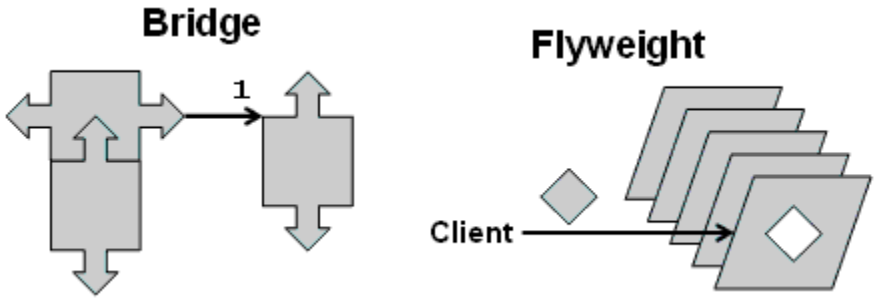
Design Patterns

Outline



- Design Patterns:
  - definition
  - description template
  - properties
- „Gang of Four" catalogue of Design Patterns:
  - structural
  - behavioral
  - creational
- Side topics:
  - Code-smells, antipatterns, design principles…

Design Patterns - definition

- "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"

Chrisopher Alexander, 1977

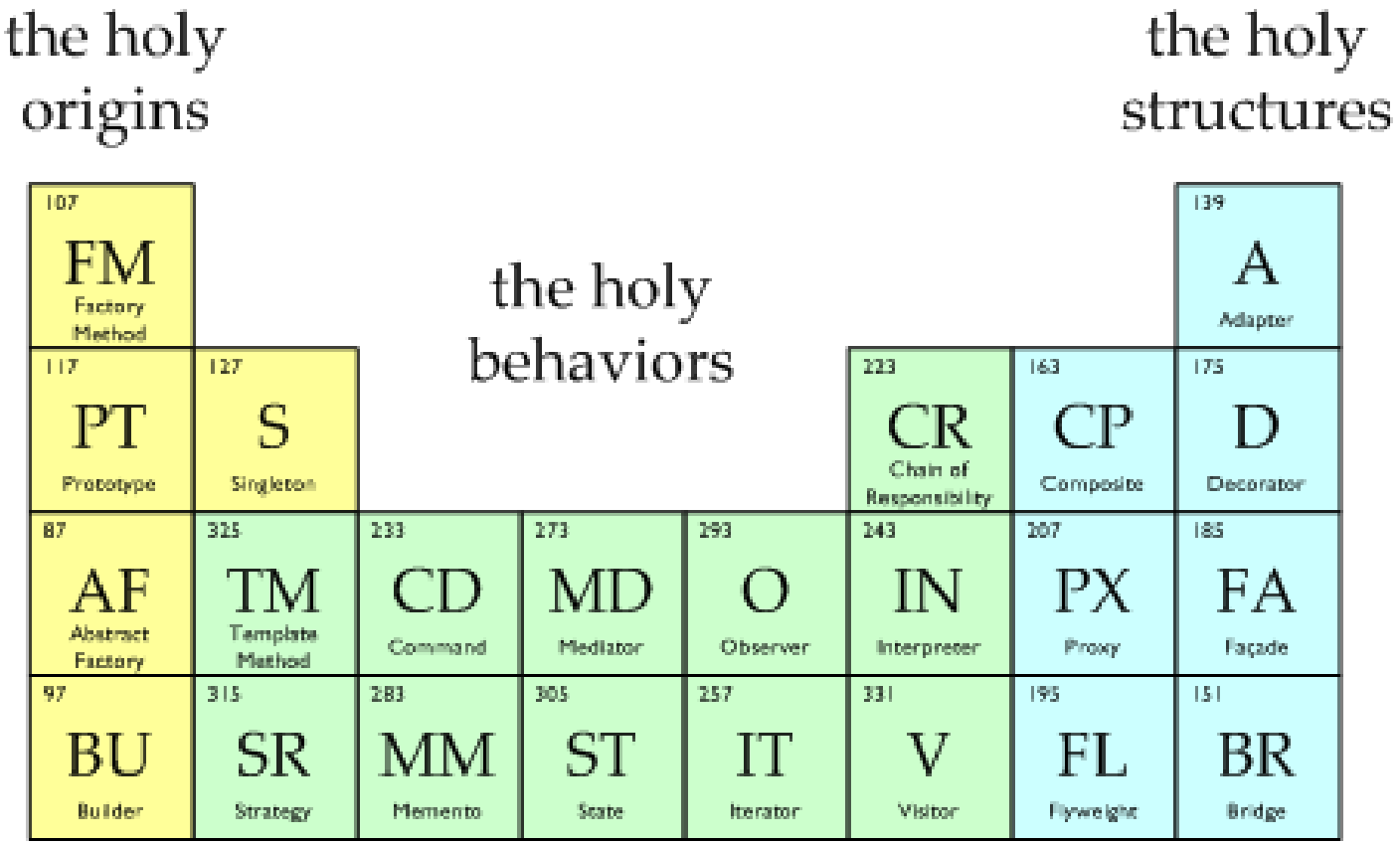
Design Patterns – the beginning

- Model-View-Controller (MVC) – a triad of classes is used to build user interfaces in Smalltalk-80 (T. Reenskauga early 80's)

- **„Gang of Four"**: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley; 1995

GoF patterns

The Sacred Elements of the Faith



Pattern elements (GoF)

- **Name** is introduced to uniquely identify and unify language. It is a handle we can use to describe a design *problem*, its *solutions*, and *consequences* in a word or two.
- **Problem** describes when to apply the pattern. It explains the problem and its context. It might concern design problems, symptoms (e.g. code-smells), list of conditions, etc.
- **Solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. It doesn't describe a particular concrete design or implementation – it is a kind of a template.
- **Consequences** are the results and trade-offs of applying the pattern (it terms of e.g. *reuse*, *flexibility*, *extensibility*, *portability*)

Why use design patterns?

- They result from many practical experiences.
- Design patterns set the terminology:
  - Facilitates communication with other designers and programmers,
  - It imposes a specific design terminology.
- They simplify the restructuring of existing systems.
- They enable reuse of proven solutions.

- But …
- Design pattern is a semi-finished product.
  - They must be processed and implanted in the whole project

Patterns classification (GoF)

- Structural – Compose classes or objects
  - Adapter, Decorator, Facade, Composite, Bridge, Proxy, Flyweight.
- Behavioral - Allow flexible change of behavior
  - Interpreter, Iterator, Chain of responsibility, Mediator, Template method, Observer, Visitor, Memento, Command, State, Strategy.
- Creational – Facilitate object creation process
  - Builder, Abstract factory, Factory method, Prototype, Singleton.

Other patterns

- Patterns start but do not end with GoF
- All patterns are based on certain foundations of objectivity
  - Inheritance and polymorphism
  - Interfaces
  - Delegation
- There are also other types of patterns:
  - Concurrency, (e.g., Active Object, Thread Specific Storage, Thread Pool Pattern, Monitor Object, …)
  - Architectural (SOA, Client-Server, Three-tier, Pipeline, …),
  - Specific for a specific application field (Active Record, Domain Model, Metadata mapping, …)
  - …

Reuse in Frameworks

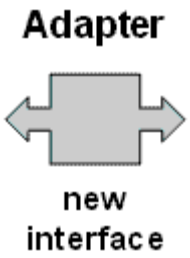
- Software Frameworks have design patterns implemented in a reusable project.
  - .NET Framework
  - Spring Framework
  - Ruby on Rails
  - Eclipse Framework, NetBeans Framework
  - Symfony Framework
  - Struts
  - Hibernate
  - …

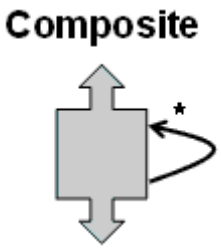
Structural patterns

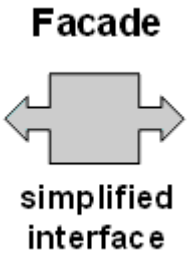
Gang of Four

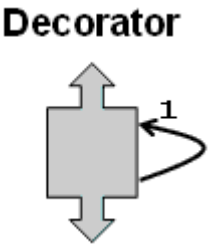
Roadmap

- Adapter
- Composite
- Facade
- Decorator







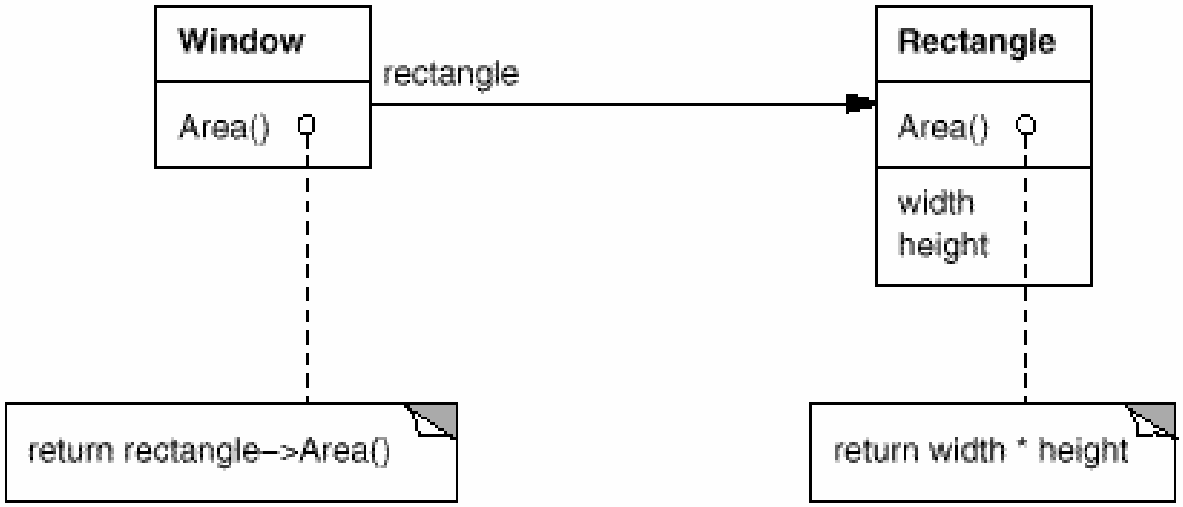


Basic concepts

- object …
- interface …
  - polymorphism …
- type …
- class …
  - class vs interface inheritance?

What is delegation?

- Simply: forwarding (delegating) a request (operation) by the object receiving the message for execution to another object (the so-called delegate)
- Increase reuse by using aggregation instead of inheritance (Principle!)

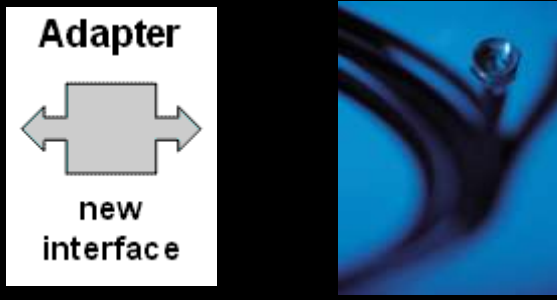


Structural patterns

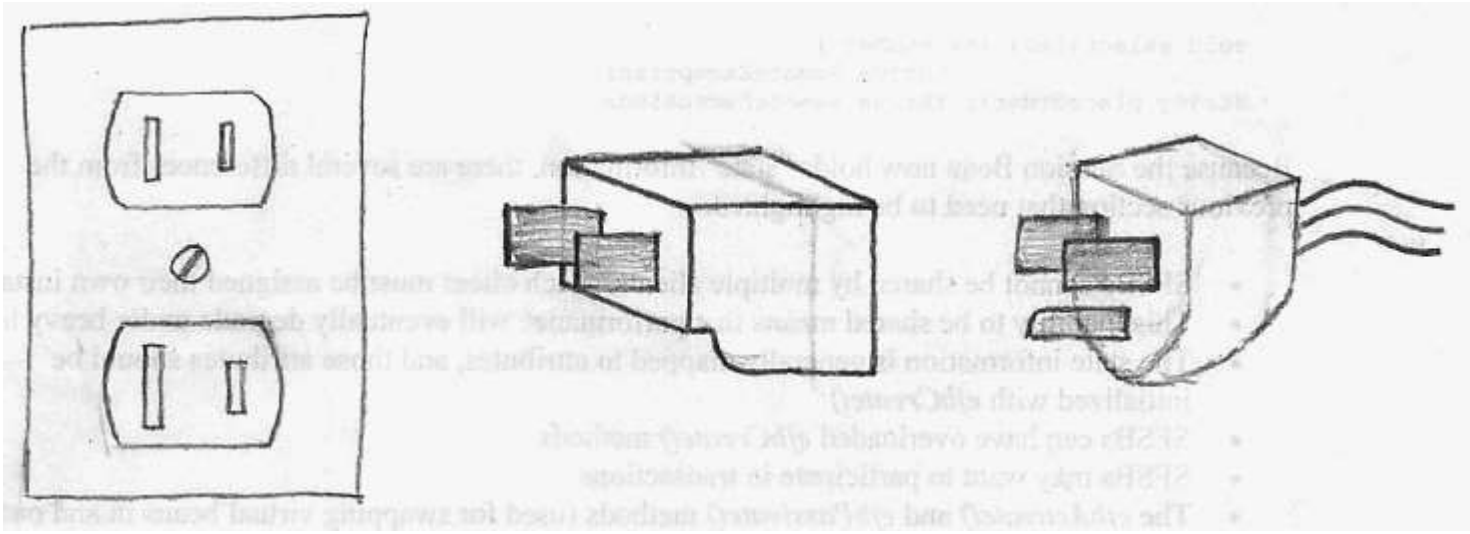
- **Structural patterns** concern common ways of organizing objects of different types so that they can cooperate with each other.

- Organization, management, composition, defining and redefining of structures.

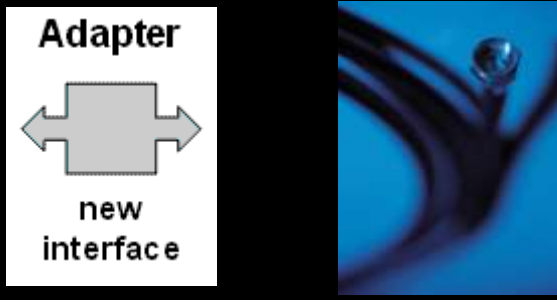
Adapter



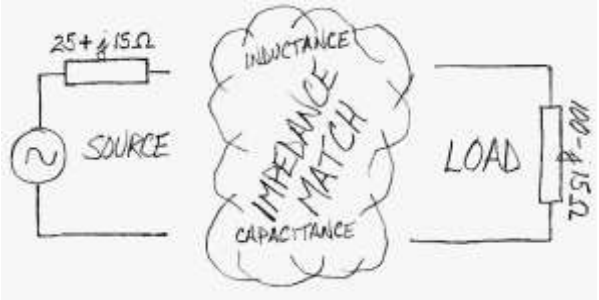
- Convert the interface of a class into another interface clients expect.



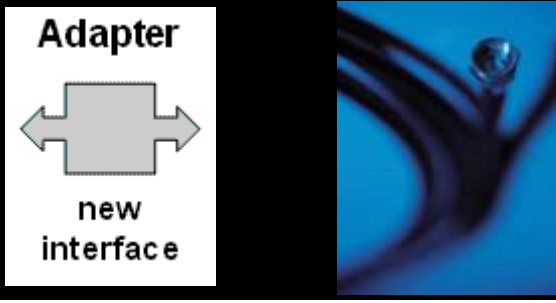
Adapter - Problem



- An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.
  - Incompatible interface
  - Impedance mismatch

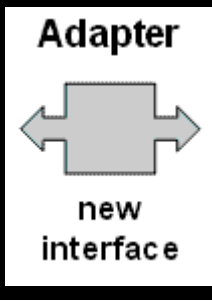


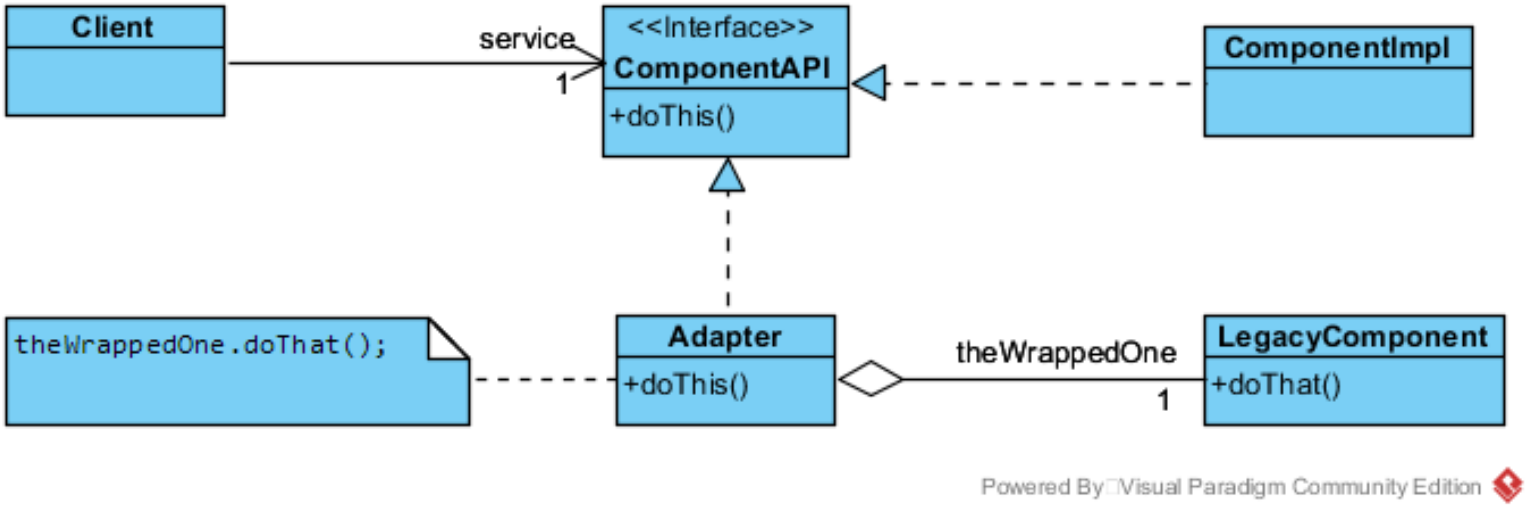
Adapter - Solution



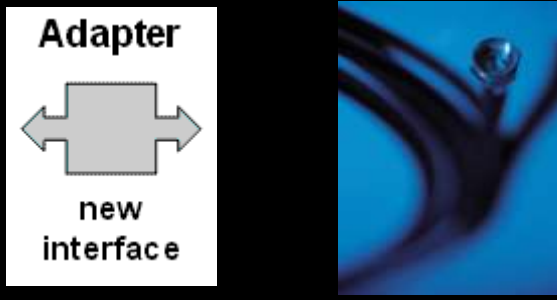
- We wrap the existing code with new interfaces.
- We adjust the impedance of old components to the new system
  - often an apparent solution! –
    i.e., like sewing on an old patch to a new trousers
- Structure: Wrapper / Delegation.

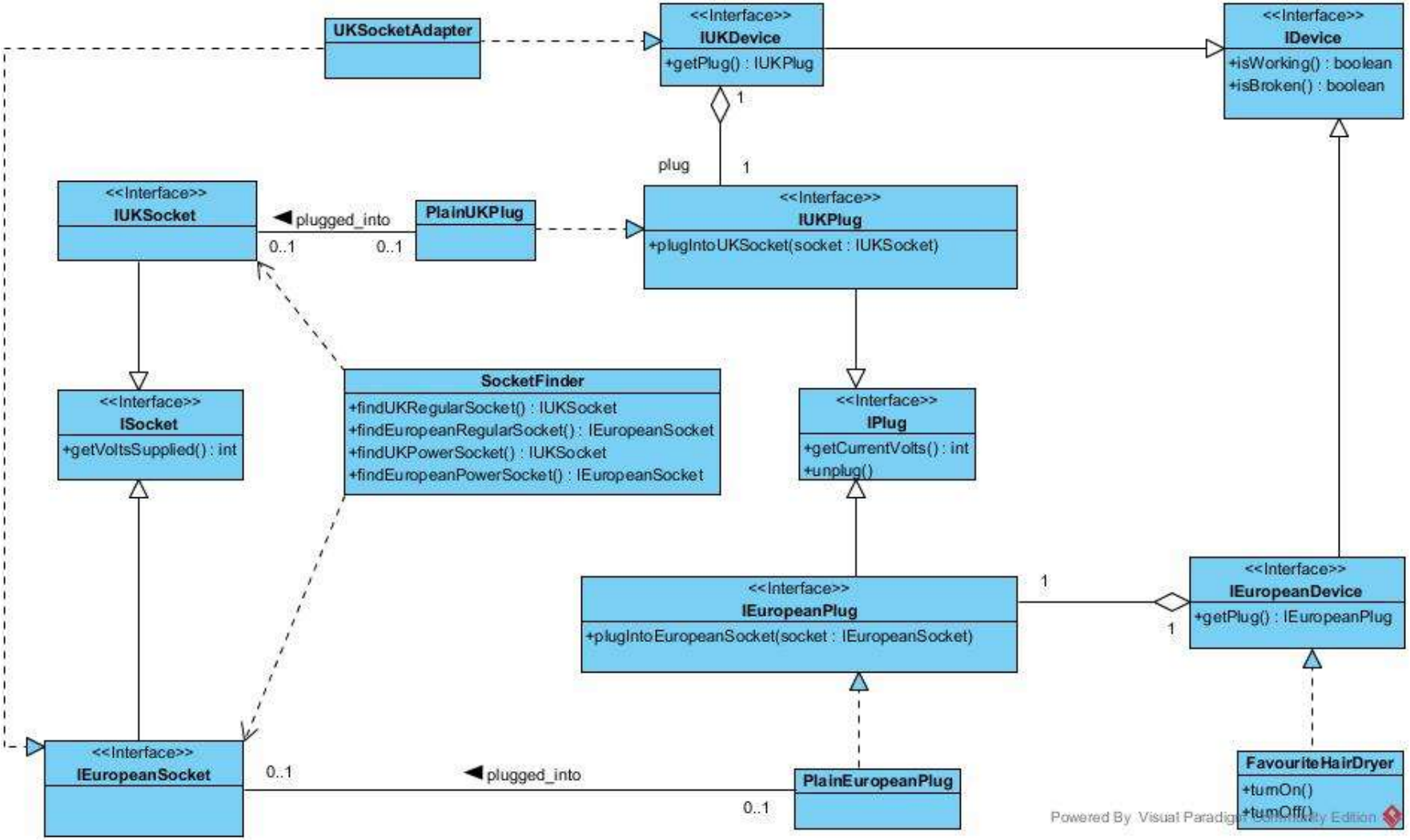
Adapter – Class diagram



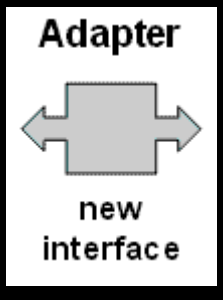


Adapter – Example





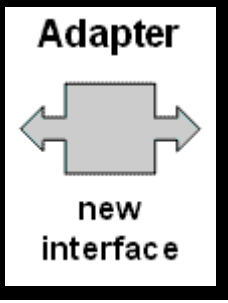
Adapter - Consequences

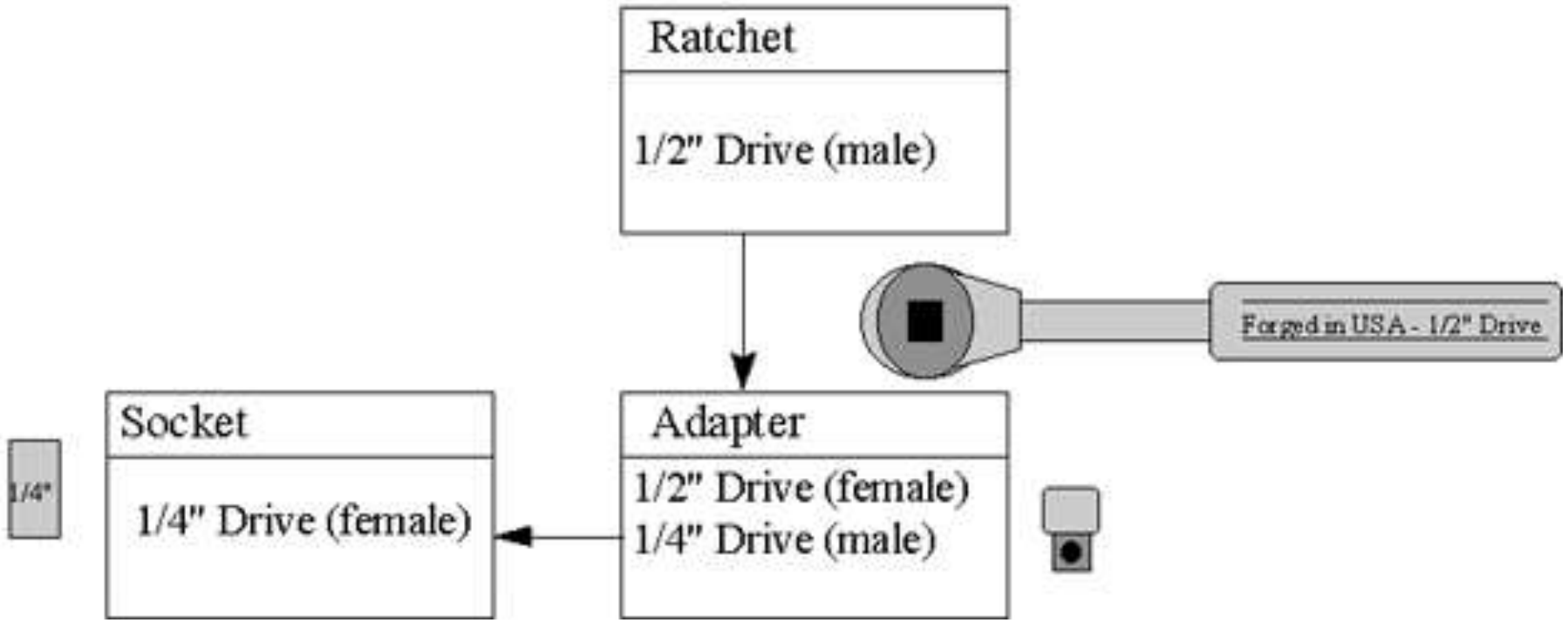


- The client and the adapted component (class, method, etc.) remain independent.
- You can use adapter classes to determine which object method is to be called by the client (for example, one adapter calls a method that draws a solid line and the other calls a method that draws a dashed line)

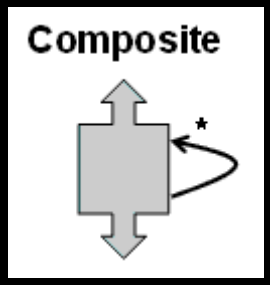
- The adapter adds an intermediate layer in the program:
  - negative impact on performance (for low-level components),
  - difficulty understanding the application (on implementation level).

Adapter – non-software example



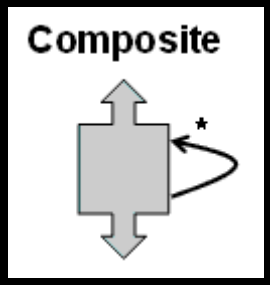


Composite



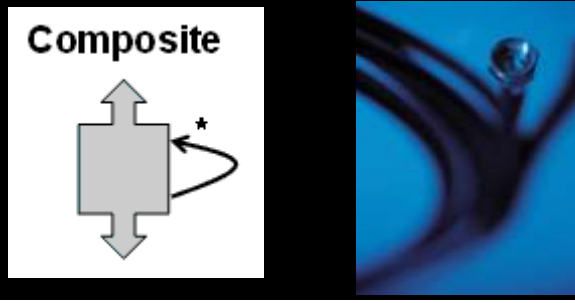
- Compose objects into tree structures to represent whole-part hierarchies.
  - Defining an interface that considers both individual objects and groups of objects.
- Composite lets clients treat individual objects and compositions of objects uniformly.

Composite - Problem



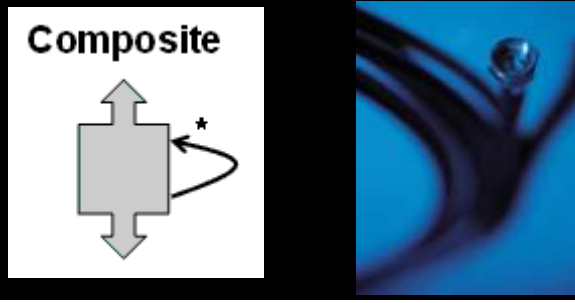
- Decomposition of a complex object into a hierarchy of part-whole objects.
- Having to query the "type" of each object before attempting to process it is not desirable.
- Many different objects are used in a similar way and have almost identical service code.

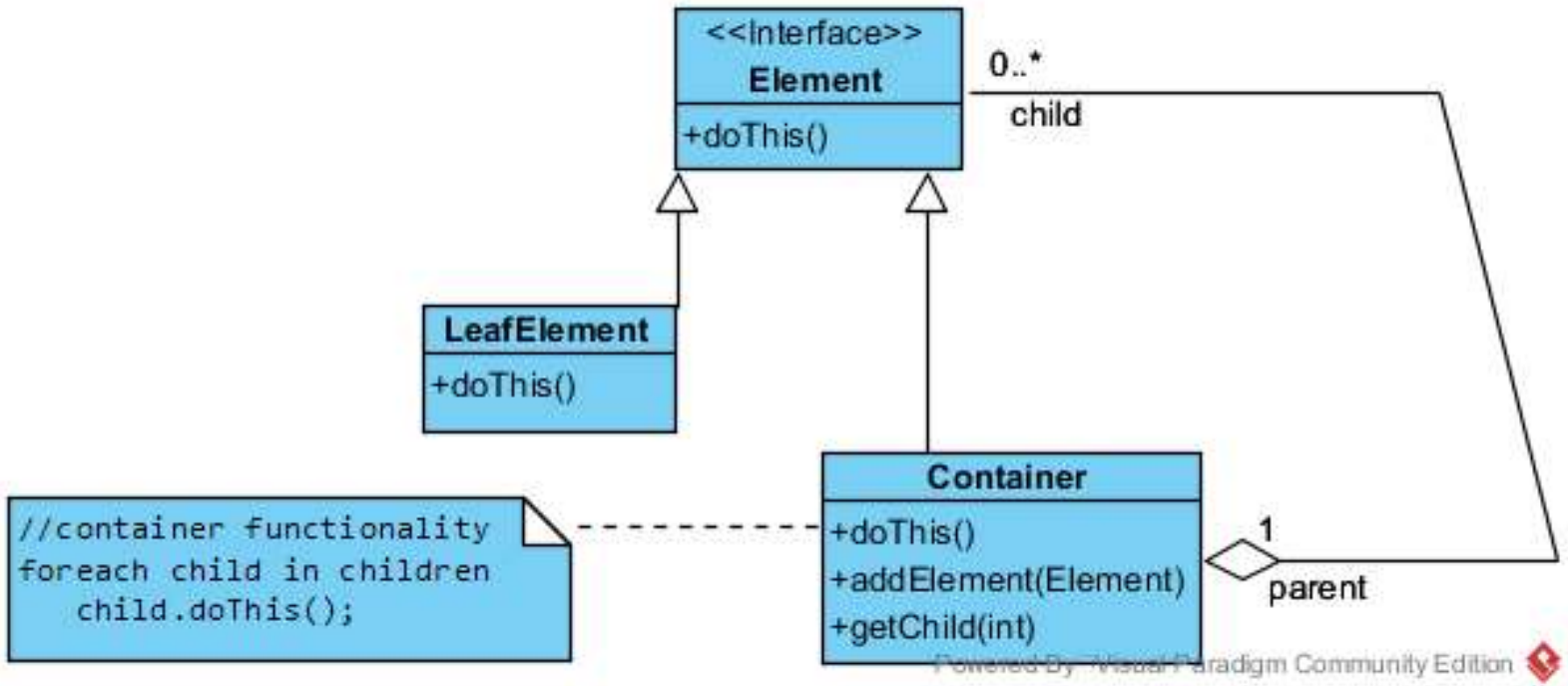
Composite - Solution



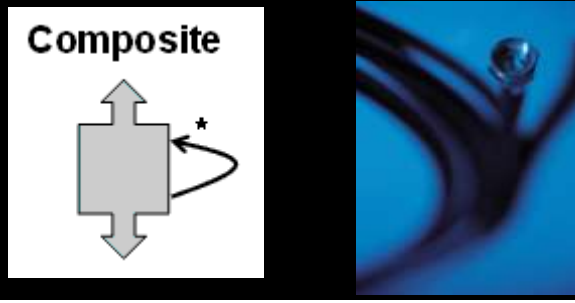
- Define a base class that makes composites and element objects interchangeable
- Recursive composition
- 1-to-many "has a" up the "is a" hierarchy

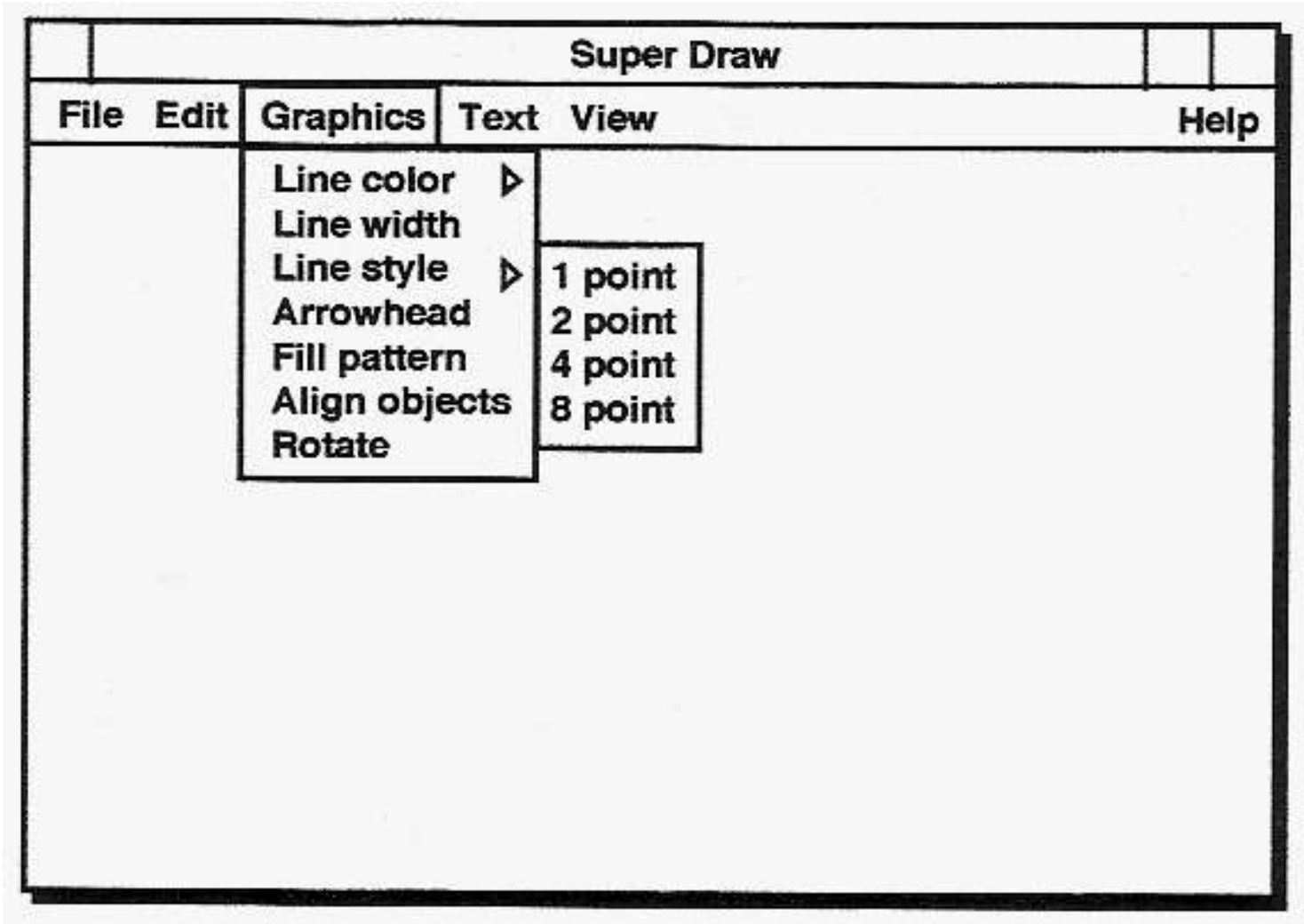
Composite – Class diagram



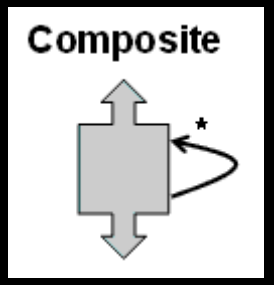


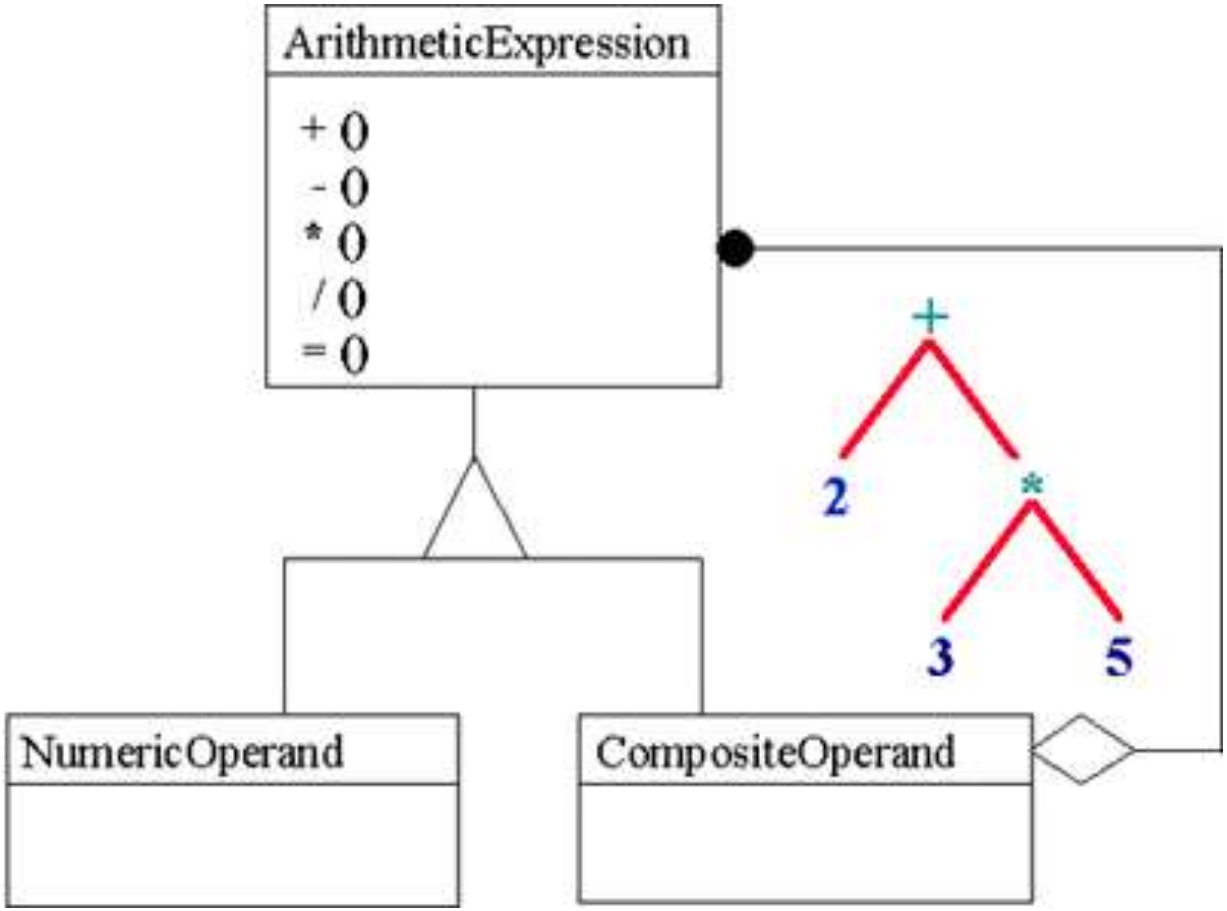
Composite – Example



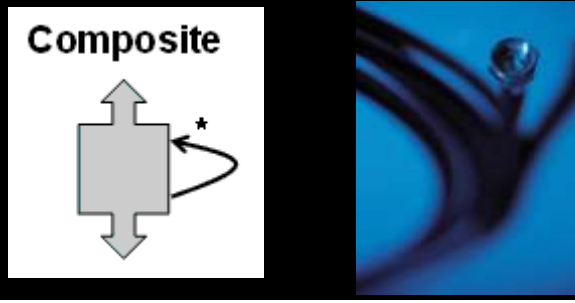


Composite – non-software example





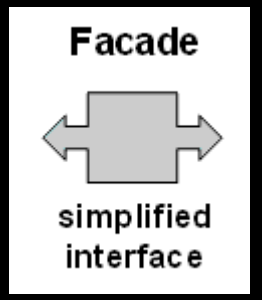
Composite - Consequences



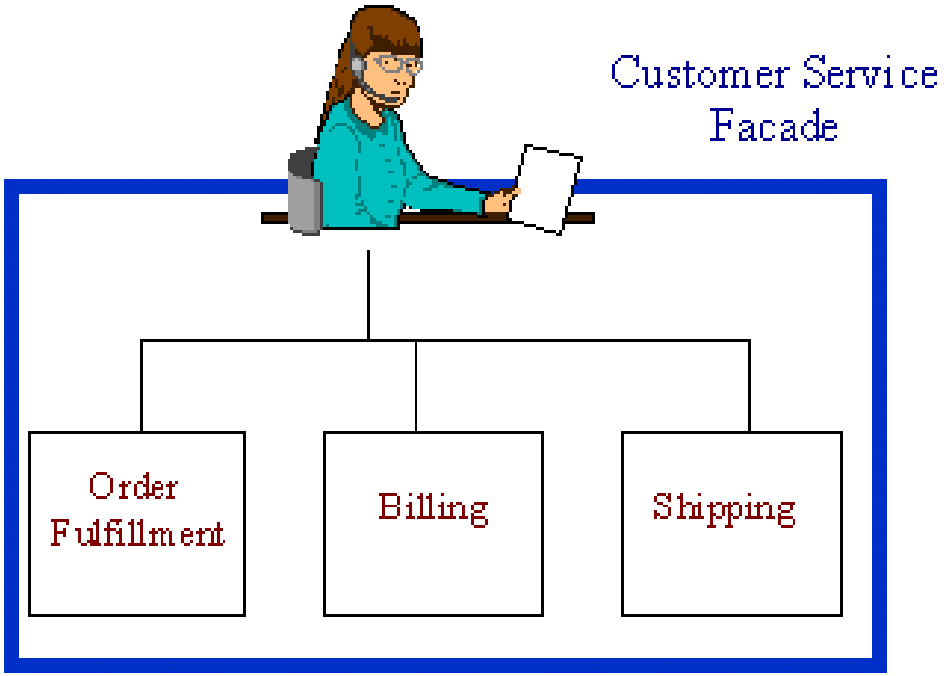
- Providing a common interface to objects that make up the tree structure.
- The client does not need to be interested in the hierarchy of components.
- Components can recursively delegate client request processing up or down the hierarchy.
- Specific components can implement their own unique operations (but for simplicity, they can be moved to more general classes)

- Any composite class can be a child of a composite object.
  - The introduction of stricter rules requires a code conscious of the types involved (e.g. **instanceof** clauses)

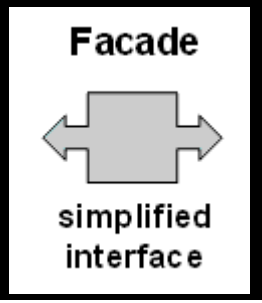
Facade



- Provide a simplified interface to a set of interfaces in a subsystem.
- Delivering one object outside to allow communication with a set of classes.

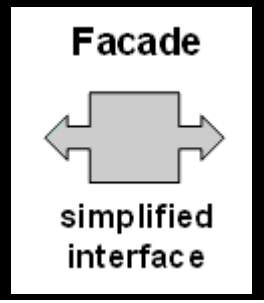


Facade - Problem



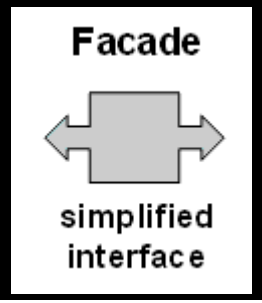
- There are many dependencies between classes implementing abstractions and client classes, noticeably increasing its complexity.
- The need to simplify the client (e.g., reducing the risk of errors).
- Increase the degree of reuse of the system or library.
- Designing classes to work in clearly separated layers.

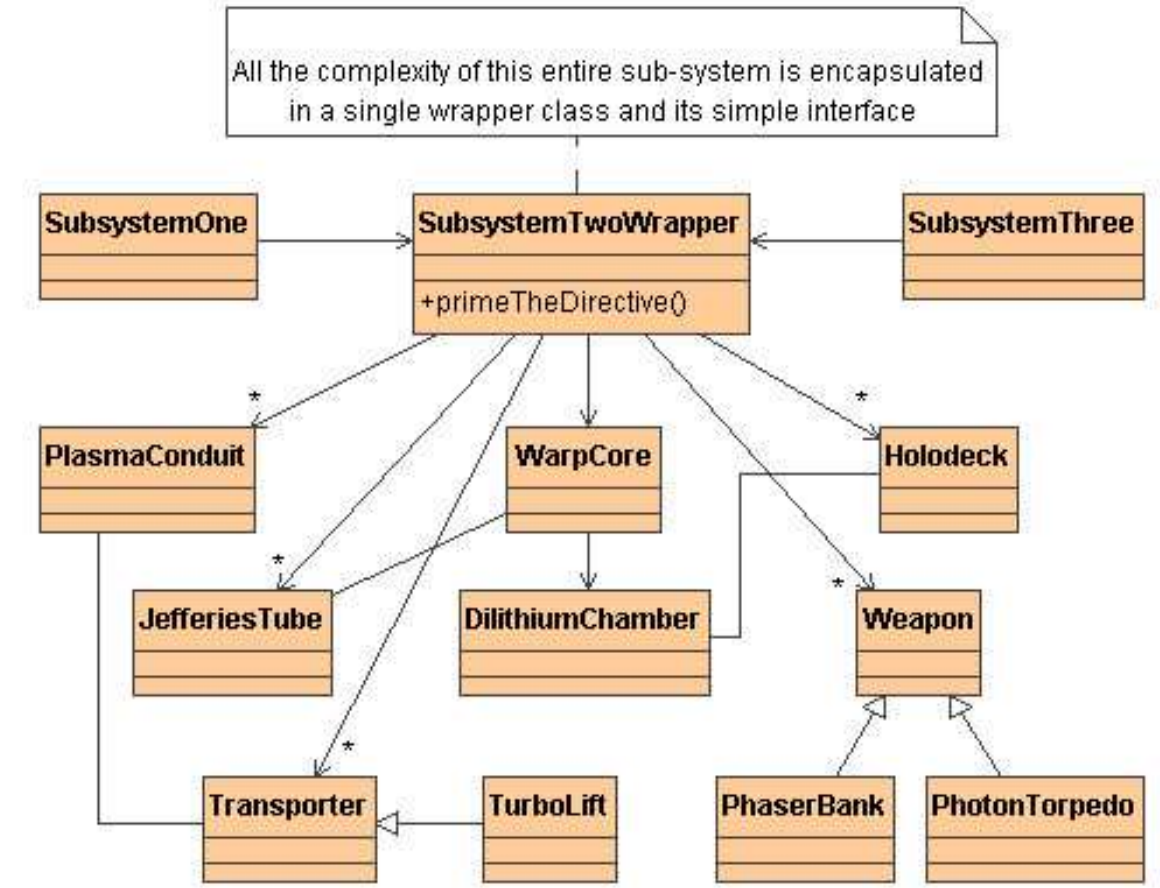
Facade - Solution



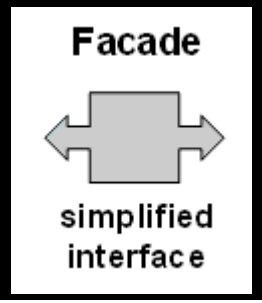
- Wrapping the existing system with a **new** interface.
- A simple entry point for a large subsystem.
- Adding an intermediate layer that hides the complexity of the legacy system

Facade – Class diagram

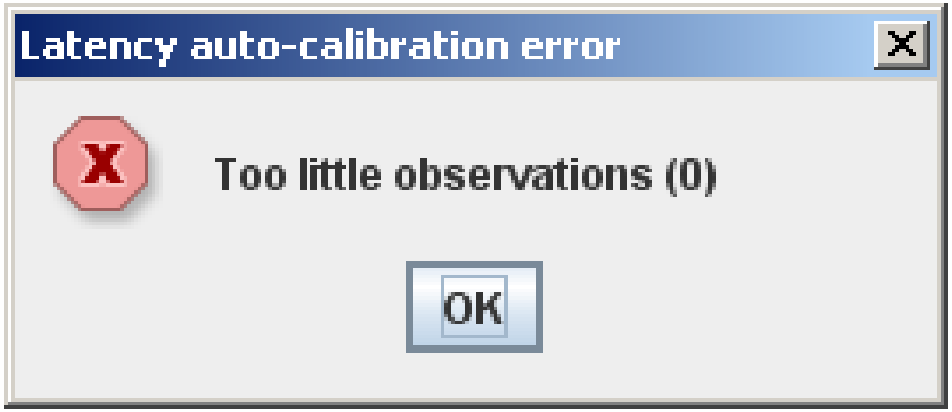




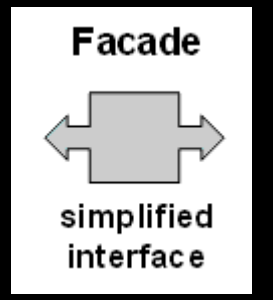
Facade – example



- Java – class JOptionPane
  package javax.swing
- C# - class MessageBox
  package System.Windows.Forms

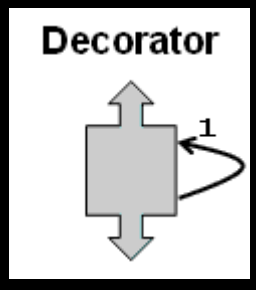


Facade - Consequences



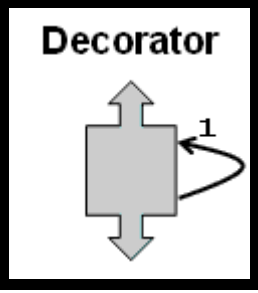
- Inserting the facade classes simplifies the client by shifting client dependencies to the facade.
- The client does not need to know the classes behind the facade.
- Changing the implementation (e.g., fixing) of the classes behind the facade, i.e. those that implement abstractions, is possible without affecting the client's code.

Decorator



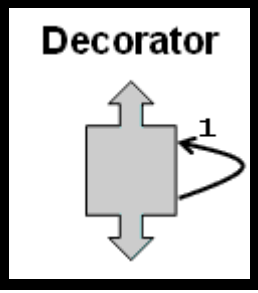
- Attach additional responsibilities to an object dynamically.
- Provide a flexible alternative to subclassing for extending functionality.

Decorator - Problem



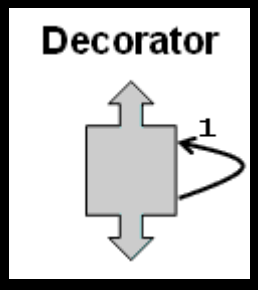
- You want to add behavior or state to individual objects at run-time.
- Inheritance is not feasible because it is static and applies to an entire class.

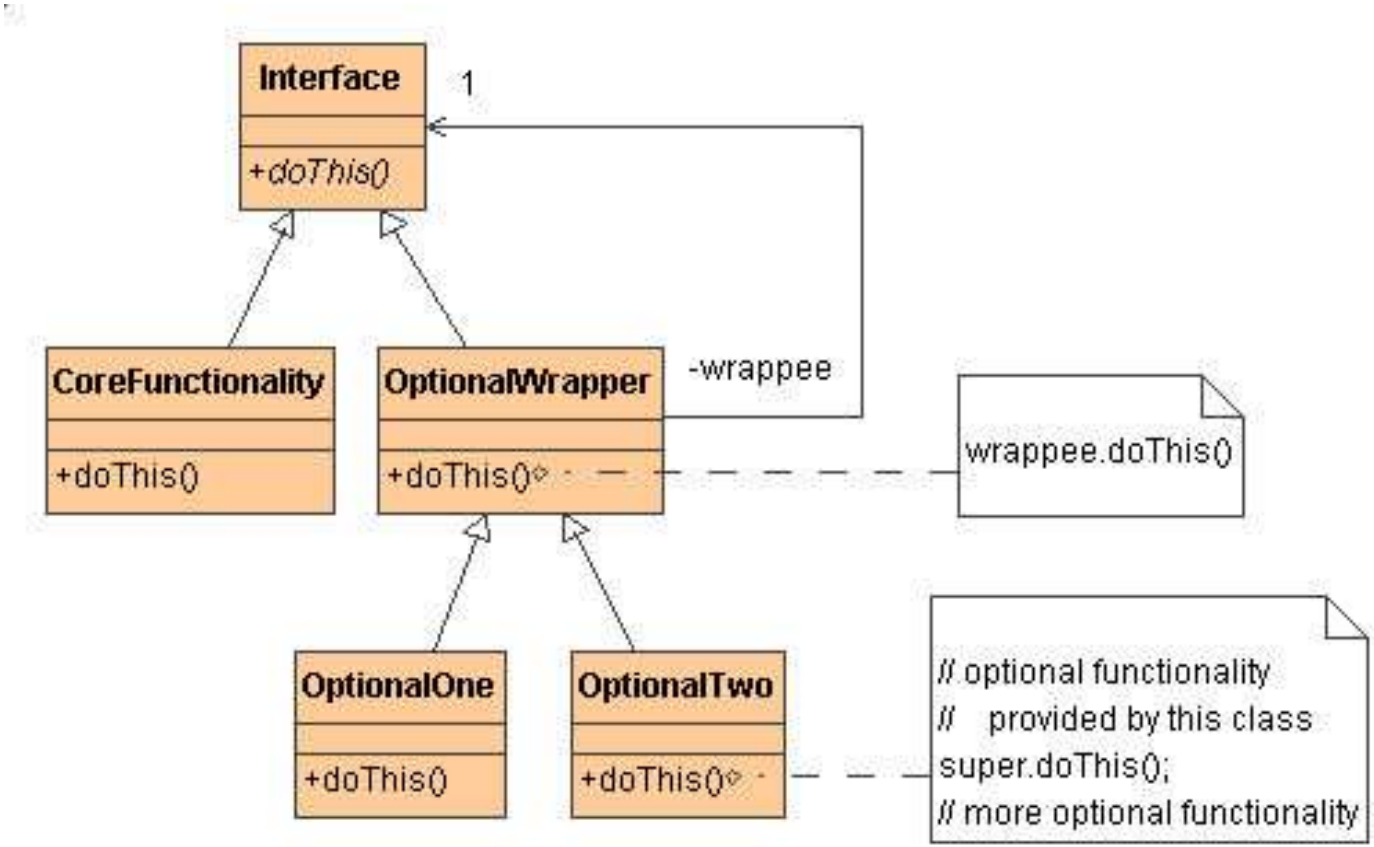
Decorator - Solution



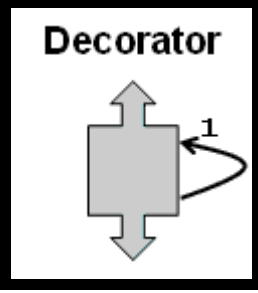
- We introduce a single core object shielded by as many optional objects as possible.
- We use a recursive composition.
- Aggregation 1 to 1: "folds" up the hierarchy of inheritance

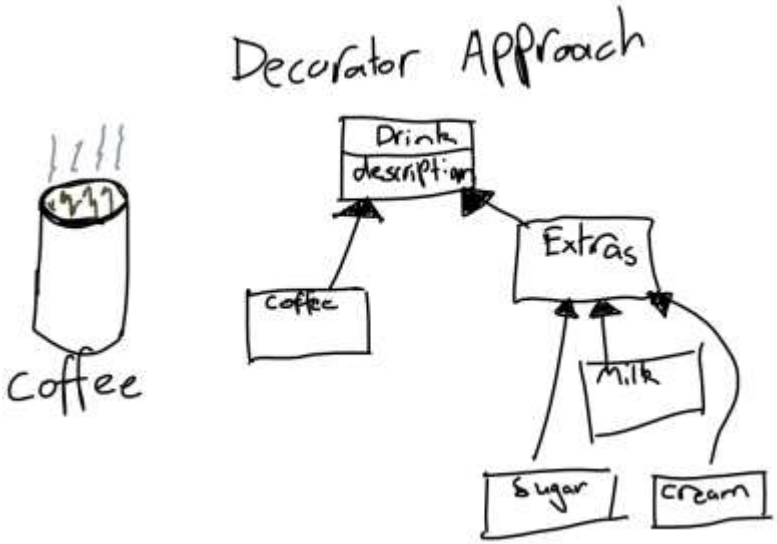
Decorator – Class diagram

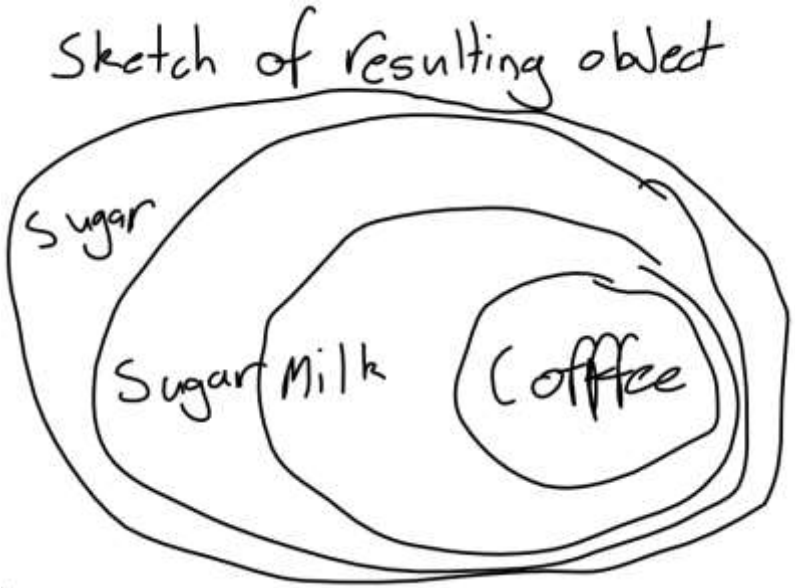




Decorator – coffee example

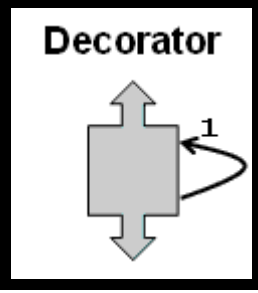


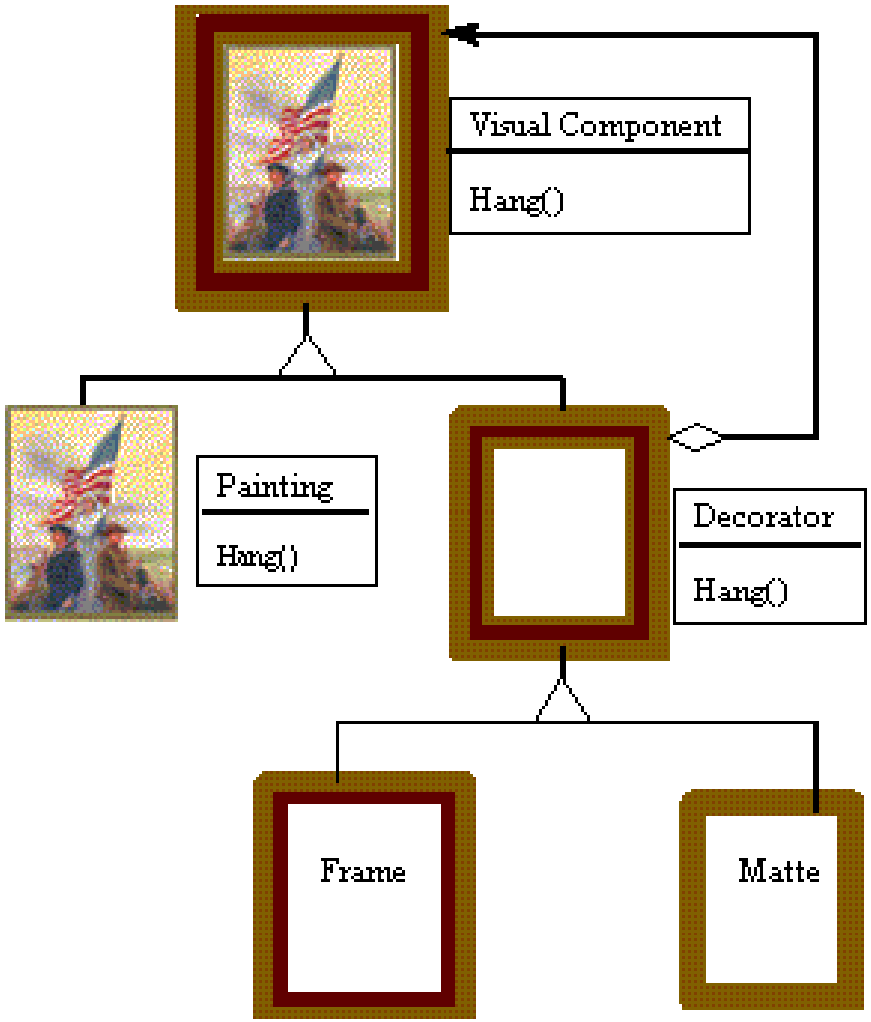




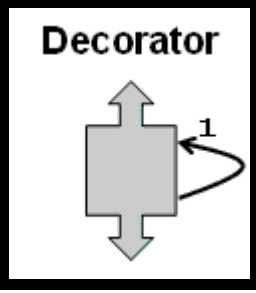
https://tylercash.xyz/post/a-look-at-decorator-patterns

Decorator – non-software example





Decorator – Consequences



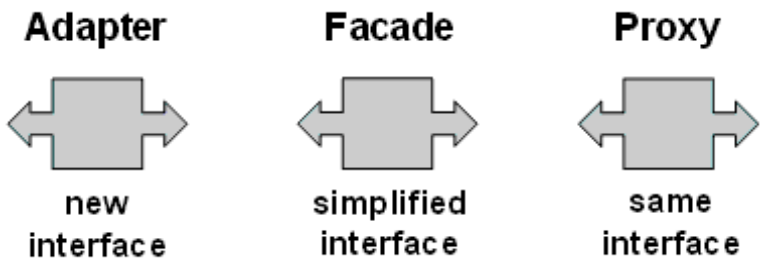
- It provides more flexibility than inheritance.
- By using different combinations of wrappers, a lot of different combinations of behaviors are created.
- Fewer classes, at the expense of more objects.
- The flexibility of wrappers rises the risk of errors.
- It is difficult to determine the identity of the object, because the true objects are hidden inside wrappers.

Composite and Decorator

- **Composite** and **decorator** organize any number of objects using recursive composition.
- **Decorator** can be seen as a degenerate **composite**, with one component, but its purpose is not aggregation.
- **Decorator** and **composite** are often used together because they complement each other.

Patterns vs interfaces

- **A given interface** is:
  - **New**, completely defined – ***Facade***
  - **Old** (for client), reuse – ***Adapter***
  - **Different** (than legacy) – ***Adapter***
  - **Extended** – ***Decorator***
  - **The same** – ***Decorator*** and ***Proxy***

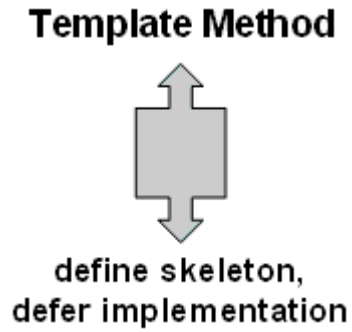


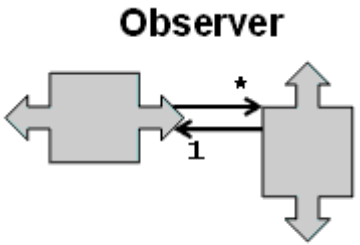
Behavioral patterns

Gang of Four

Roadmap

- Strategy
- Template Method
- Observer
- Iterator





Basic concepts

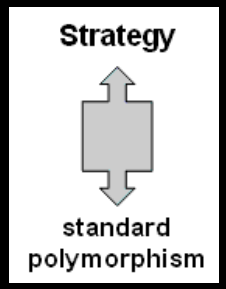
- algorithm…
- operation …
- method …
- signature…
  - polimorphism?
  - class vs interface inheritance?

Behavioral patterns

- **Behavioral patterns** are concerned with the assignment of responsibilities between objects, or, encapsulating behavior in an object and delegating requests to it.

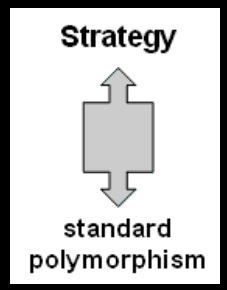
- Organization, management, combining of behavior

Strategy



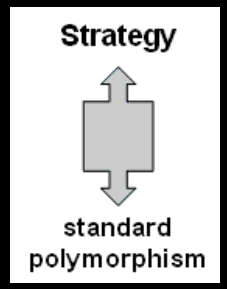
- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from the clients that use it.

Strategy – Problem



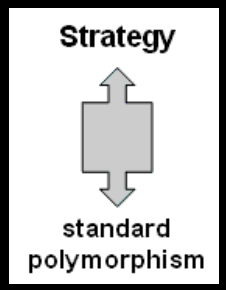
- The complexity of the code resulting from the existence of many strategies for a specific problem.
- The need to build object-oriented software with a minimized number of dependencies.

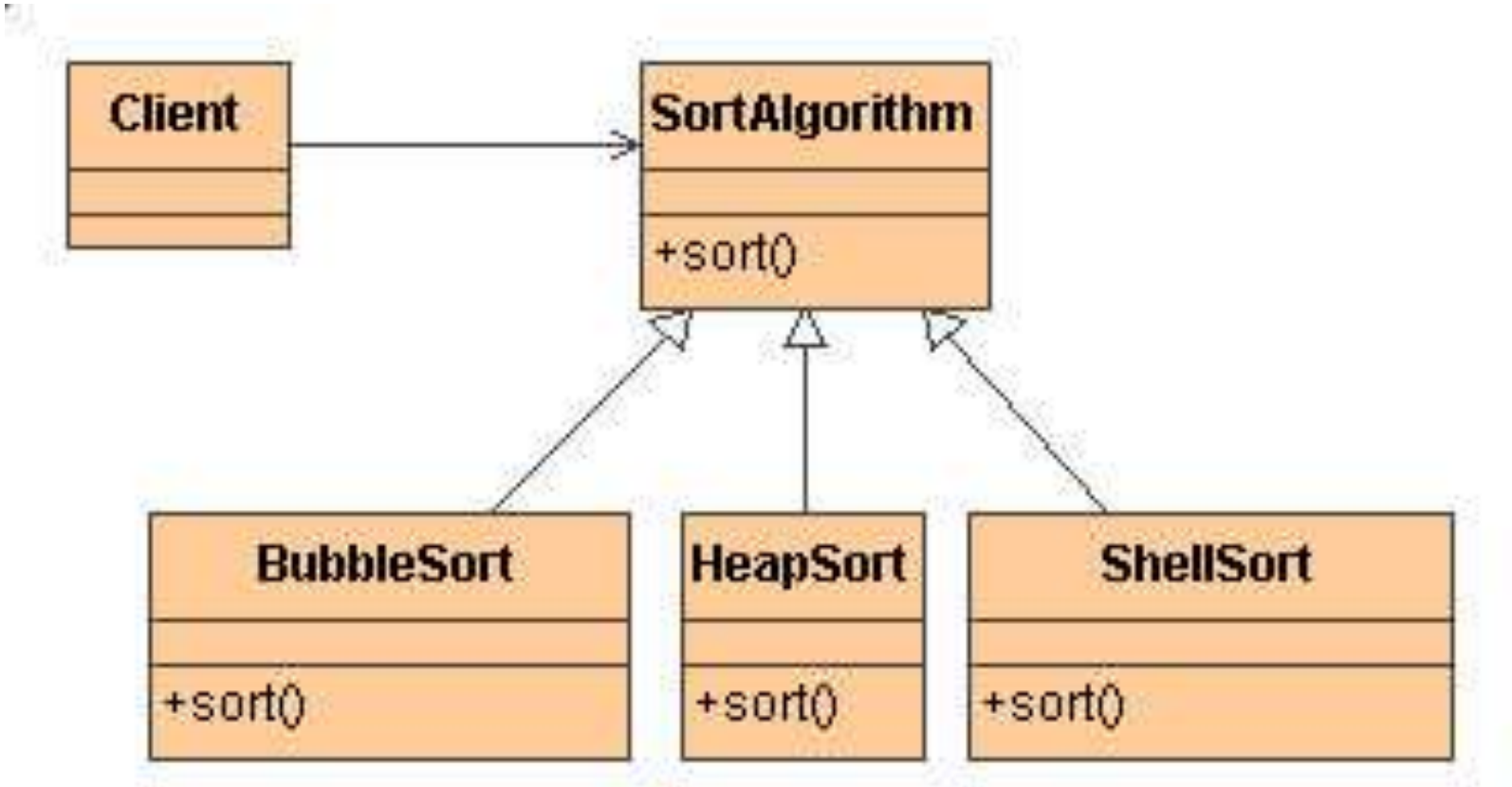
Strategy - Solution



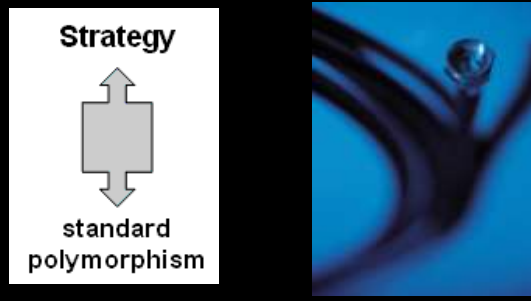
- Provide a way to configure the algorithm selection
- Wrapper / delegation structure
  - the client is a wrapper,
  - the algorithm object is a delegation.
- Adding an intermediate level for the client (couple the client to the interface).

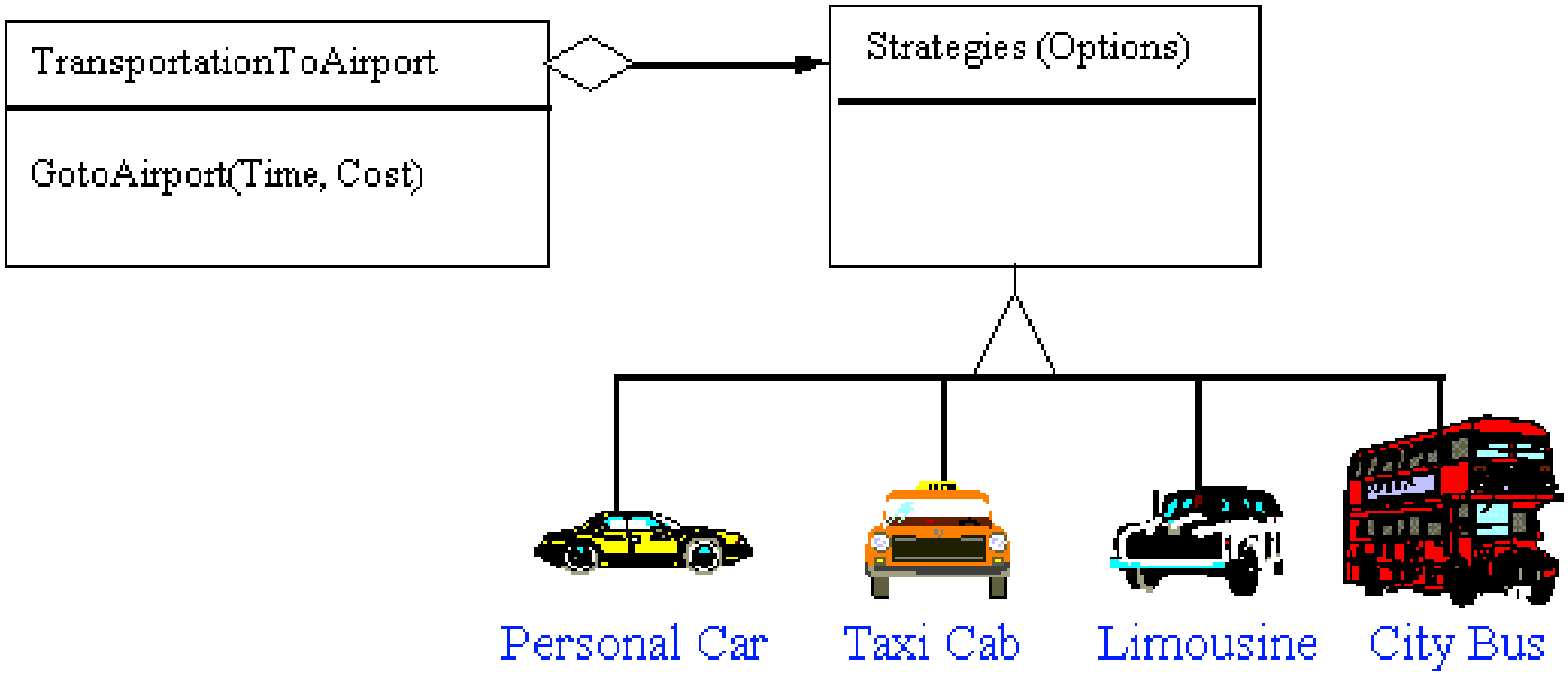
Strategy – Class diagram



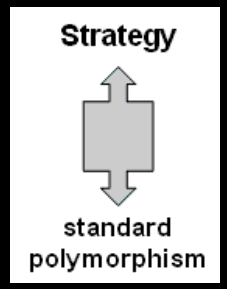


Strategy – non-software example





Strategy - Consequences



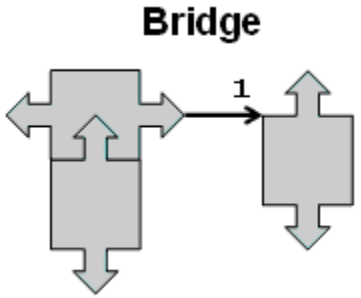
- Behavior of client objects can be specified using objects.
- The pattern simplifies the client's classes by exempting them from the responsibility of choosing behavior or implementing alternative behaviors.
- Simplifies code for client objects by eliminating *if* and *switch* statements.
- In some cases, it can increase the speed of client objects because they do not need to choose behavior.

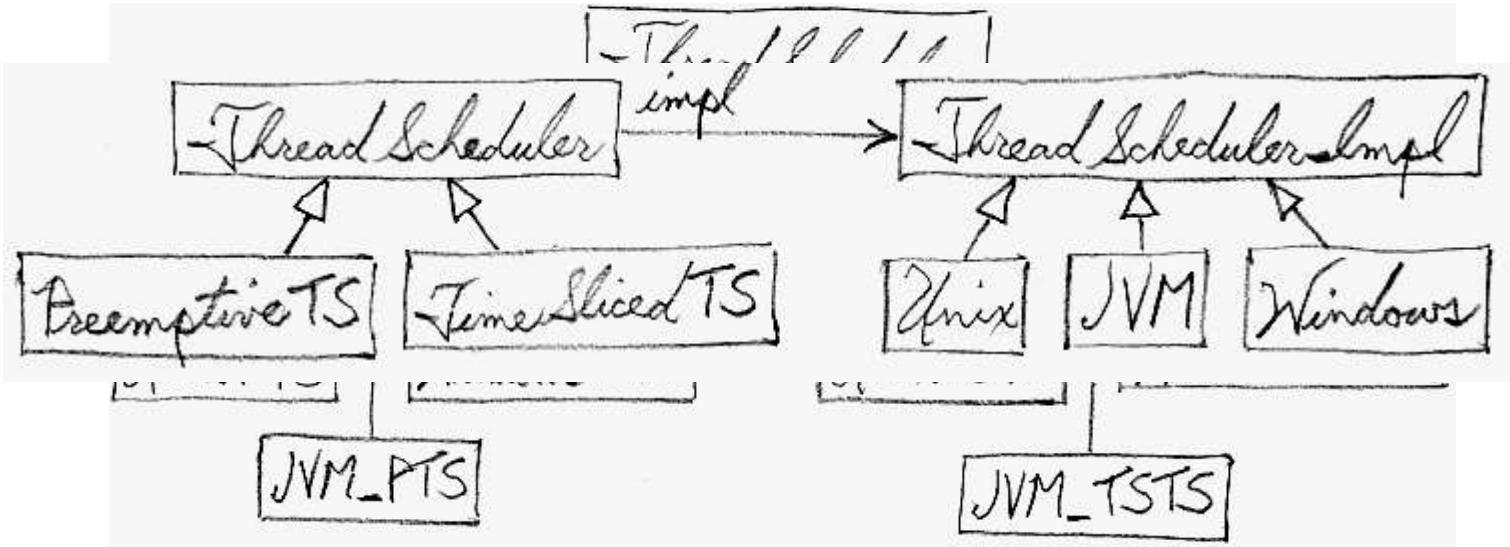
Code-smell?

- A symptom in the source code that is likely to point to a deeper problem.
- It is not a mistake in itself - it is a warning bell that reveals places that may prove problematic when developing the code.
- e.g. code repetition, long functions, large classes, extensive switch statements …

Strategy in the Bridge pattern

- Implementation of a multiplatform mechanisms for thread scheduling.

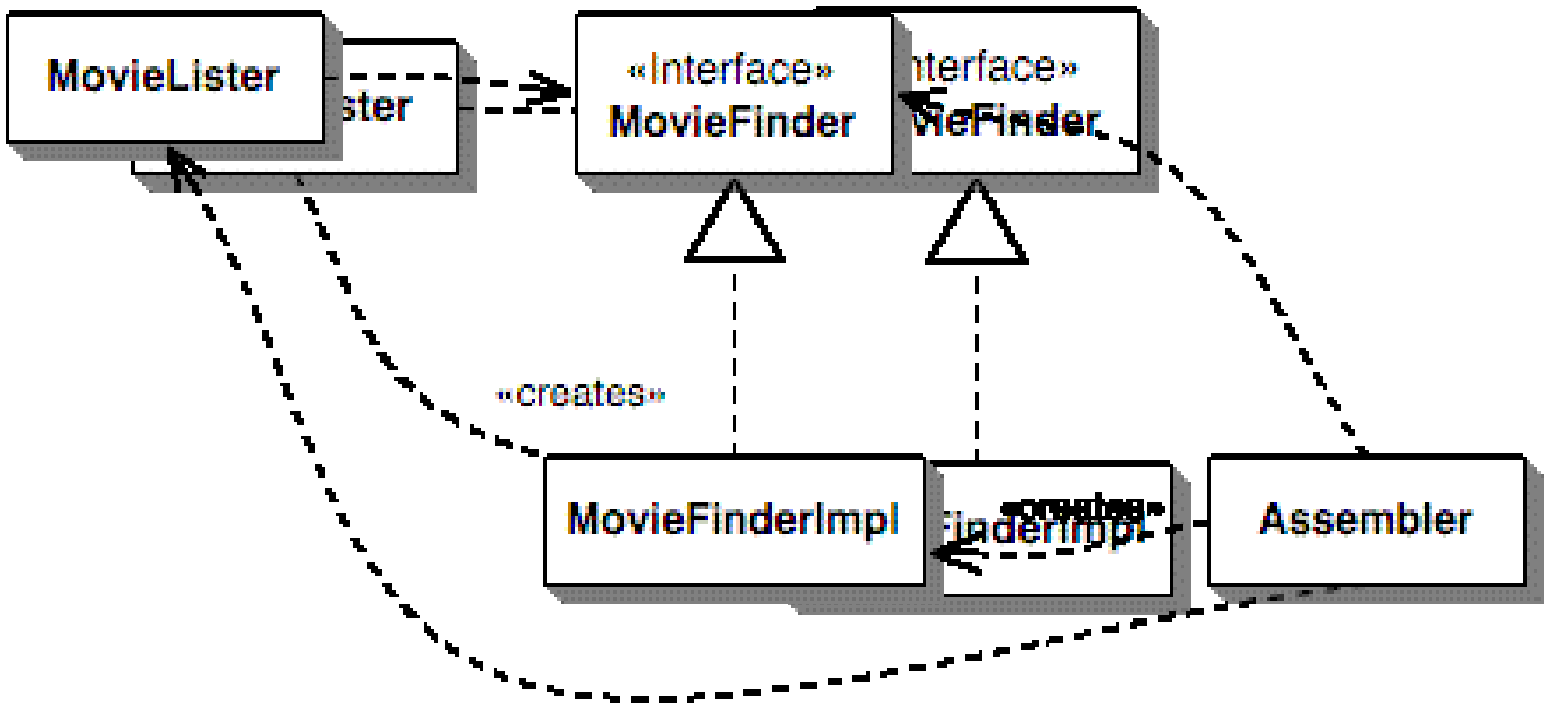




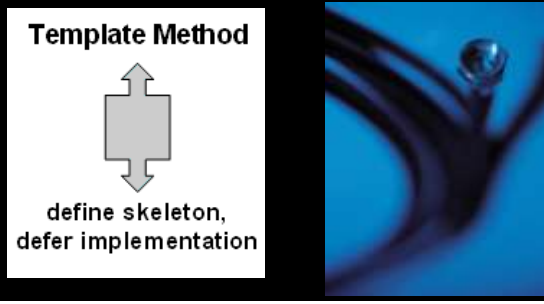
Configuration and Dependency Injection

- Inversion of Control (Hollywood Princilple) – "do not call us, we will call you".
  - Framework configures applications and calls utilities components.
  - This is what distinguishes the framework from the library.
- Dependency Injection – a way of reducing dependencies between components only to interfaces

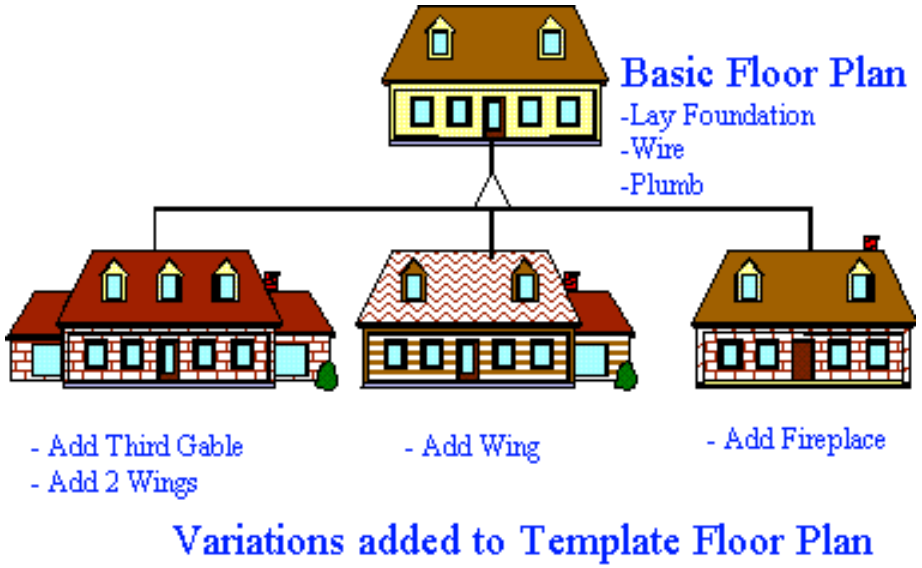
Dependency Injection – Naïve



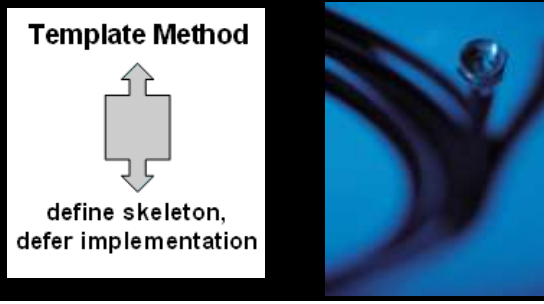
Template Method



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

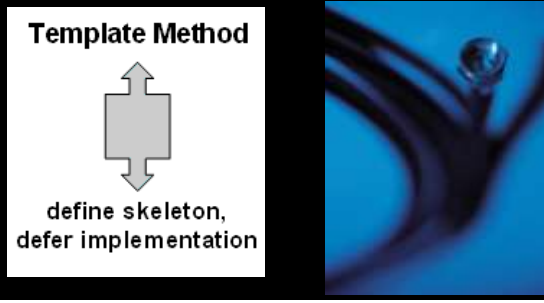


Template Method – Problem



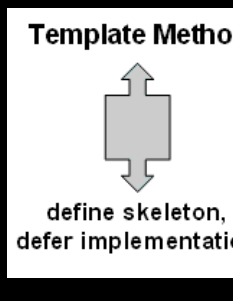
- Two different components have significant similarities, but demonstrate no reuse of common interface or implementation.
- If a change common to both components becomes necessary, duplicate effort must be expended.

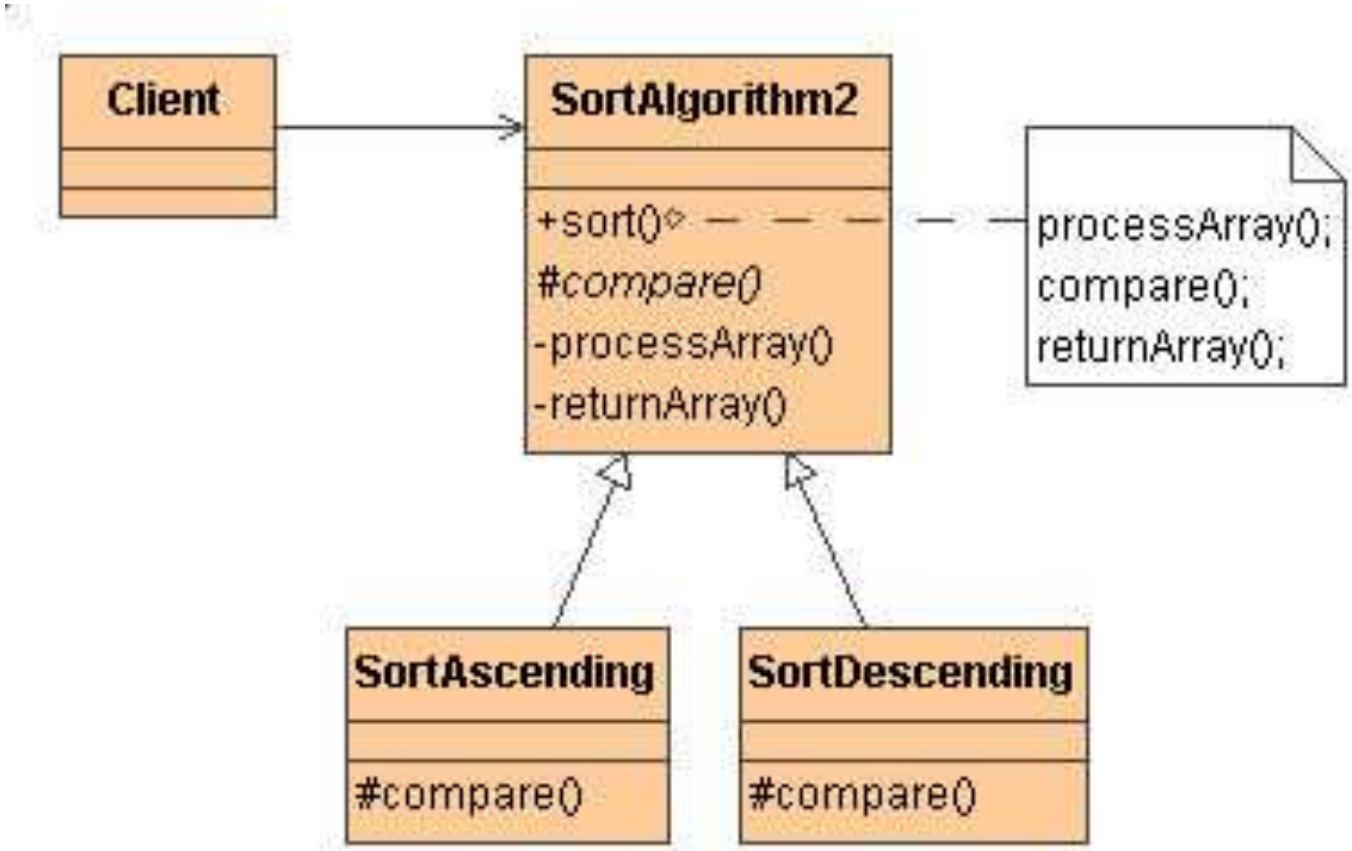
Template Method – Solution



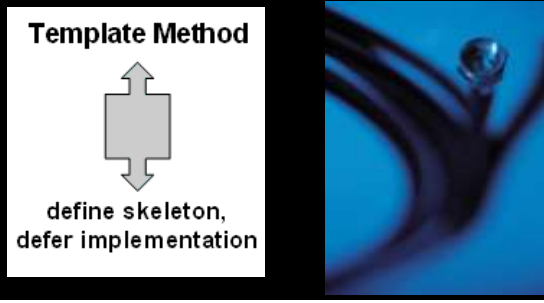
- Provide a way to configure the algorithm step.
- The algorithm step **defined** in the **base class** is left to be **implemented** in **derived classes**.

Template Method – Class Diagram





Template Method – Consequences



- A programmer writing a subclass of an abstract template class is forced to override those methods whose implementation is necessary to complete the superclass logic.
- Well-designed class template has a structure that provides a developer guidelines for the basic structure of its subclasses.

Patterns quiz – *Strategy* or *Template Method*?

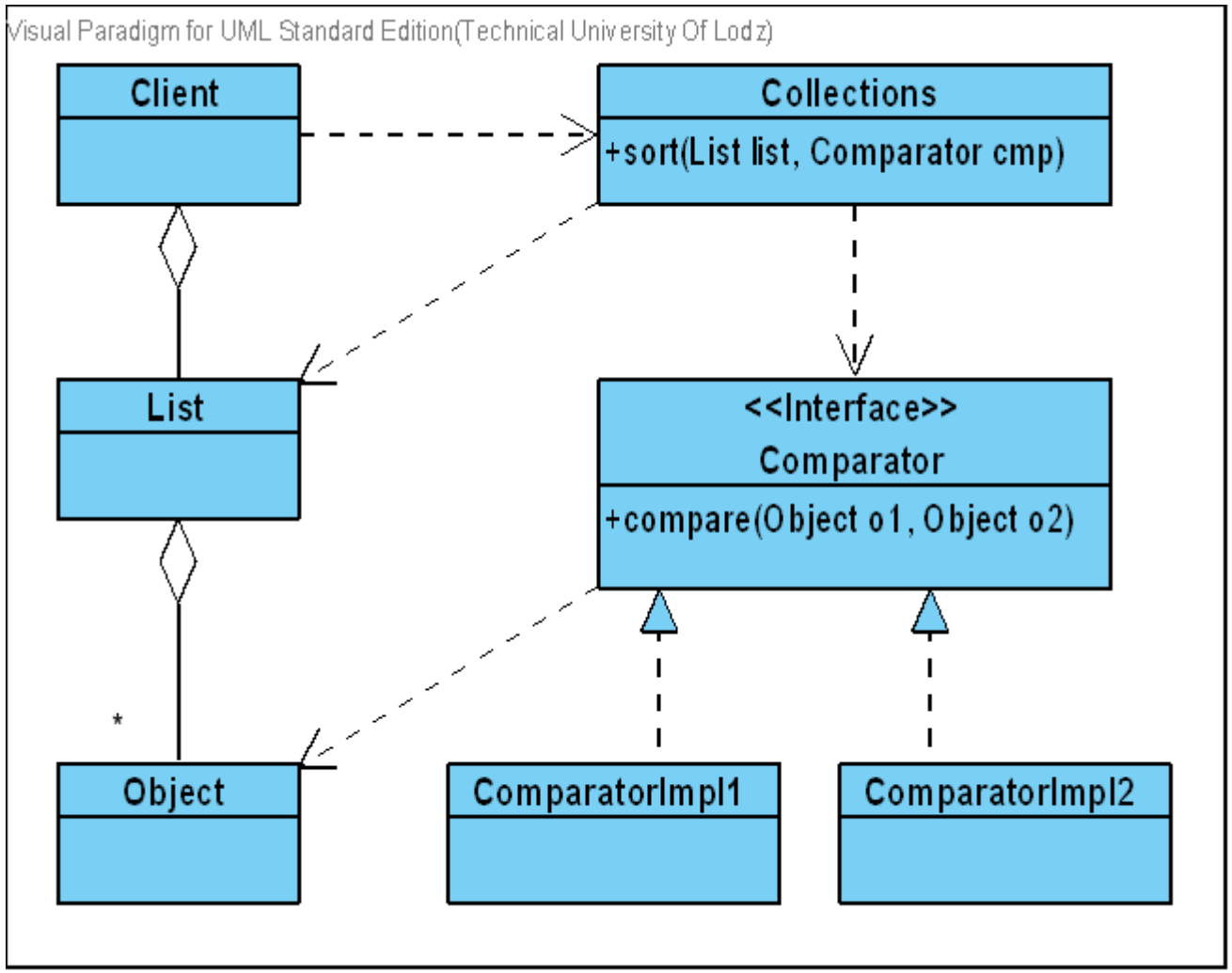
Strategy

standard polymorphism

?

Template Method

define skeleton, defer implementation



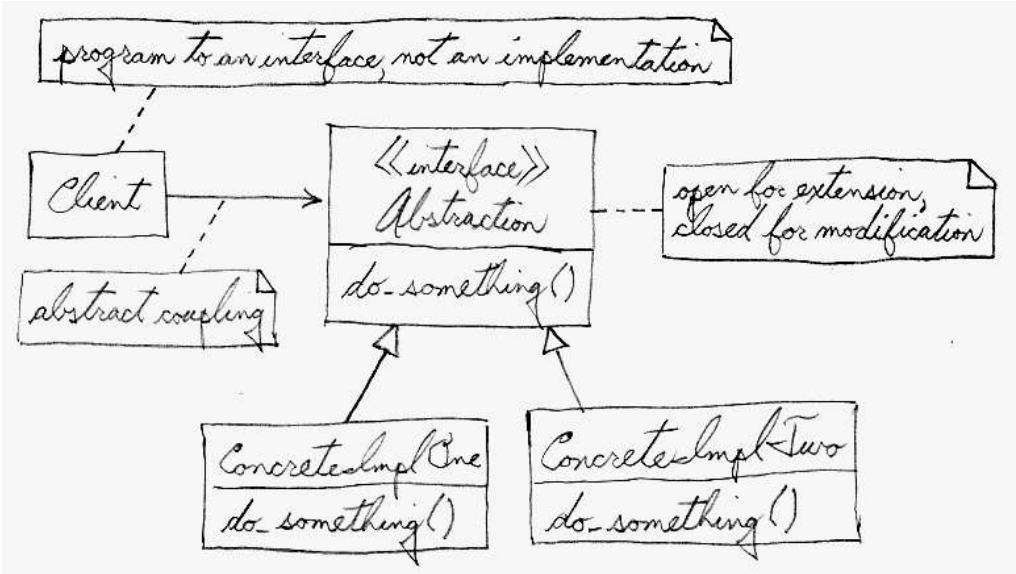
Strategy vs Template Method

- The change is made:
  - in the **part** of the algorithm through **inheritance** - ***Template Method***,
  - the **entire** algorithm through **delegation** - ***Strategy***.
- The logic is modified:
  - whole class - ***Template Method***,
  - individual objects - ***Strategy***.

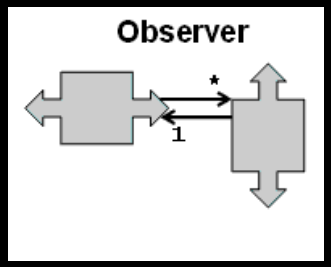
Open-Closed Principle

- *„Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"*

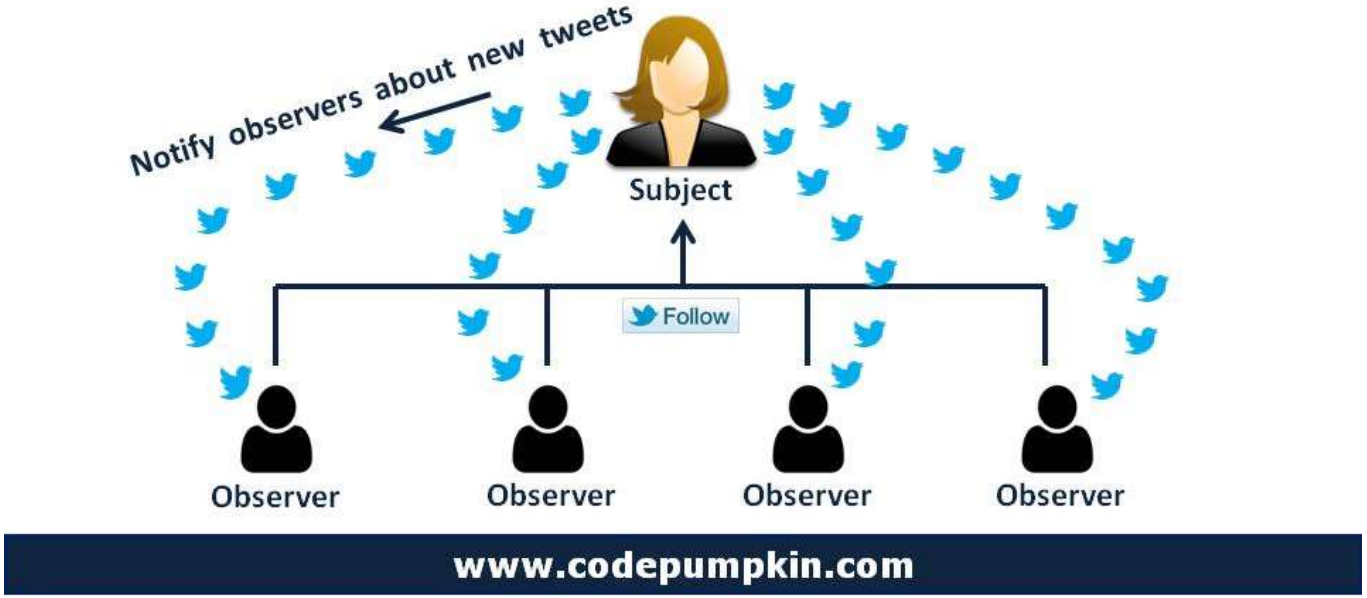
Bertrand Meyer, 1988



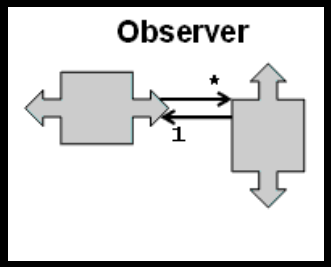
Observer



- It enables the separation of the object of knowledge of the objects of its subsidiaries

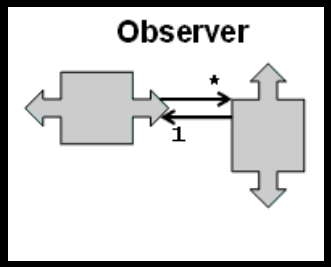


Observer - Problem



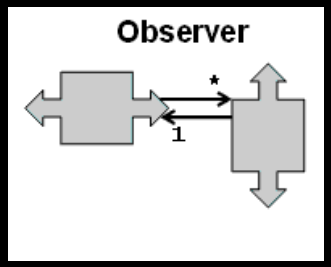
- A large monolithic design does not scale well as new graphing or monitoring requirements are levied.
- An object is burdened with the responsibility of informing its clients about changes to important attributes, even though the client "knows" which attributes are significant to him.

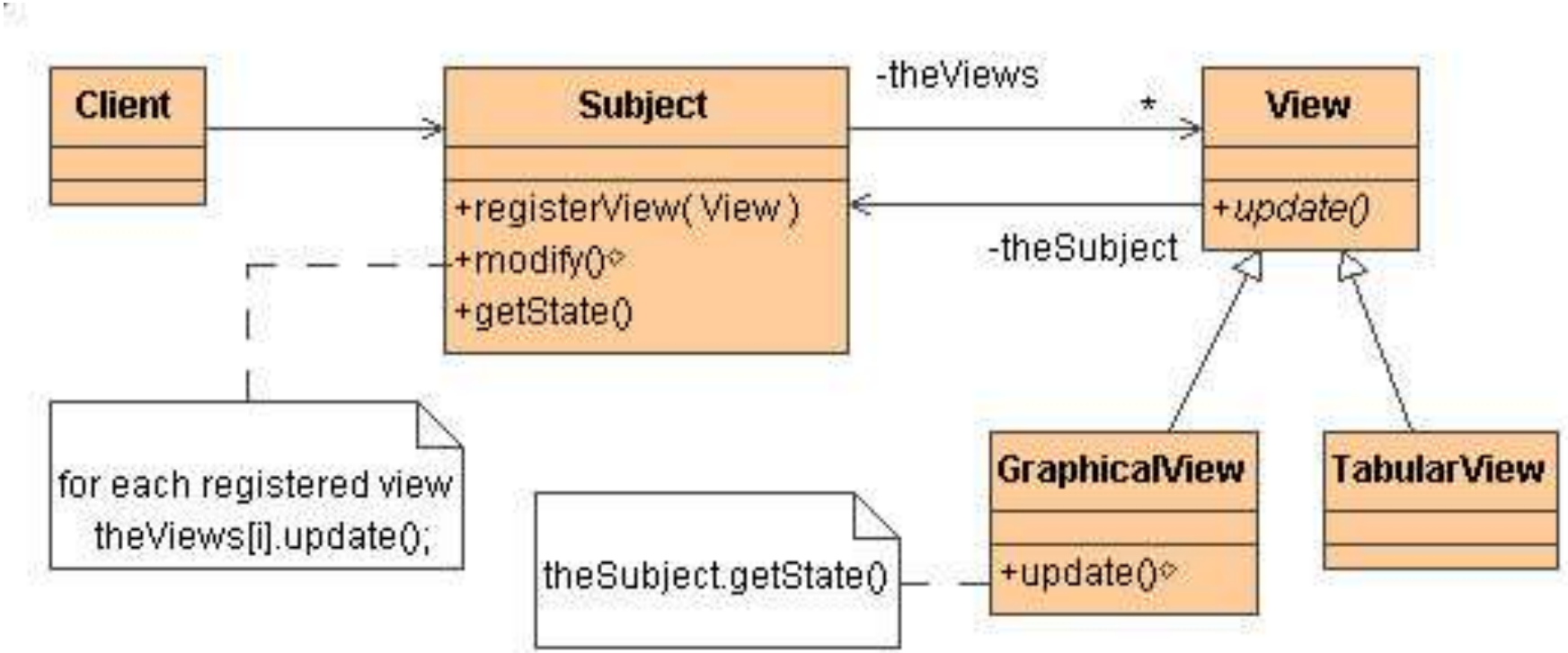
Observer - Solution



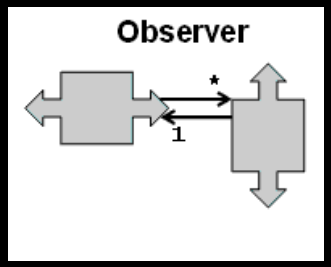
- *Subject* represents the core (of business logic) abstraction.
- *Observer* represents the variable (a user-configurable, optional functionality) abstraction.
- *Subject* prompts *Observer* objects to do their thing.
- Each *Observer* can call back to the *Subject* as needed.
- Structure: Wrapper (*Subject*) / Delegation.

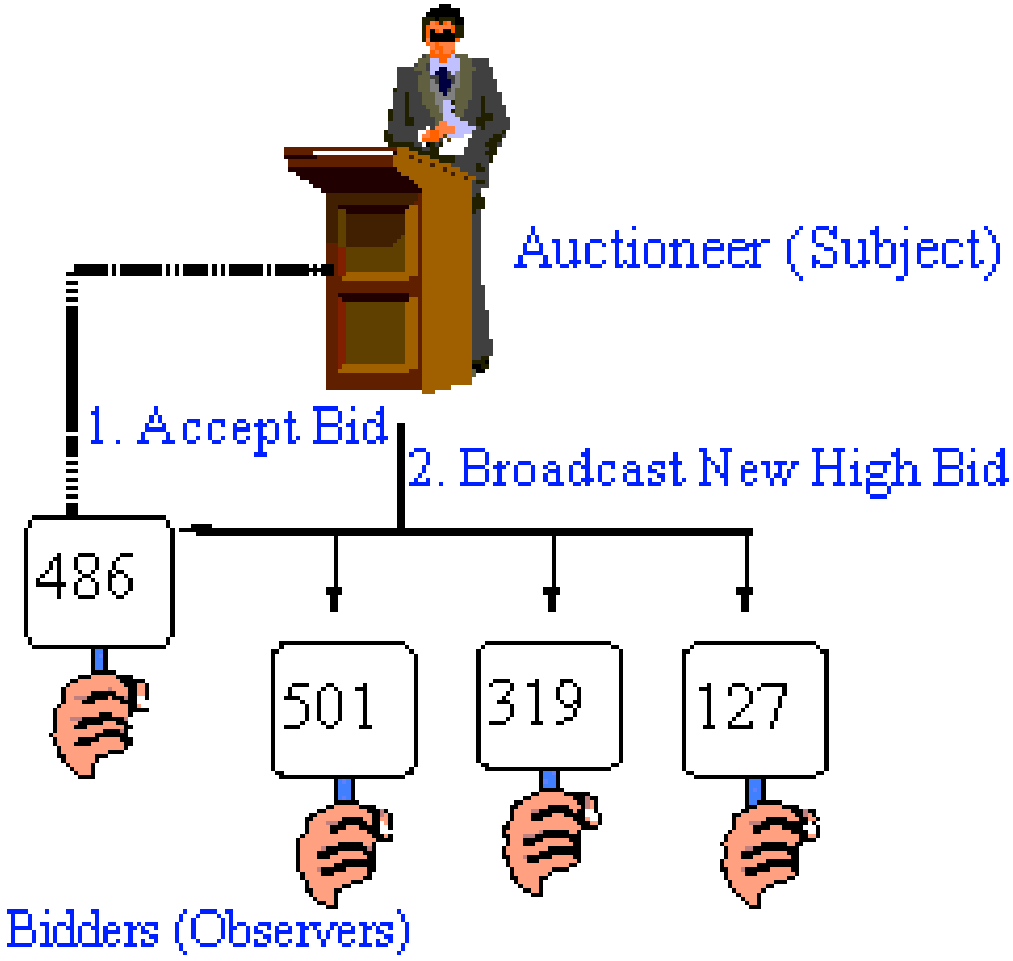
Observer – Class Diagram



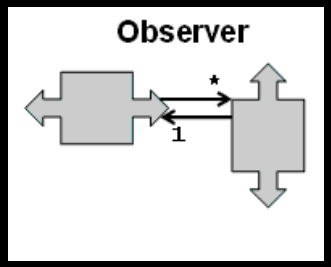


Observer – non-software example



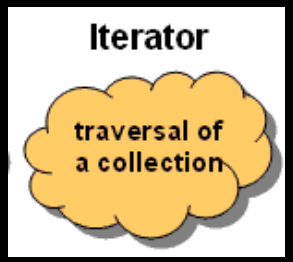


Observer – Consequences

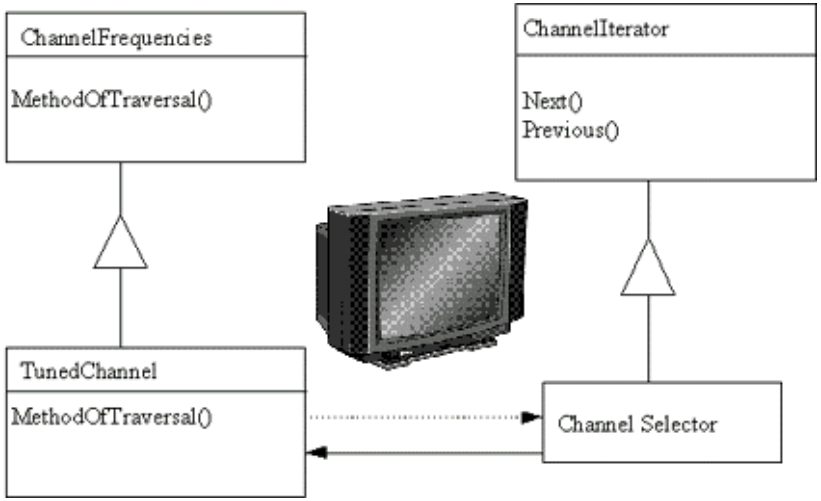


- The object provides notification to other objects without the sender and recipient's awareness of each other.
- Delivery of notifications can take a long time.
- The risk of cyclical notifications.

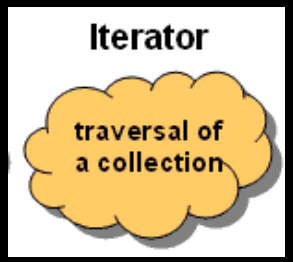
Iterator



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

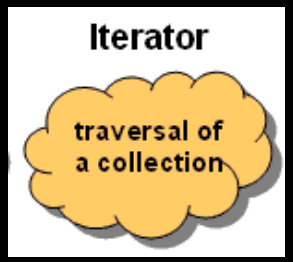


Iterator – Problem



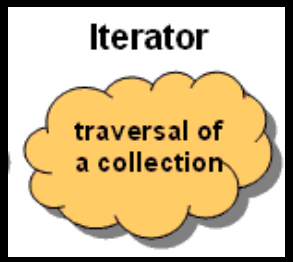
- We need to unify ("abstract") the mechanism of traversing (browsing) widely different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

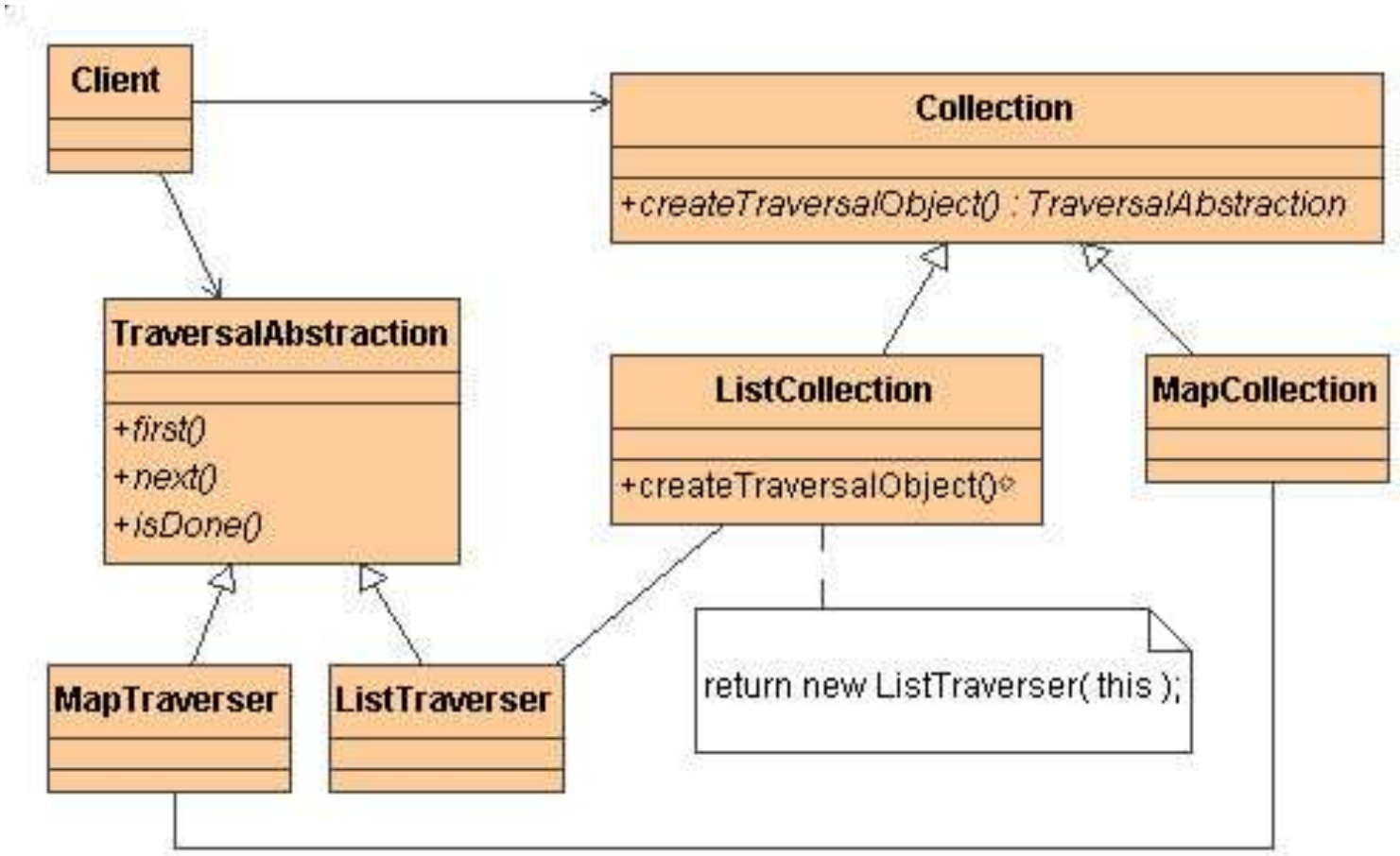
Iterator – Solution



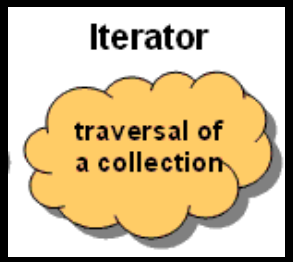
- We apply polymorphism for traversal (iterating).
- Promote to "full object status" the traversal of a collection.

Iterator – Class diagram





Iterator – Consequences



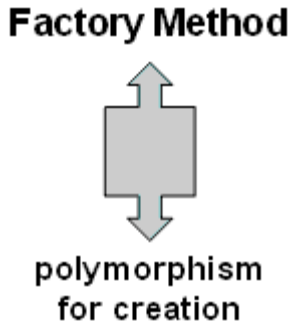
- It is possible to access object collection without knowing the source of the objects.
- Using many iterator objects, it is easy to own and manage many "transitions" at once.
- The collection class can provide different types of iterators to traverse collections in a variety of ways. (e.g. keys and index values)
- *Iterator* can be used for traversing *composites*

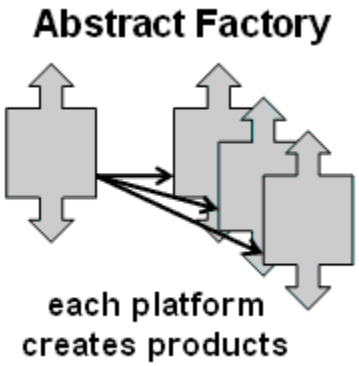
Creational patterns

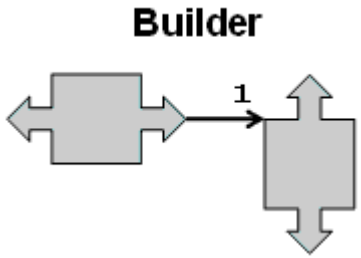
Gang of Four

Roadmap

- Factory Method
- Abstact Factory
- Singleton
- Builder





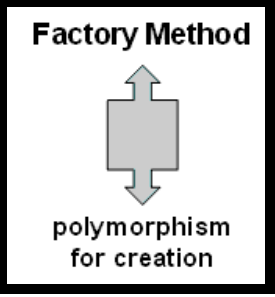


Creational patterns

- **Creational patterns** concern simplifying the process of creating objects when it requires making decisions.

- They allow clients to create new objects differently than just by calling the class constructors.

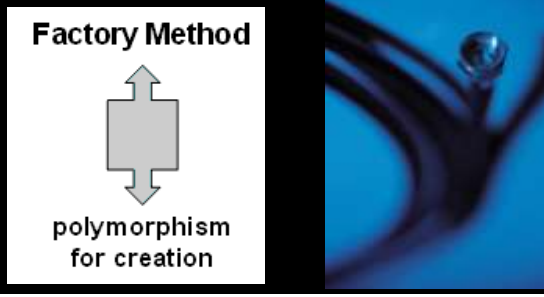
Factory Method



- Release the client from the obligation to "know" a particular class whose instance is to be created.

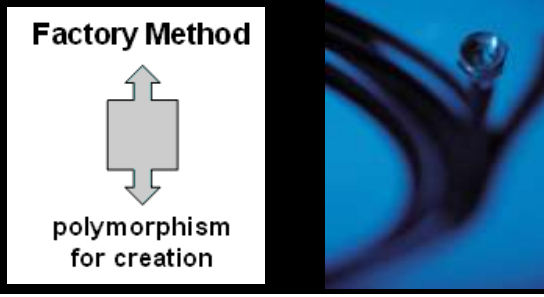
```
Pet p = new Parrot()
```

Factory Method – Problem



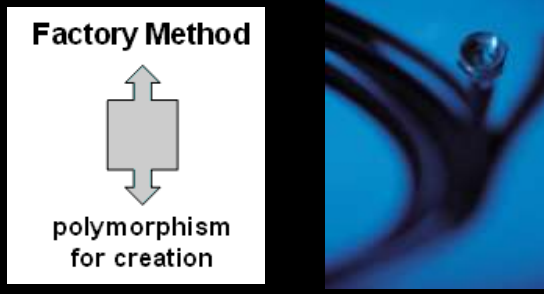
- A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.
- Classes must initiate object creations without having dependencies with the class of the created object.
- The set of created classes can grow dynamically when new classes are made available.

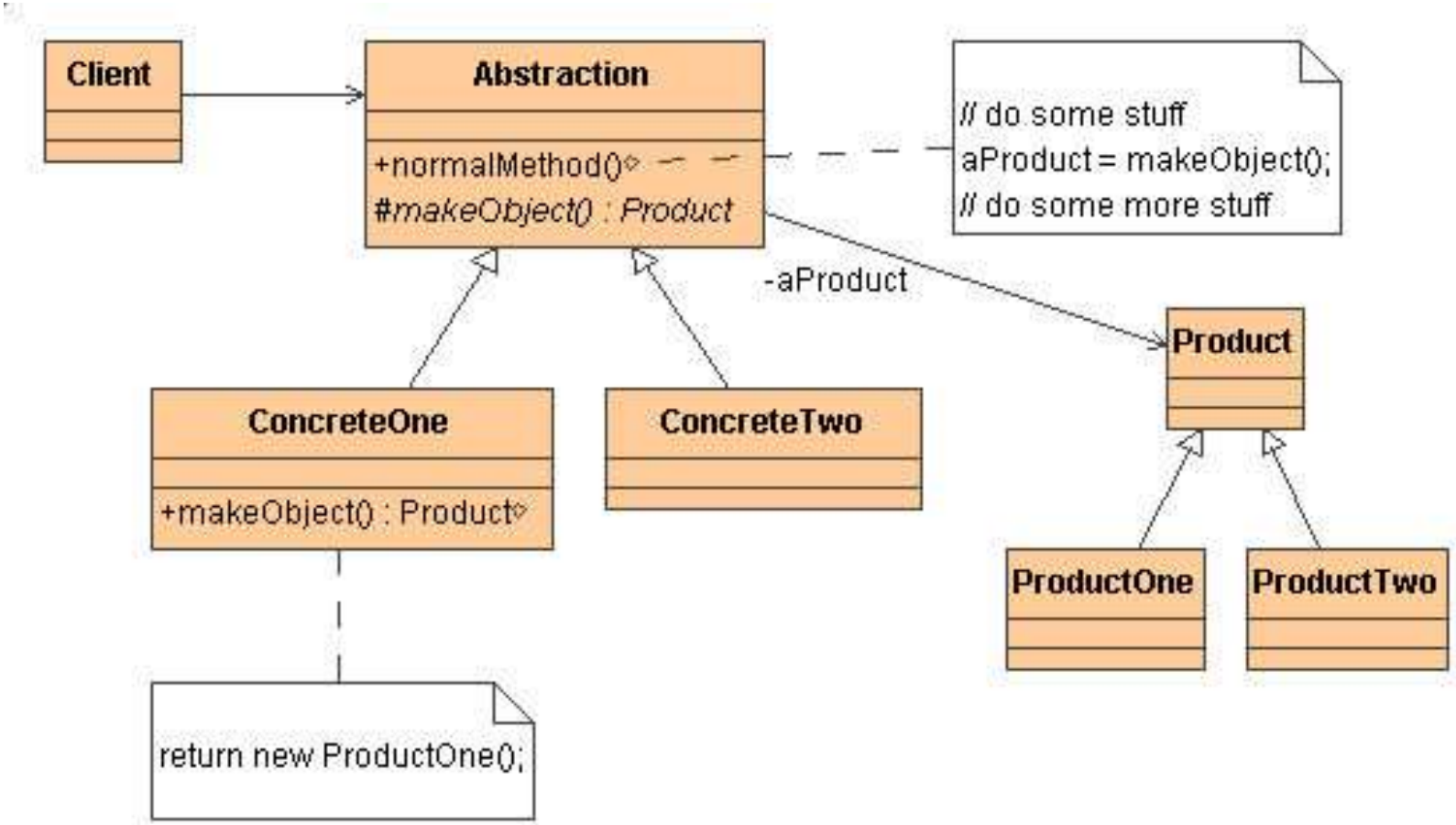
Factory Method – Solution



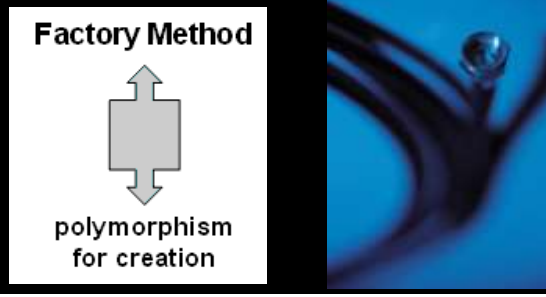
- Indirect creation using inheritance.
- Defining a "virtual" constructor.
- The new operator considered harmful.

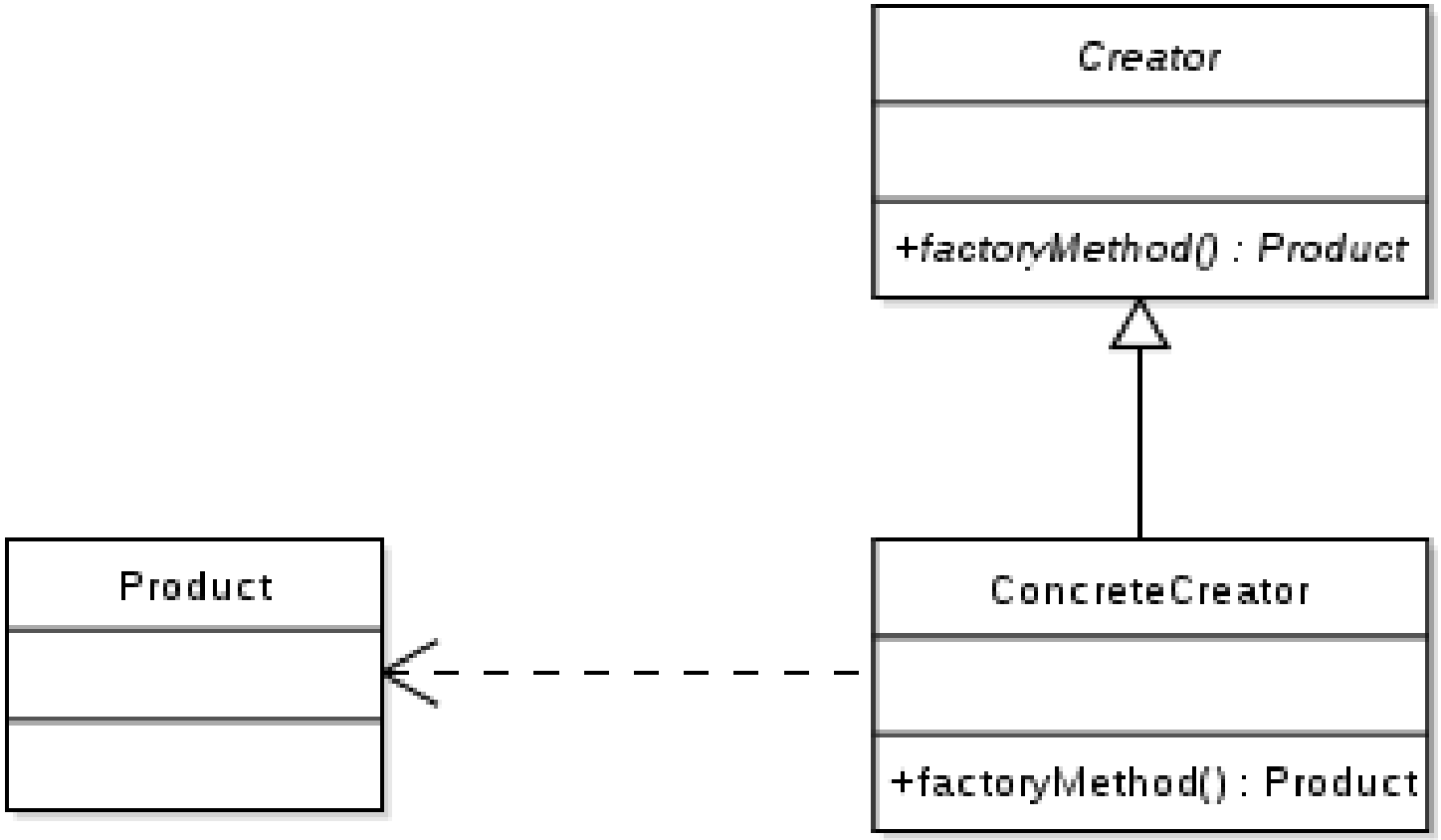
Factory Method – Class Diagram



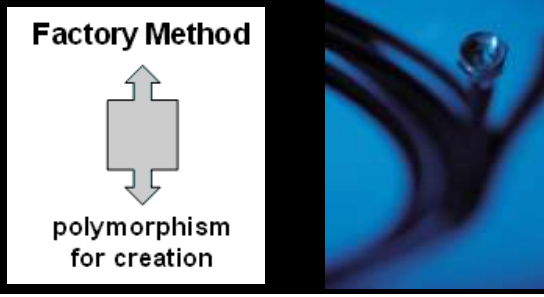


Factory Method – public variant





Factory Method and Iterator

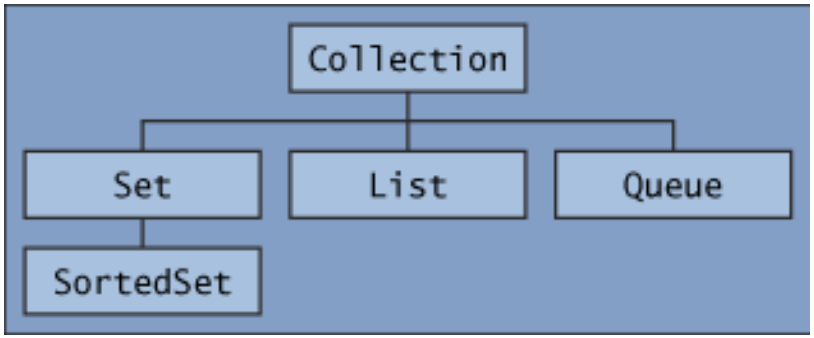


Iterators

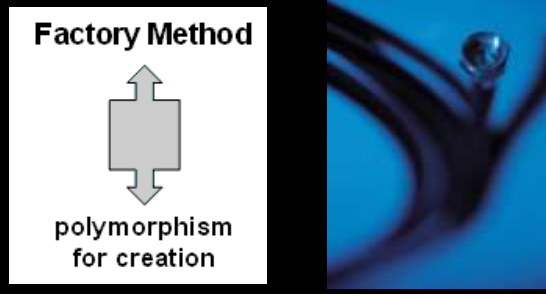
```
static void filter(Collection<?> c) {
        for (Iterator<?> it = c.iterator(); it.hasNext(); )
                if (!cond(it.next()))
                        it.remove();
}
```

for-each Construct

```
for (Object o : collection)
        System.out.println(o);
```

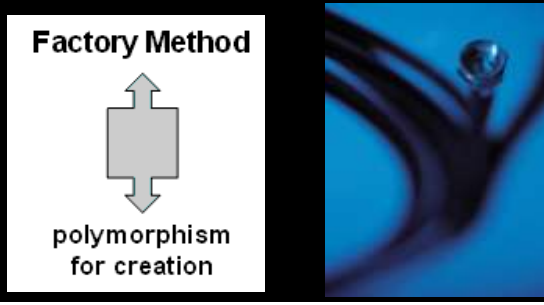


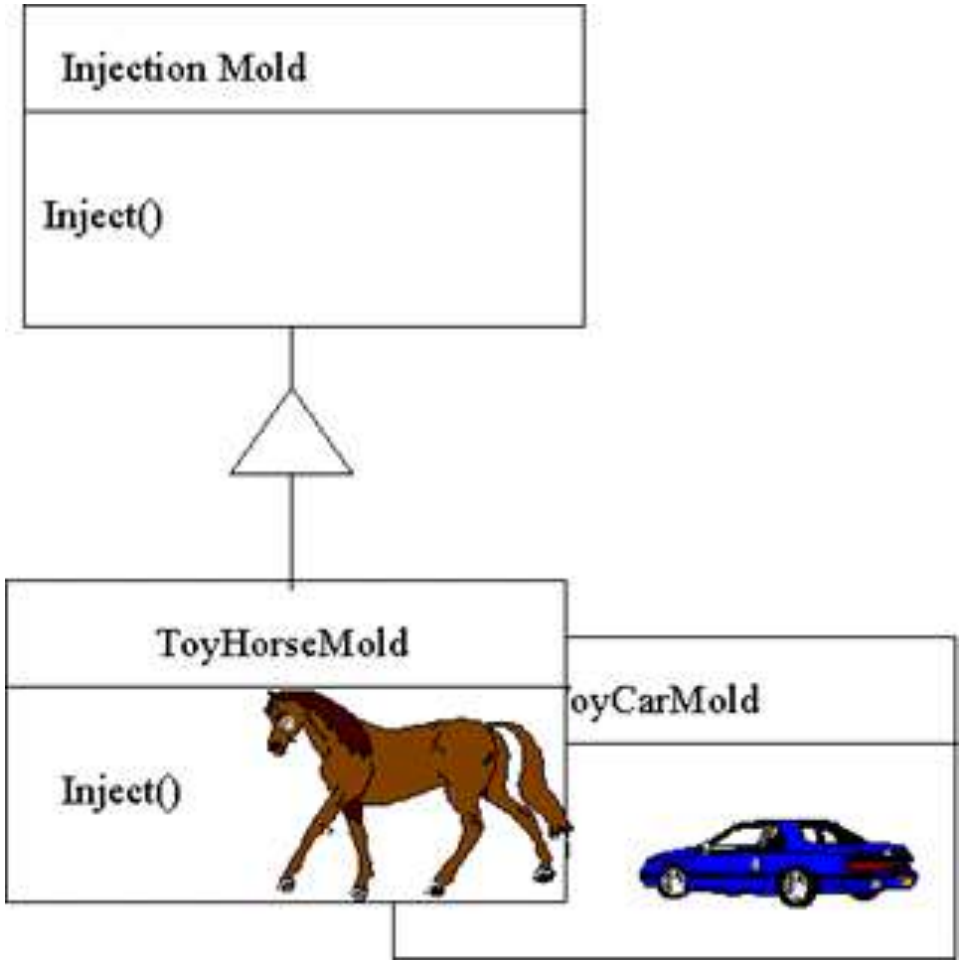
Factory Method – Consequences



- The class asking for object creations is independent of the classes of objects produced.
- The set of product classes that can be created can be dynamically changed.
- An additional intermediate layer between object creation initiation and the determination of which object class will be created makes it difficult for programmers to understand the code.

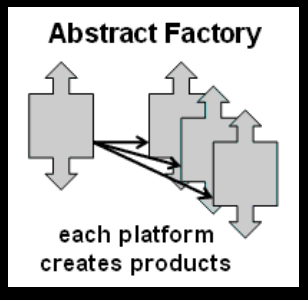
Factory Method – non-software example

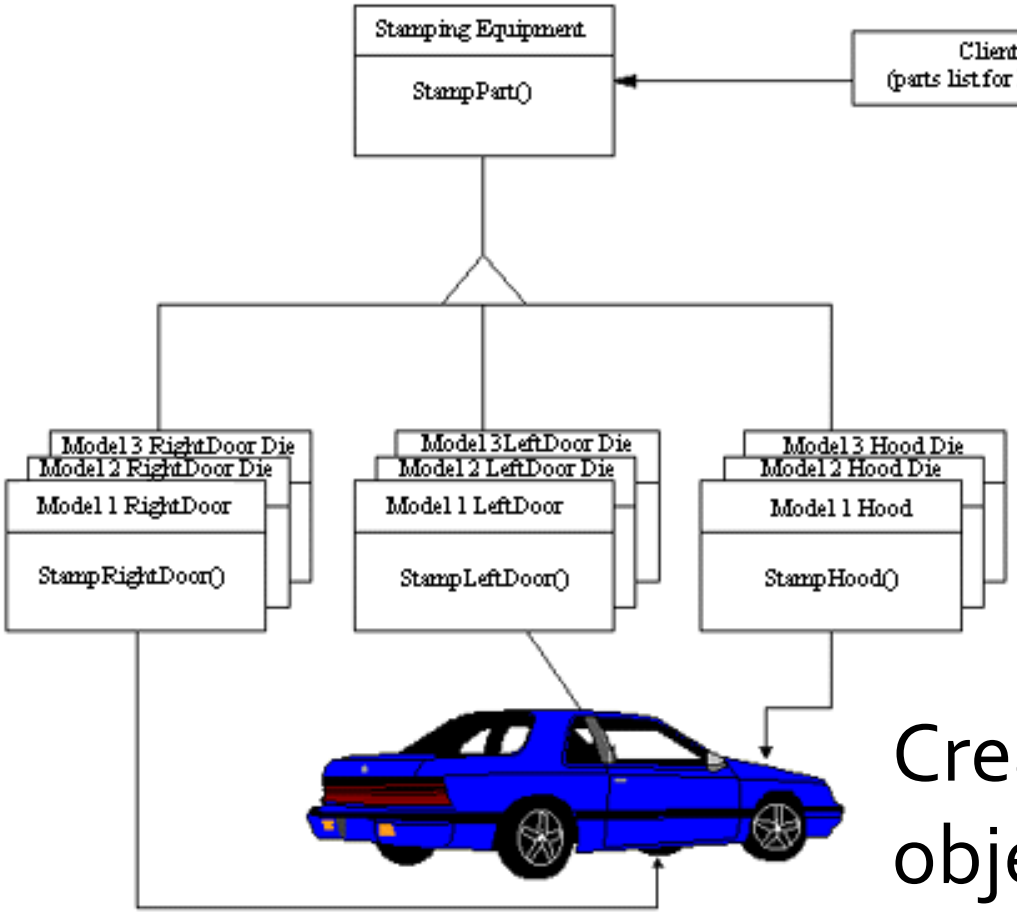




Wilhelmina Barns-Graham Trust

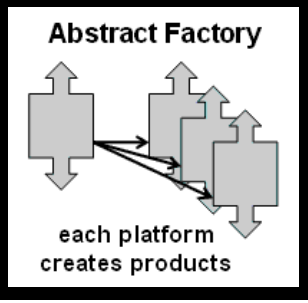
Abstract Factory





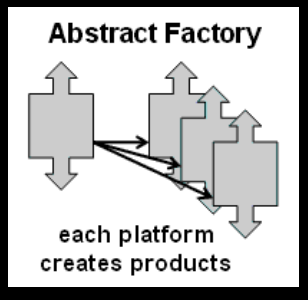
Creating a family of objects with a specific common feature.

Abstract Factory – Problem



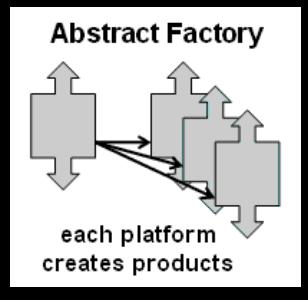
- We create an application that is to be portable.
- The need for encapsulation of multiplatforms, e.g.
  - operating system (e.g., GUI windows),
  - database.
- Existing #ifdef case expressions often can not be fully predicted and served.
- Our system should be independent of many products with which it works.

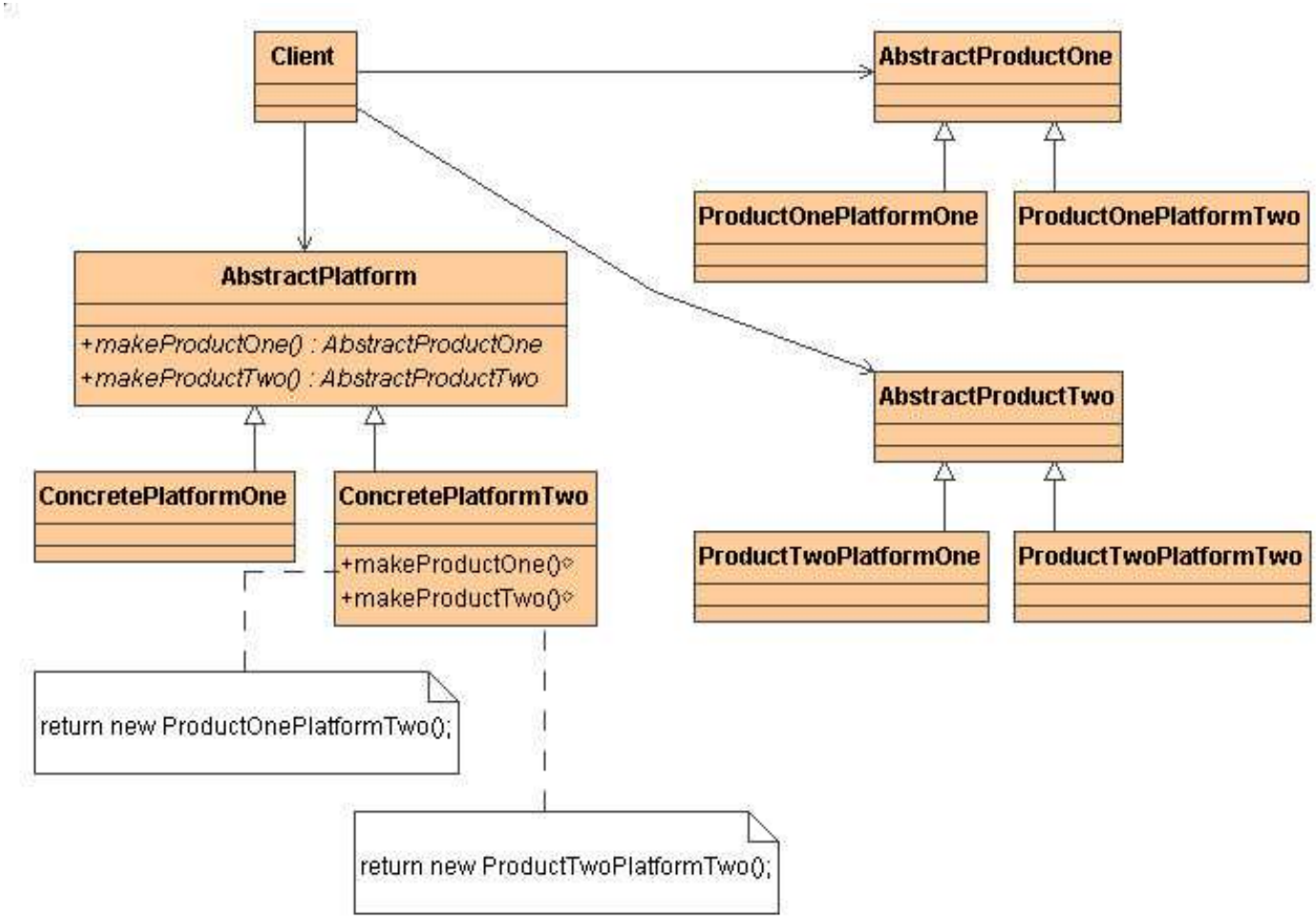
Abstract Factory – Solution



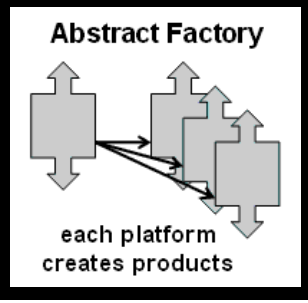
- Intermediary layer that provides the creation service.
- Support for the "family" of products.
- Support for many creation strategies:
  - selection of a derived class,
  - reusing objects in the cache,
  - distributed creation,
  - choice of platform or address space.
- Avoiding "new" construction (considering it to be harmful)

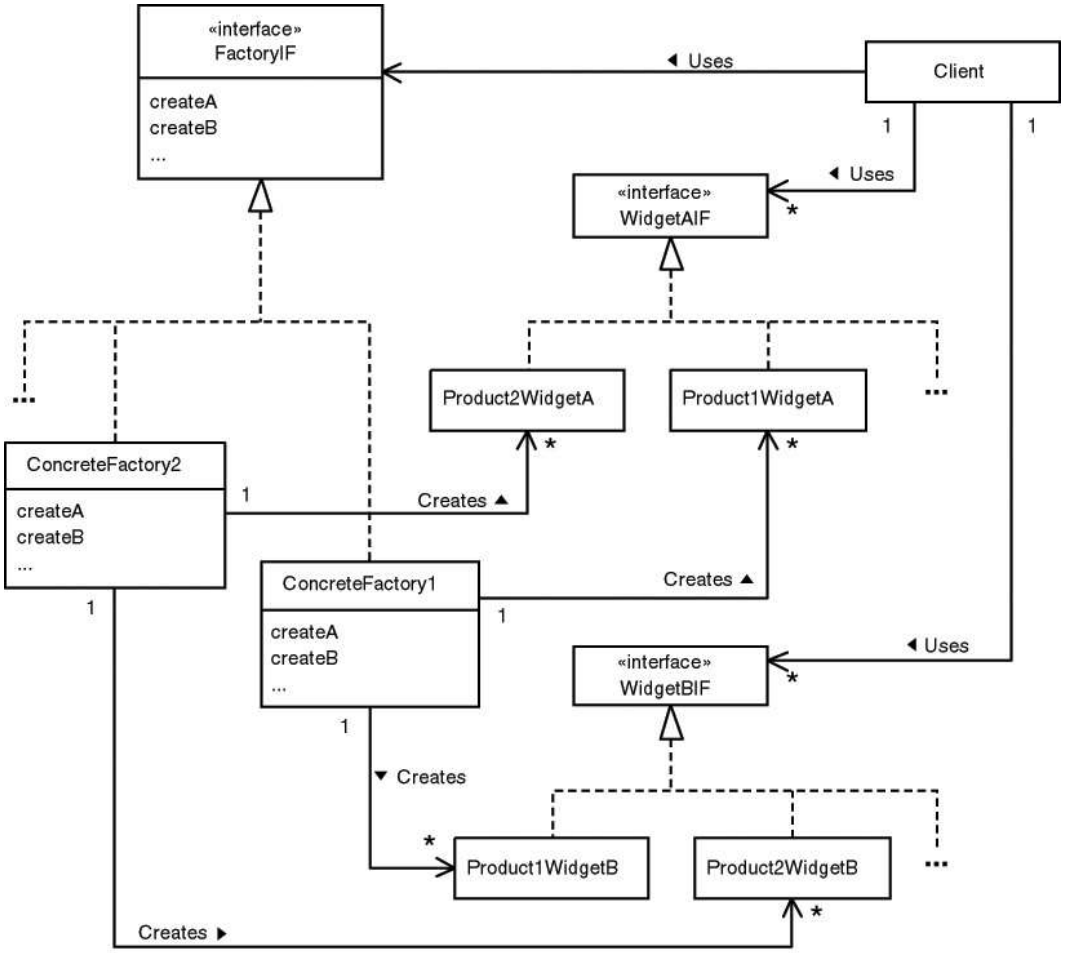
Abstract Factory – Class Diagram



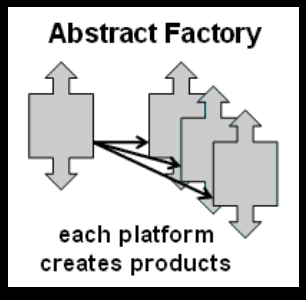


Abstract Factory – widgets example





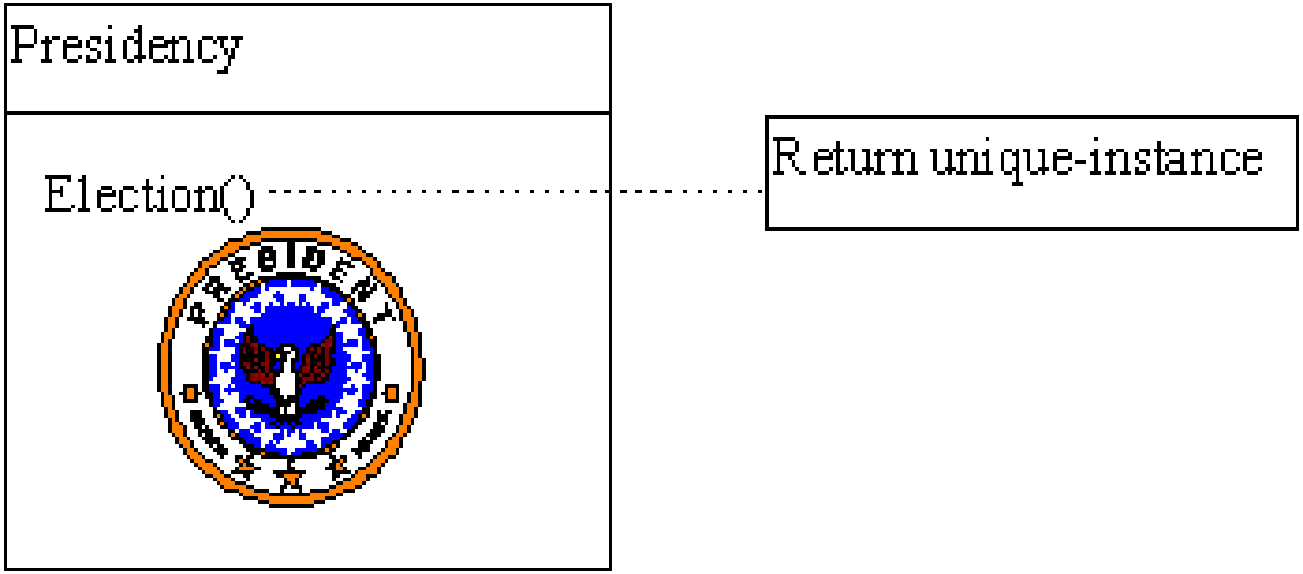
Abstract Factory – Consequences



- The client classes are independent of the specific product classes (widgets) that they use.
- Adding (as opposed to writing) the factory class so that the customer can work with additional products is simple.
- Forcing client classes to go through the factory interface to create specific product objects ensures that the client uses a consistent set of objects to work with the product.
- Write a new set of classes to interact with the product is laborious.

Singleton

- Provides a way to concentrate **all responsibility** in **one instance** of the class



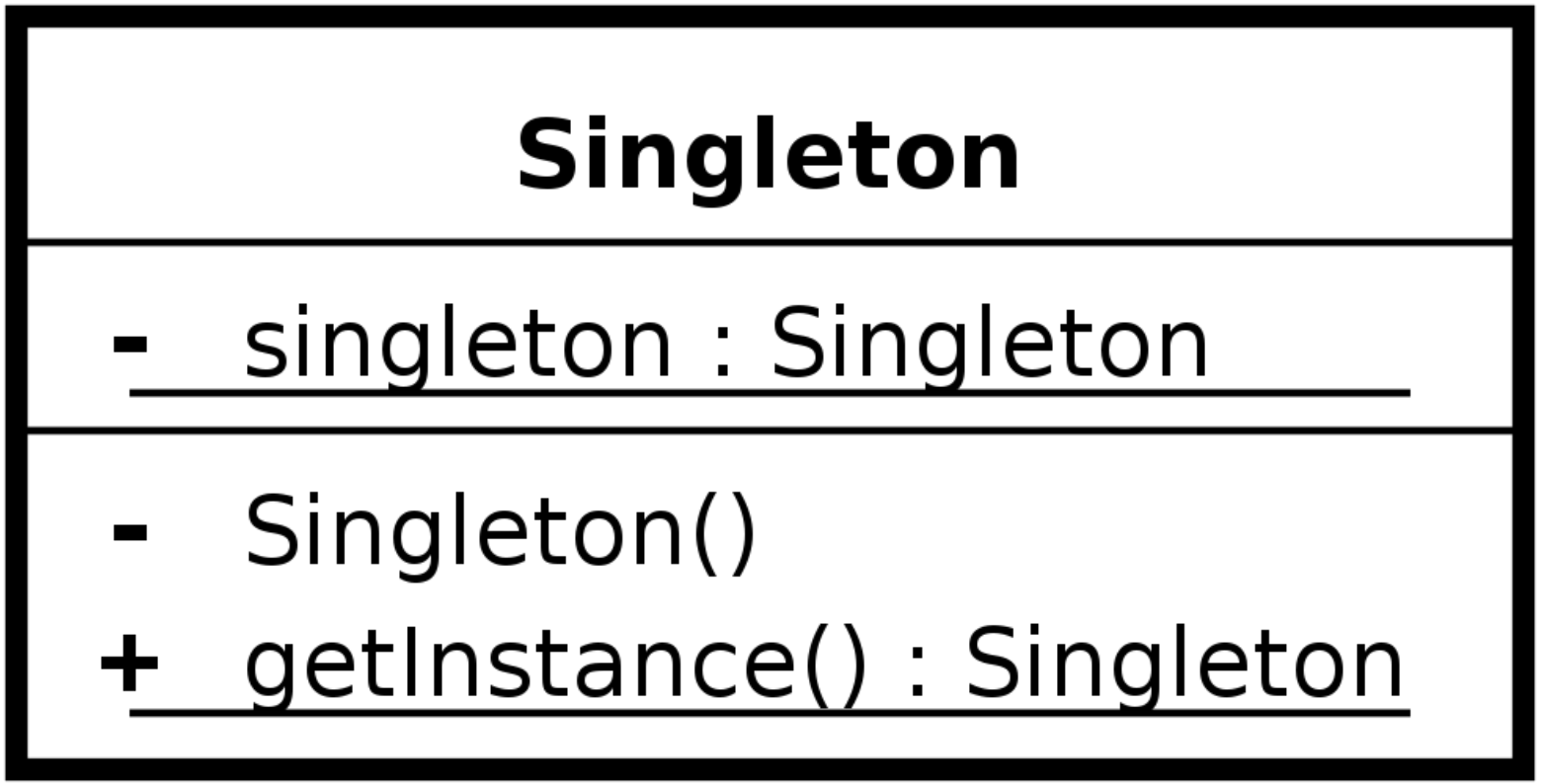
Singleton – Problem

- Application needs one, and only one, instance of an object.
- Additionally, this object is to be available globally, and its initialization is usually delayed until the first access attempt (***lazy initialization***).

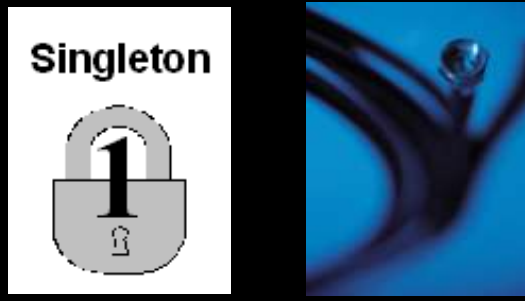
Singleton – Solution

- We enforce a certain number of instances of the class (**single instance** by using a *private constructur* and *static attribute*).
- Lazy initialization (initialization on first use).
- Global access (accessor function is a *public static method*).

Singleton – Class Diagram



Singleton – Consequences



- There is exactly one instance of the singleton class
- The *getInstance*() method of the singleton class encapsulates *the creation policy* for the singleton class so that the classes using it do not depend on the implementation details of the method.

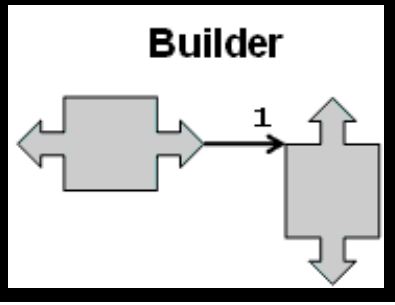
- Other classes that want to refer to the only singleton instance must use the static *getInstance*() method.

- Inheritance after singleton class is uncomfortable:
  - you need to create a non-private constructor,
  - you can not override the static *getInstance*().

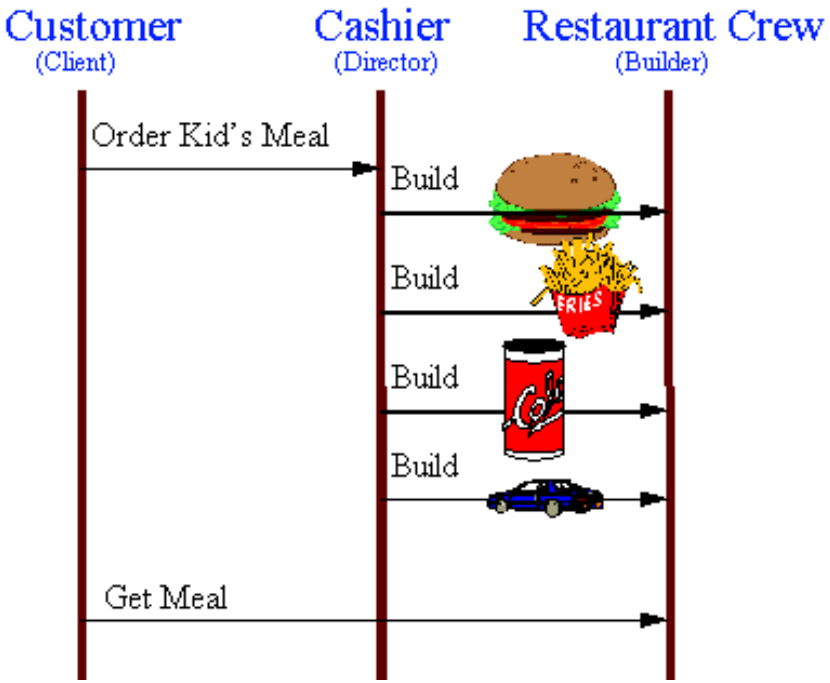
Singleton and other patterns

- Other creational patterns (**Abstract Factory**, **Builder**, and **Prototype**) can use **Singleton** in their implementation.
- **Facade** objects are often **Singletons** because only one **Facade** object is required.
- Objects representing states in a **State** pattern are often **Singletons**.

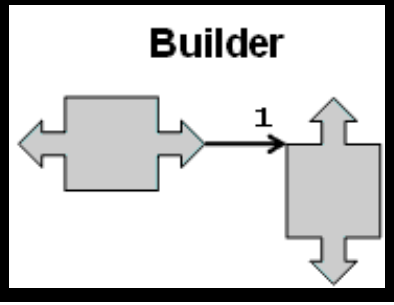
Builder



- Gradual gathering information about the object before its construction.

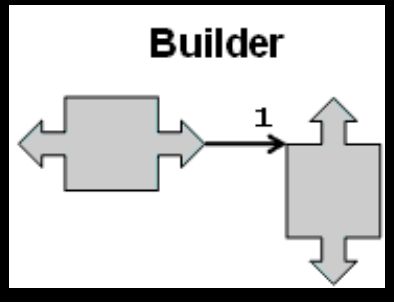


Builder – Problem



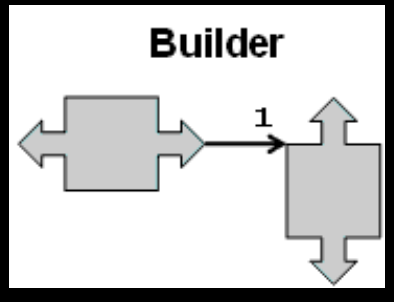
- Program should enable the creation of many types of external representations of the same data.
- A content is on the secondary storage and a representation is to be built into a main memory
- Classes responsible for content delivery should be independent of any external data representations and classes that create them.
- Classes responsible for the construction of external data representations are to be independent of the classes providing content.

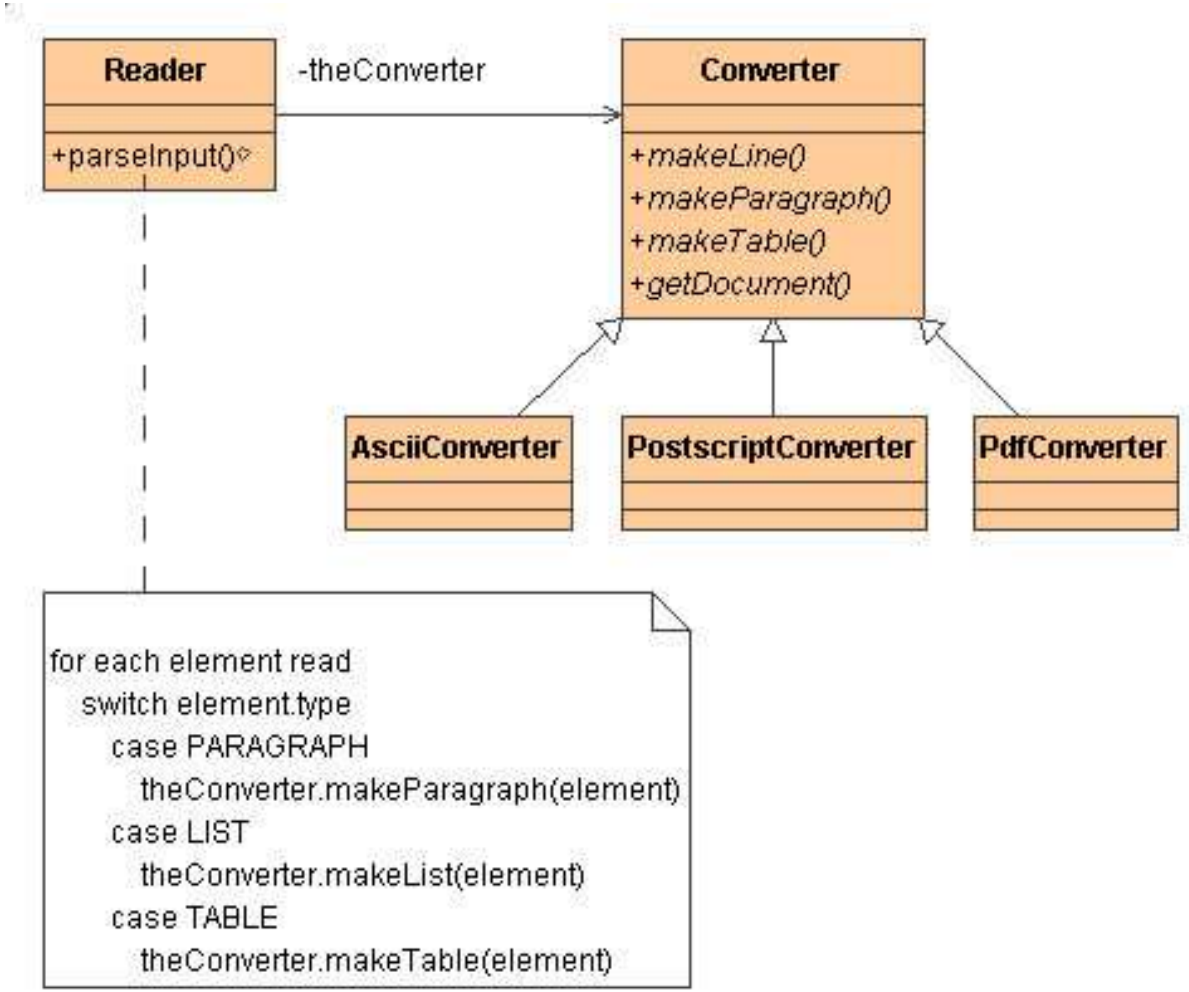
Builder – Solution



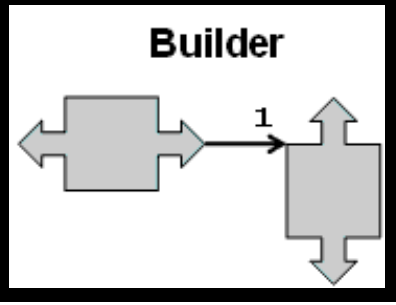
- By using one "input" we enable to obtain many results ("outputs").
- Structure: Wrapper / Delegation:
  - the wrapper directs the creation / composition algorithm,
  - each delegation encapsulates the target output.

Builder – Class Diagram





Builder – Consequences



- Determination of content and design of a specific data representation are independent.
- An external representation of the product may change without affecting content delivery facilities.
- Builder objects can work with various content delivery objects without the need for any changes.

Creational Patterns Properties

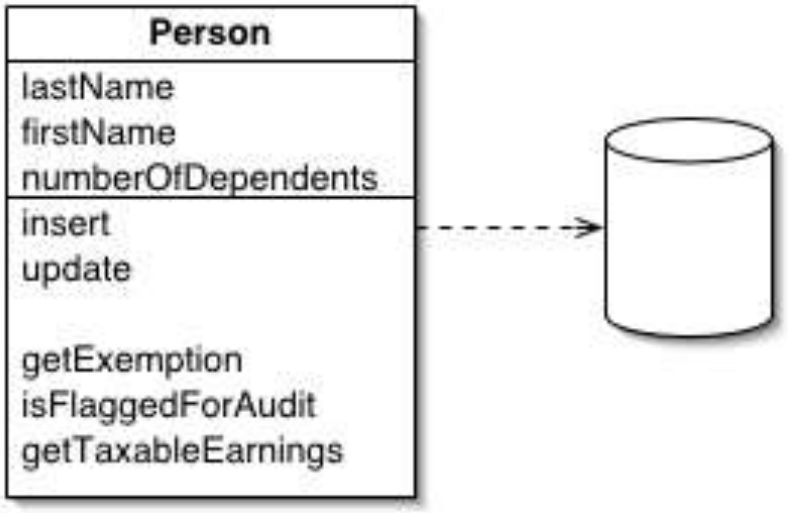
- Often the design evolves from **factory method** to another more flexible creational pattern.
- **Builder** provides better control over the structure than other patterns (e.g. **factory method**) giving an object managing a step by step creation of the object (the remaining patterns form the object in one step).
- **Builder** often creates a **composite**.
- **Builder** refers to creating as a **strategy** to algorithms.

SOLID and Design Patterns

Active Record example study

Active Record pattern

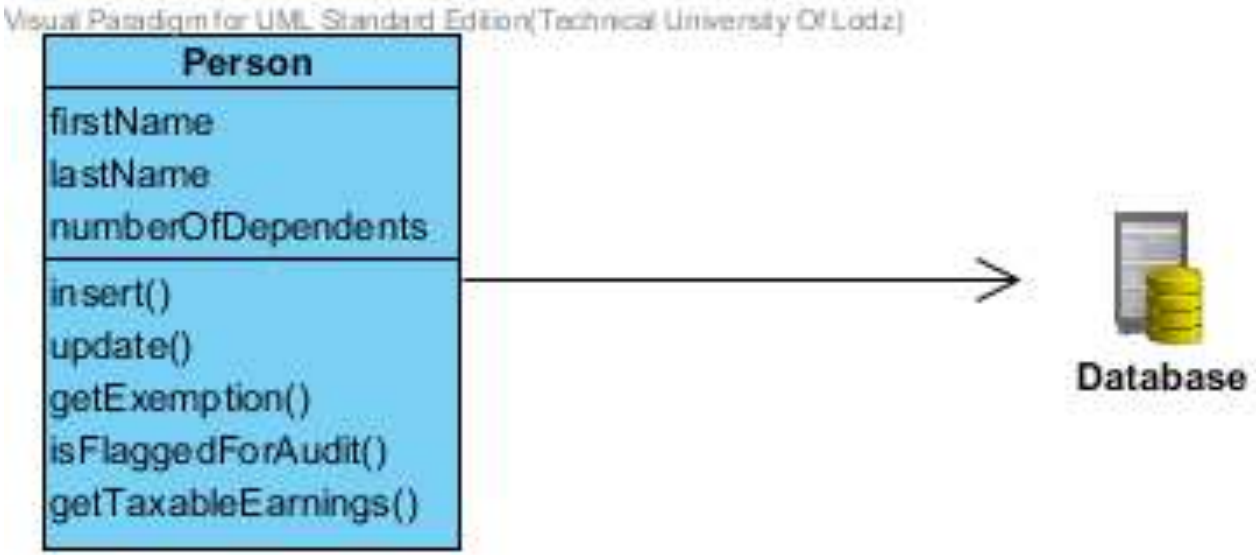
An object that wraps a row of a table or a database view, encapsulates access operations - and implements elements of the realm of domain logic.



Signle Responsibility Principle

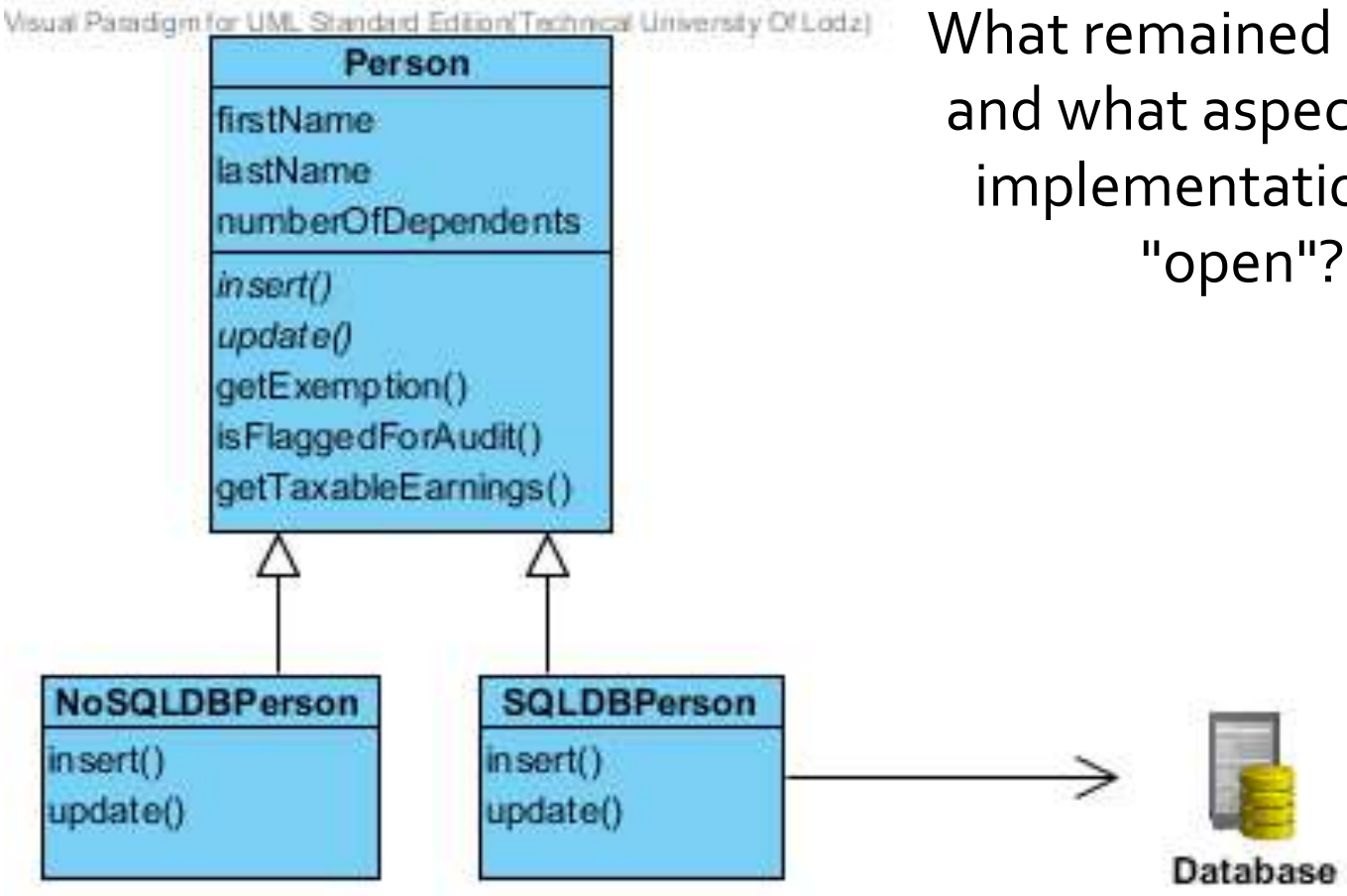
ActiveRecord and SRP

How much and what are the responsibilities of the Person?
Does this pattern meet the SRP principle?



Open Closed Principle

OCP through inheritance

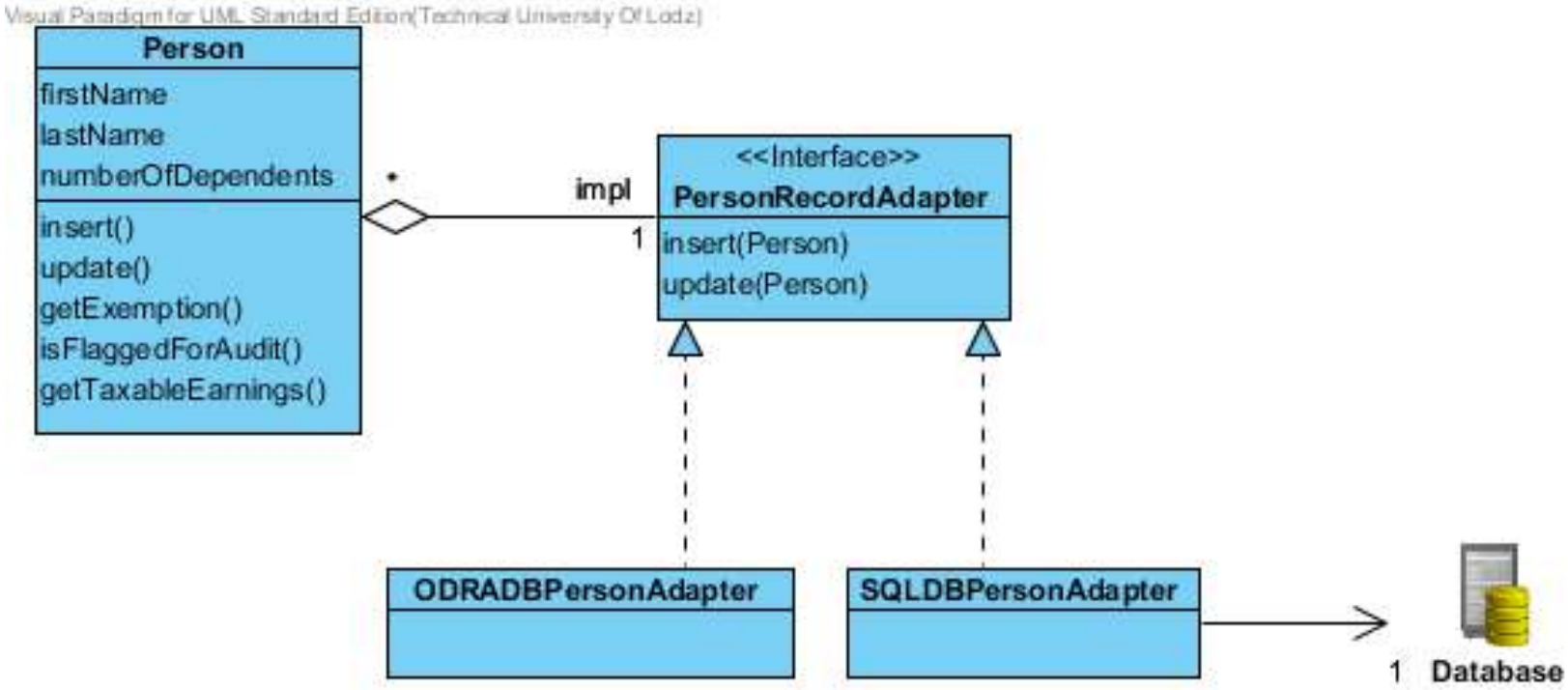


What remained "closed" and what aspect of the implementation was "open"?

Liskov Substitution Principle

LSP – Strategy pattern

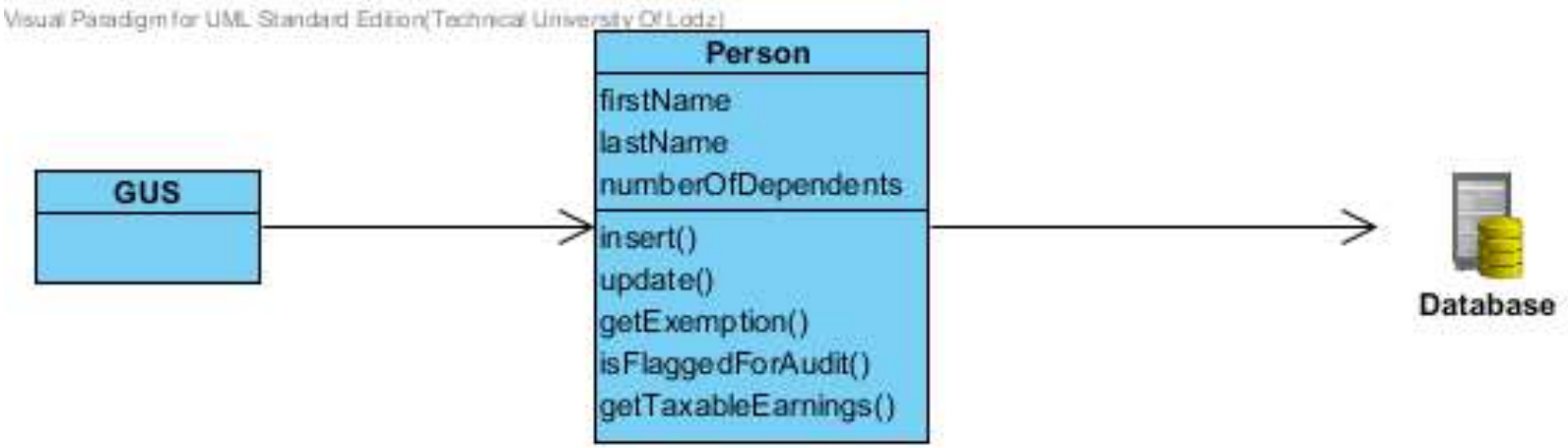
The PersonRecordAdapter interface specifies the "contract" or responsibility of the classes implementing it (and their subclasses).



Interface Segregation Principle

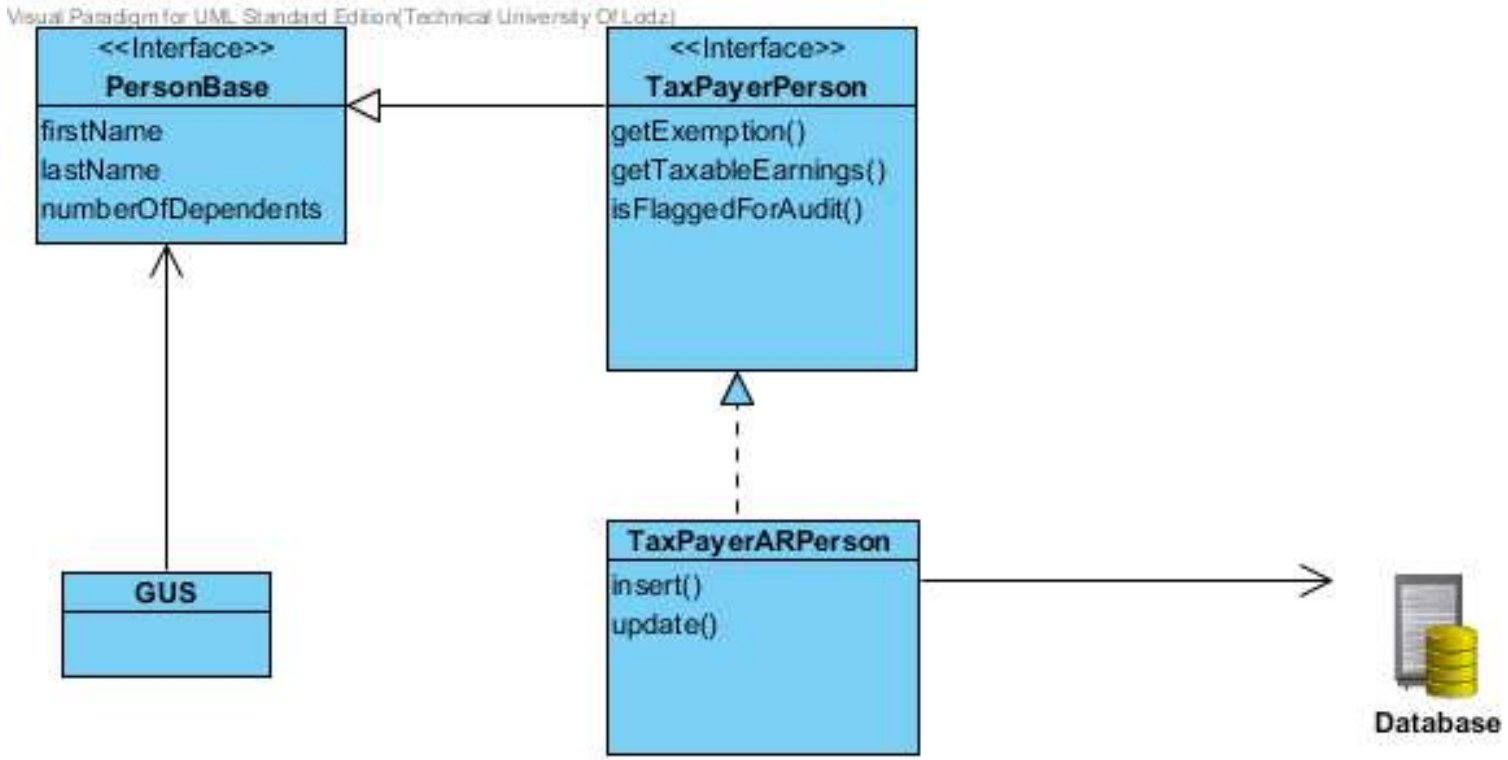
ISP and Active Record pattern

The Person class interface combines several responsibilities (data, tax logic, operations on a database) by charging potential clients with each.

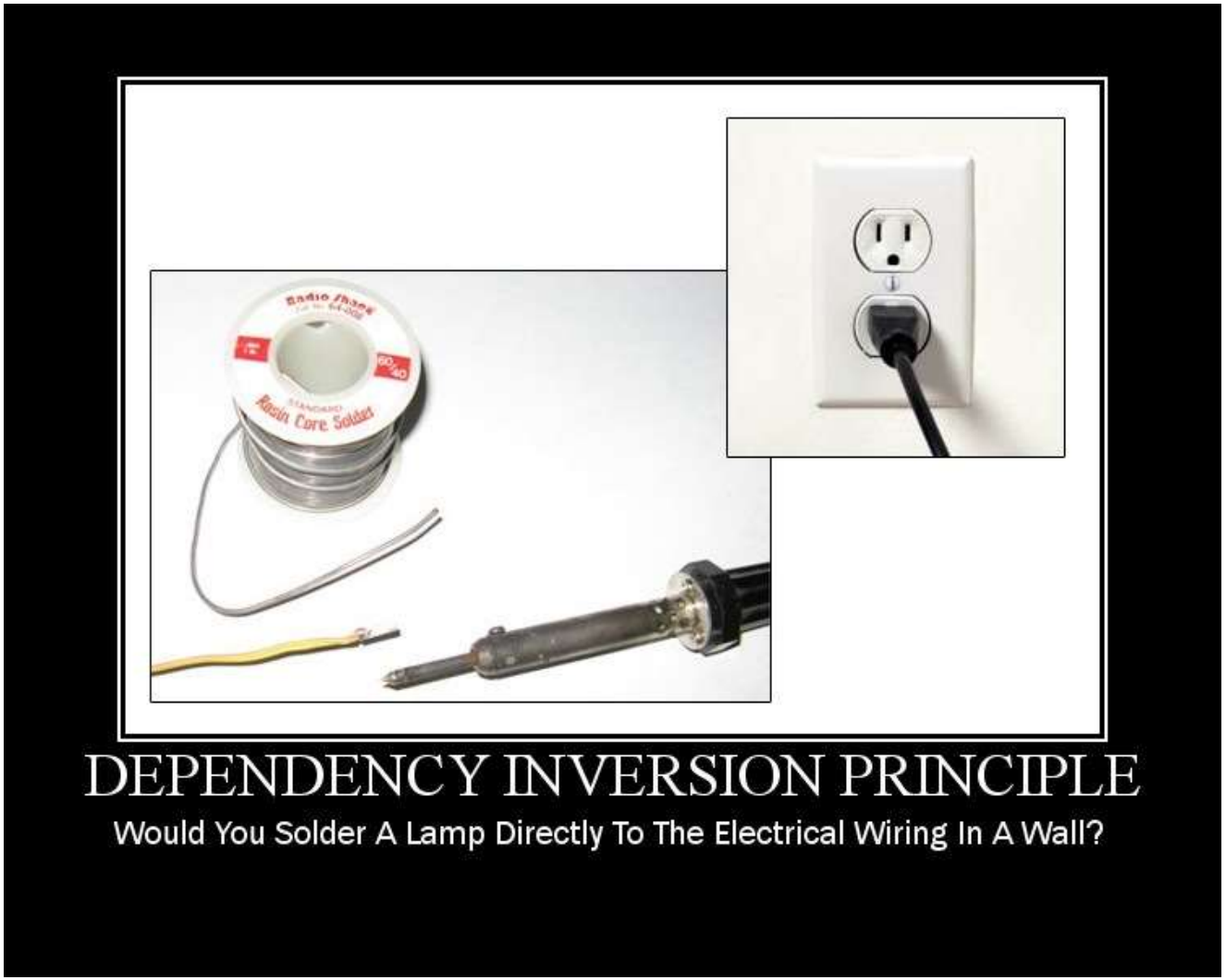


ISP interface ordering

The GUS/CSO depends only on changes in the PersonBase interface.

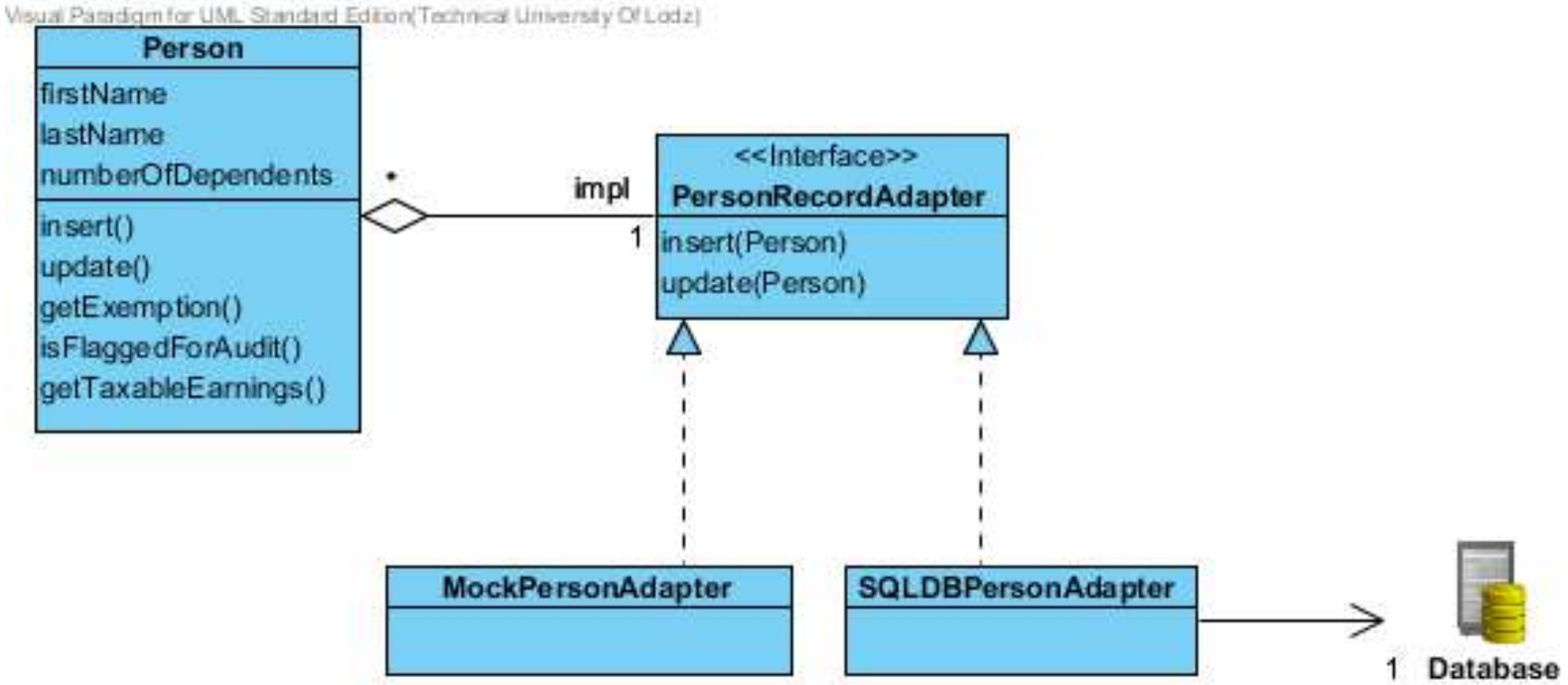


Dependency Inversion Principle



DI – component testing

How to test the operation of *insert* and *update* methods and methods dependent on them in the absence of a database?



References

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley; 1995
- Steven John Metsker: *Design Patterns in C#*, Addison-Wesley Professional; 2004
- Robert C. Martin: *Clean Code: A Handbook of Agile Software Craftsmanship*; 2008
- Martin Fowler, Kent Beck: *Refactoring: Improving the Design of Existing Code,* Addison-Wesley Professional; 1999
- Craig Larman: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development;* Prentice Hall, 2004

On-line resuorces

- http://www.vincehuston.org/dp/
- http://hillside.net/patterns/patterns-catalog
- http://en.wikipedia.org/wiki/Design_pattern_(computer_science)
- *plenty more…*