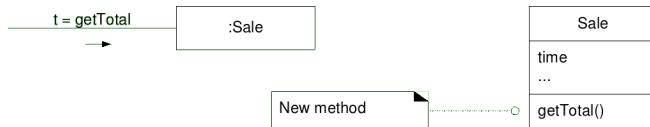
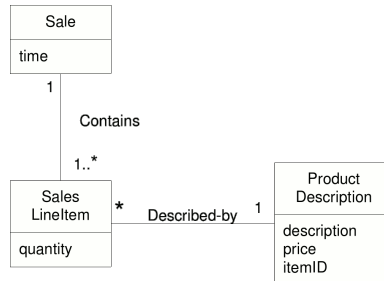
GRASP: Information Expert

Problem What is a basic principle by which to assign responsibilities to an object?

Solution Assign a responsibility to the class that has the information needed to respond to it.

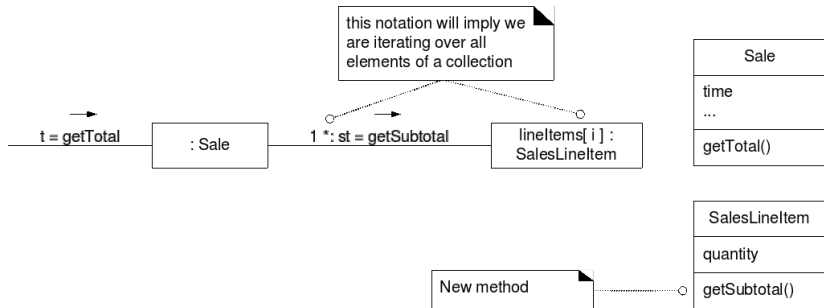
- It may not be possible for some (technical) reasons.
(e.g., when the information is loaded from a database)
- Use this principle in order to get low coupling and high cohesion.

An Example: Who Provide “getTotal()” Operation?



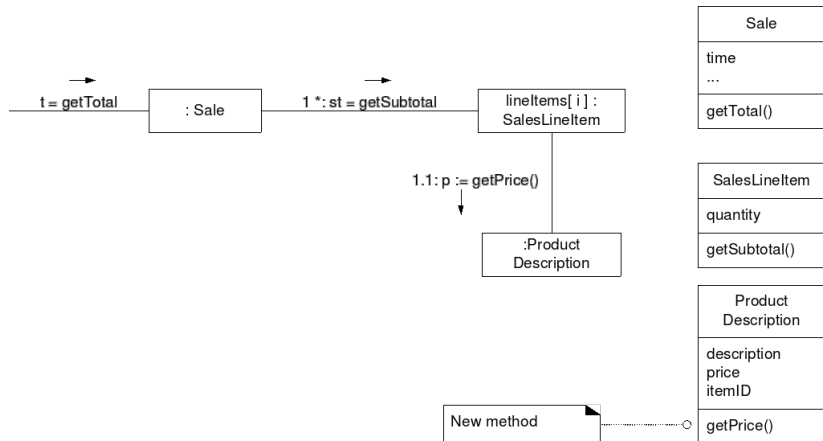
(adopted from “Applying UML and Patterns” by Craig Larman)

The Example: with “getSubTotal()” Operation



(adopted from "Applying UML and Patterns" by Craig Larman)

The Example: with “getPrice()” Operation



(adopted from “Applying UML and Patterns” by Craig Larman)

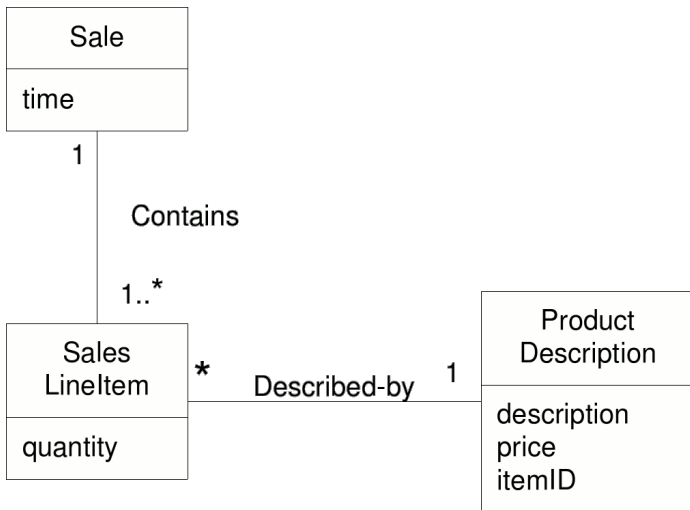
GRASP: Creator

Problem Who creates an object *A*?

Solution Assign class *B* the responsibility to create an instance of class *A* if

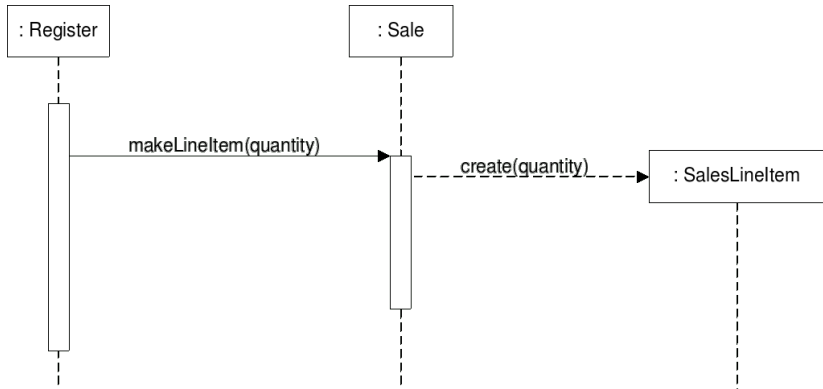
- *B* contains or completely aggregates *A* (composition/aggregation),
 - or *B* records *A* (e.g., as a register),
 - or *B* closely uses *A*,
 - or *B* has the initializing data for *A*.
-
- For a complex creating process, use the Factory pattern or others.
 - This principle keeps low coupling, as *B* usually already depends on *A*.

An Example: Who Does Create “SalesLineItem”?



(adopted from “Applying UML and Patterns” by Craig Larman)

The Example: “Sale” Will Create “SalesLineItem”



(adopted from “Applying UML and Patterns” by Craig Larman)

GRASP: Controller

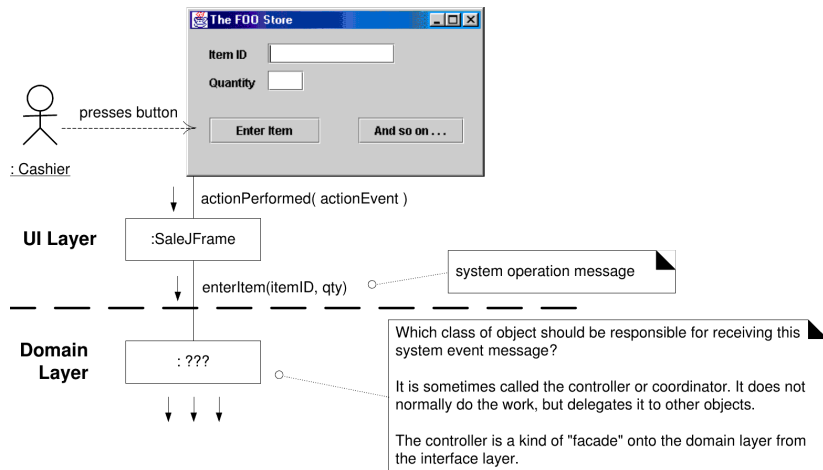
Problem What first object beyond the UI layer receives and coordinates a System Operation / a message from the UI layer?

Solution Assign the responsibility to an object which represents one of

- the overall “system” as a root object,
(aka “facade controller”; usually a device running the system)
- or a use case scenario within which the system operation occurs.
(aka “use-case/session controller”; per each use case or several of them)

- A controller should not implement, just delegate, control, and coordinate.
- Use “facade controller” if there is a lot of various system operations.
- Use “use-case/session controller” otherwise (low coupling/high cohesion).

An Example: “enterItem” UI Layer and Controller

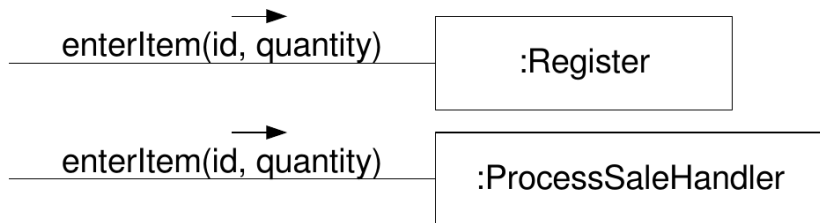


(adopted from “Applying UML and Patterns” by Craig Larman)

The Example: “enterItem” Controllers

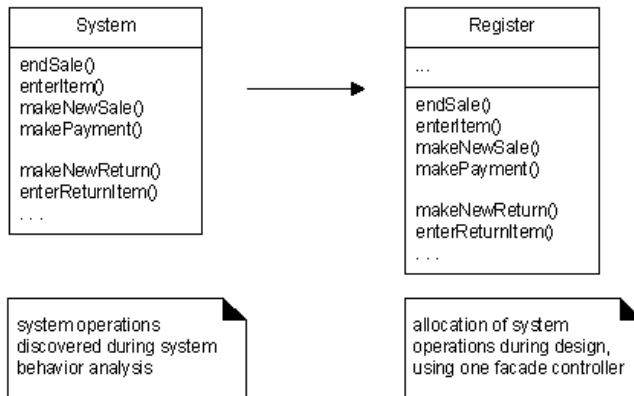
According to the Controller principle, the class acting as a controller for “enterItem” can be

- the overall “system” as “Register” facade controller,
- or the scenario as “ProcessSaleHandler” use-case controller.



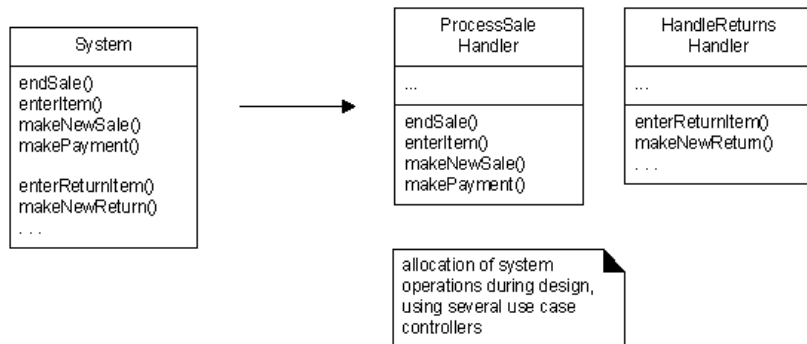
(adopted from “Applying UML and Patterns” by Craig Larman)

The Example: Facade Controller



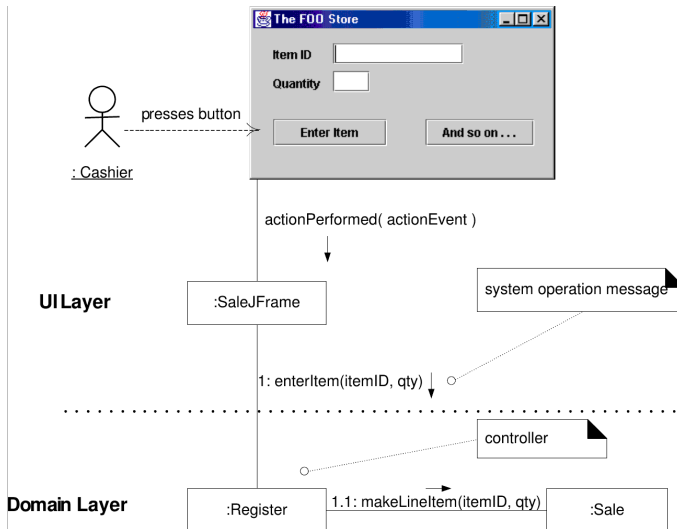
(adopted from "Applying UML and Patterns" by Craig Larman)

The Example: Use-Case Controller



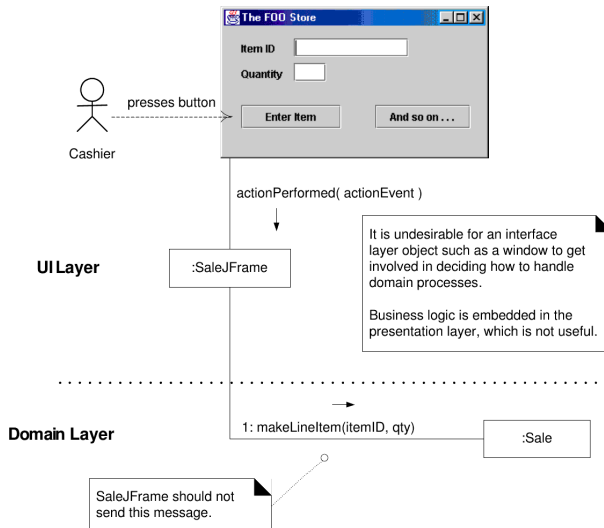
(adopted from "Applying UML and Patterns" by Craig Larman)

The Example: Delegation in the Controller



(adopted from "Applying UML and Patterns" by Craig Larman)

The Example: Domain Objects are not Controllers



(adopted from "Applying UML and Patterns" by Craig Larman)

GRASP: Low Coupling

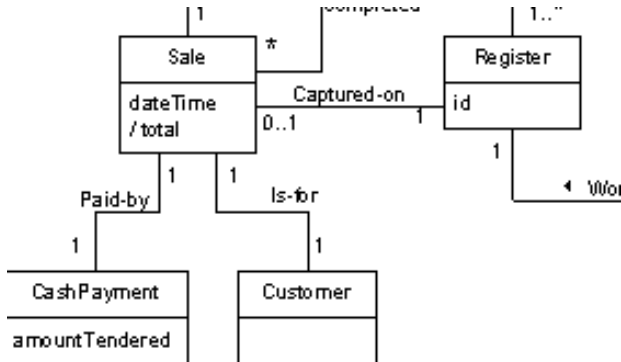
Problem How to keep objects focused, understandable, and manageable, and, as a side effect, support the low coupling?

Solution Assign responsibilities so that cohesion remains high. Use this criteria to evaluate alternatives.

-
- high cohesion
(the degree to which the elements of a component belong together)
 - low coupling
(the degree to which the different components depend on each other)

An Example: Payment Creation in Sale

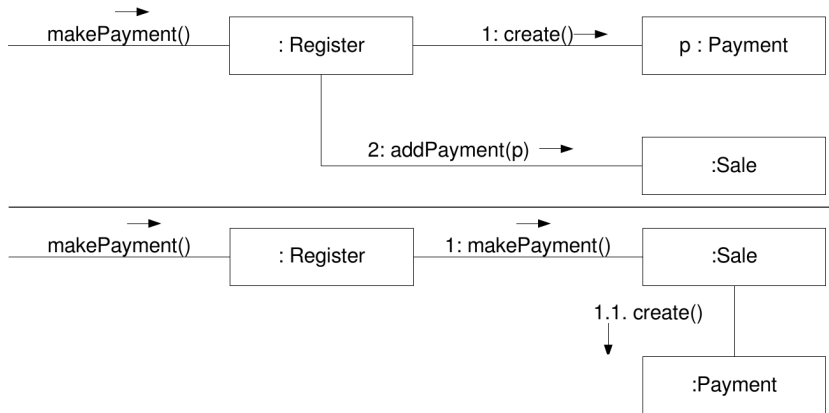
The domain model of Payment, Sale, and Register.



(adopted from "Applying UML and Patterns" by Craig Larman)

The Example: Two Ways to Payment Creation

Which way is better and why? We would prefer a low coupling.



(adopted from "Applying UML and Patterns" by Craig Larman)

GRASP: High Cohesion

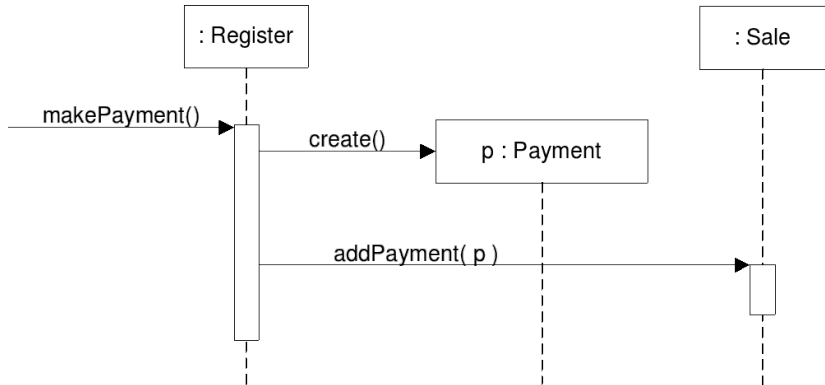
Problem How to keep objects focused, understandable, and manageable, and, as a side effect, support low coupling?

Solution Assign responsibilities so that cohesion remains high. Use this criteria to evaluate alternatives.

- High cohesion means better architecture, usually also low coupling.
- Sometimes, it makes sense to break the high cohesion principle.
(e.g., put various operation into one class for easier maintenance, for the ability to query a database from one place in the code, etc.)

The Example: Two Ways to Payment Creation (1)

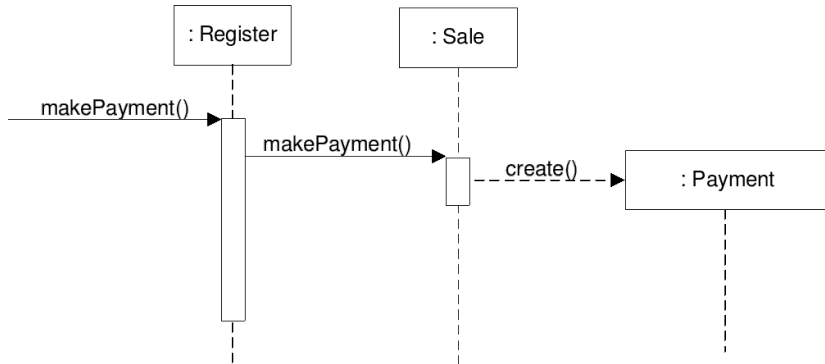
Which way is better and why? We would prefer a high cohesion.



(adopted from "Applying UML and Patterns" by Craig Larman)

The Example: Two Ways to Payment Creation (2)

Which way is better and why? We would prefer a high cohesion.



(adopted from "Applying UML and Patterns" by Craig Larman)

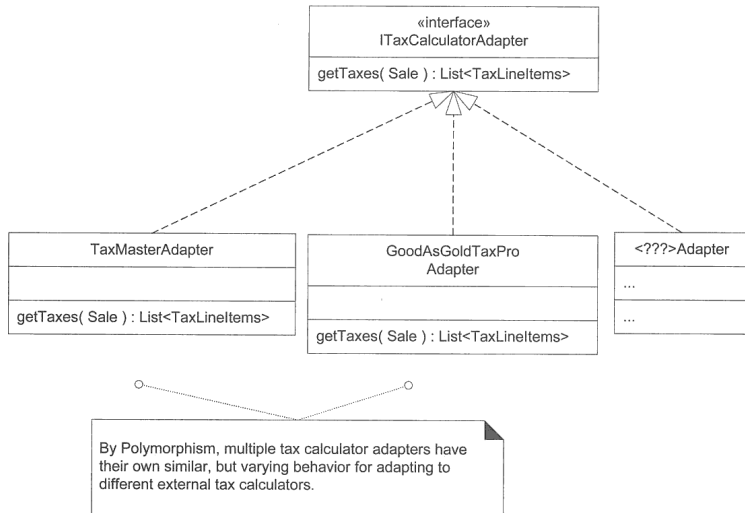
GRASP: Polymorphism

Problem Who is responsible when behaviour varies by type? How create pluggable software components that fit into a particular context?

Solution When related alternatives or behaviours vary by type (class), assign responsibility for the behaviour to the types for which the behaviour varies by using polymorphic operations. In this context, polymorphism means giving the same name to similar or related services.

- Polymorphism is easier and more reliable than using explicit selection logic, and supports the ability to add additional behaviours later on.
- However, it also increases the number classes in a design and make the code less easy to follow.

The Example: Polymorphism in Tax Calculator



(adopted from “Applying UML and Patterns” by Craig Larman)

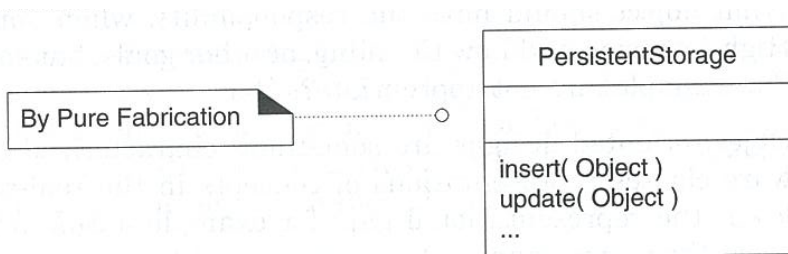
GRASP: Pure Fabrication

Problem Sometimes assigning responsibilities required by experts/users to domain objects would violate High Cohesion, Low Coupling, or Reuse. What objects should have such responsibilities?

Solution Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a domain concept.

- Assign the responsibilities to something made up just to support High Cohesion, Low Coupling, or Reuse.
- Useful for responsibilities outside of scope of the domain objects.
(e.g., providing persistence, which should be extracted into a stand-alone layer)

The Example: Pure Fabrication for PersistentStorage



(adopted from "Applying UML and Patterns" by Craig Larman)

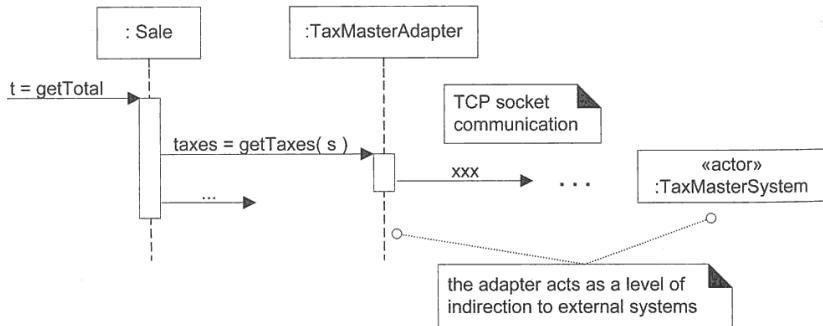
GRASP: Indirection

Problem How to assign responsibilities to avoid direct coupling? How to decouple objects so that Low Coupling is supported and reuse potential remains higher?

Solution Assign the responsibility to an intermediate object to mediate between other components or service, so that they are not directly coupled.

- For objects with dependency/coupling on technical objects/solutions. (e.g., databases for load/save of persistent objects, third party systems, etc.)
- By introducing a mediator into coupling, responsibilities are moved away from the original object into a specialised object with stable interfaces.

The Example: Indirection in Tax Calculator



(adopted from "Applying UML and Patterns" by Craig Larman)

GRASP: Protected Variations

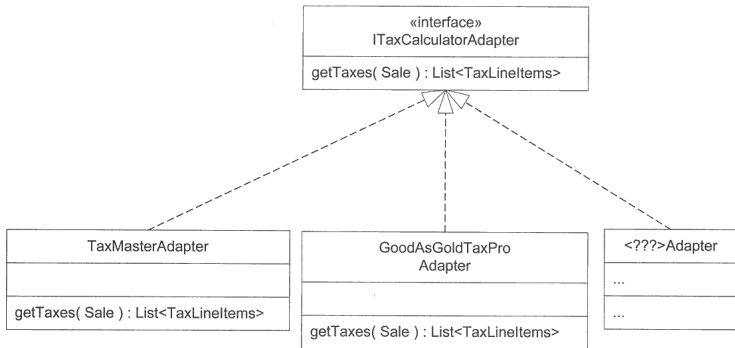
Problem How to assign responsibilities to objects, subsystems, and systems so that the variations or instability in these elements do not have an undesirable impact on other elements?

Solution Identify points of predicted variations or instability; assign responsibilities to create a stable “interface” around them.

- Point of changes = variation points and evolution points.
(will be modified in the existing system or in its future versions, respectively)
- Cooperating objects need to be protected from the changes by stable interfaces encapsulating the points of changes → protected variations.

The Example: Protected by Polymorphism/Adapter

(future changes in the tax calculator would not cause problems)



By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

Law of Demeter (LoD) / Principle of Least Knowledge

(phrasing from “A Simple way to apply Law of Demeter to your Ruby objects” by Emerson Macedo)

- ❶ Each unit should have a knowledge about other units limited only to units “closely” related to the current unit.
 - ❷ Each unit should only talk to its friends and should not talk to strangers.
 - ❸ Each unit should only talk to its immediate friends.
-
- The style rule was first proposed at Northeastern University in the fall of 1987 by Ian Holland while working on The Demeter Project.
 - A specific case of loose coupling.
(it is related to the Protected Variations principle)
 - These rules should be respected to keep a good architecture.
(however, in some cases, their breaking need not to be a problem; e.g., in method chaining/named parameter idiom)

Thank you for your attention!

Marek Rychlý

`<rychly@fit.vutbr.cz>`