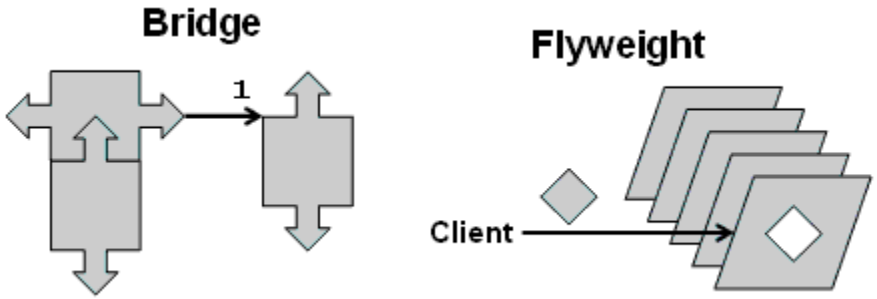
Wzorce Projektowe

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

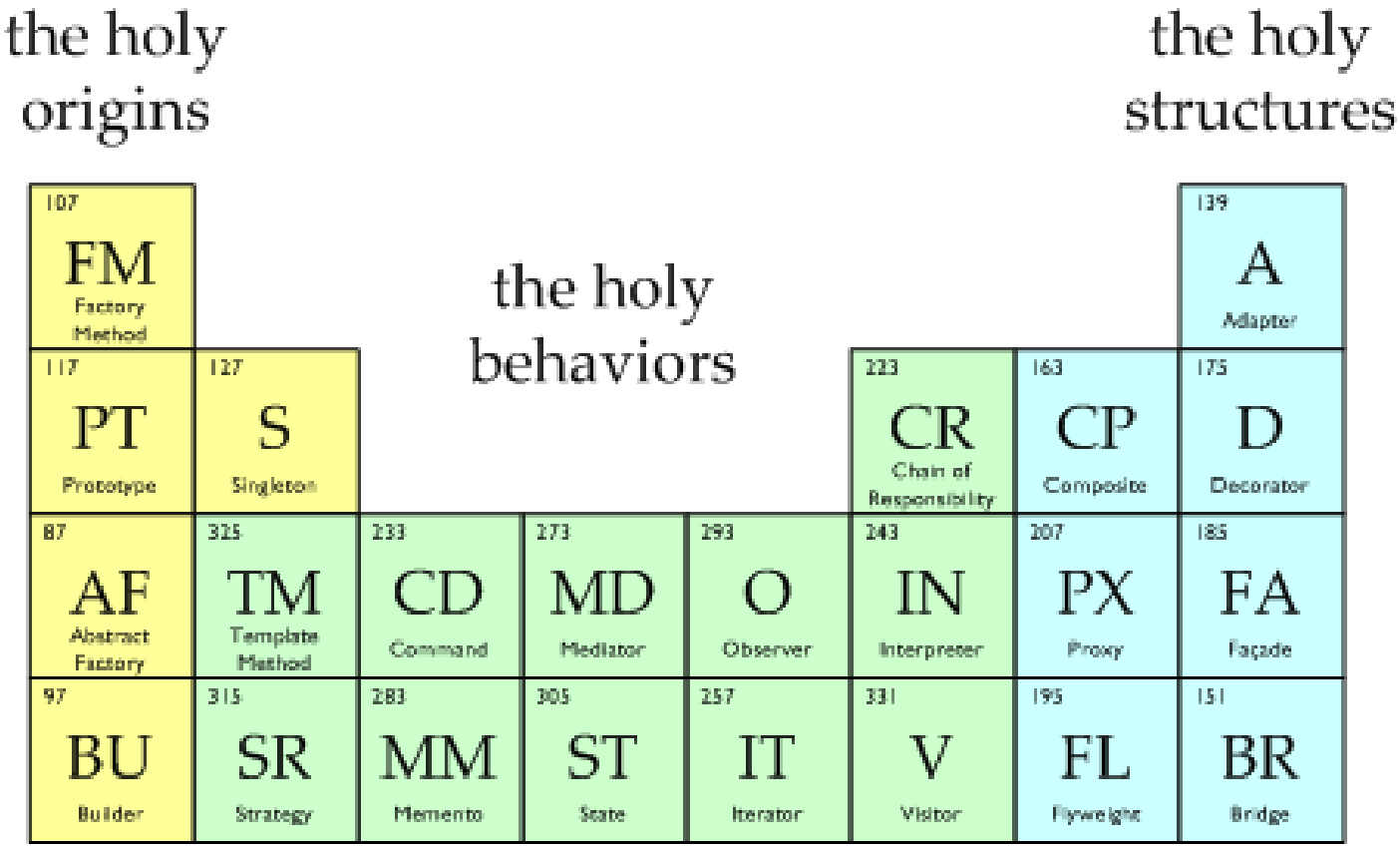
Design Patterns – the beginning

- 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

Other patterns

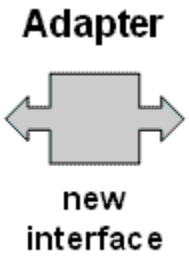
- Patterns start but do not end with GoF
- All patterns are based on certain foundations of objectivity
  - Inheritance and polymorphism
  - Interfaces
  - Delegation
- There are also other types of patterns:
  - Concurrency, (e.g., Active Object, Thread Specific Storage, Thread Pool Pattern, Monitor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

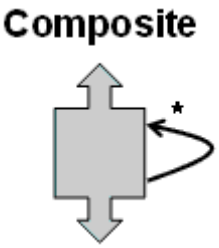
Structural patterns

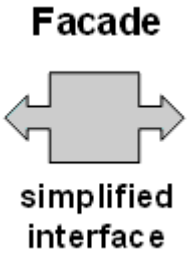
Gang of Four

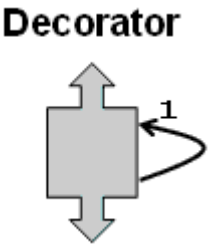
Roadmap

- Adapter
- Composite
- Facade
- Decorator







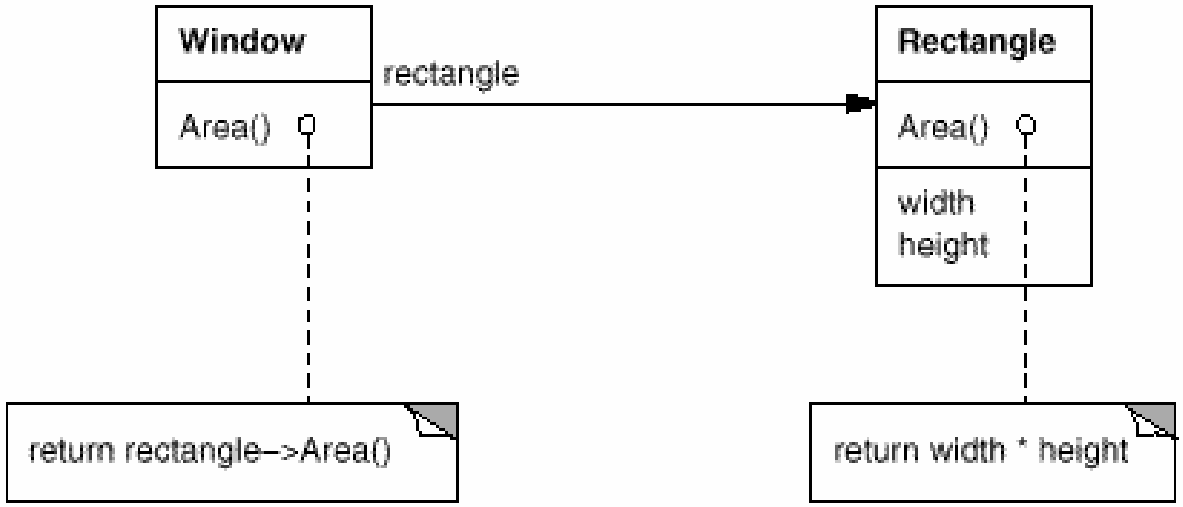


Basic concepts

- 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!)

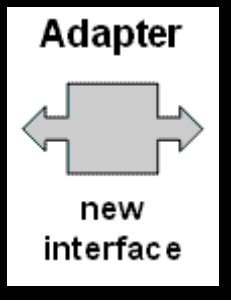


Structural patterns

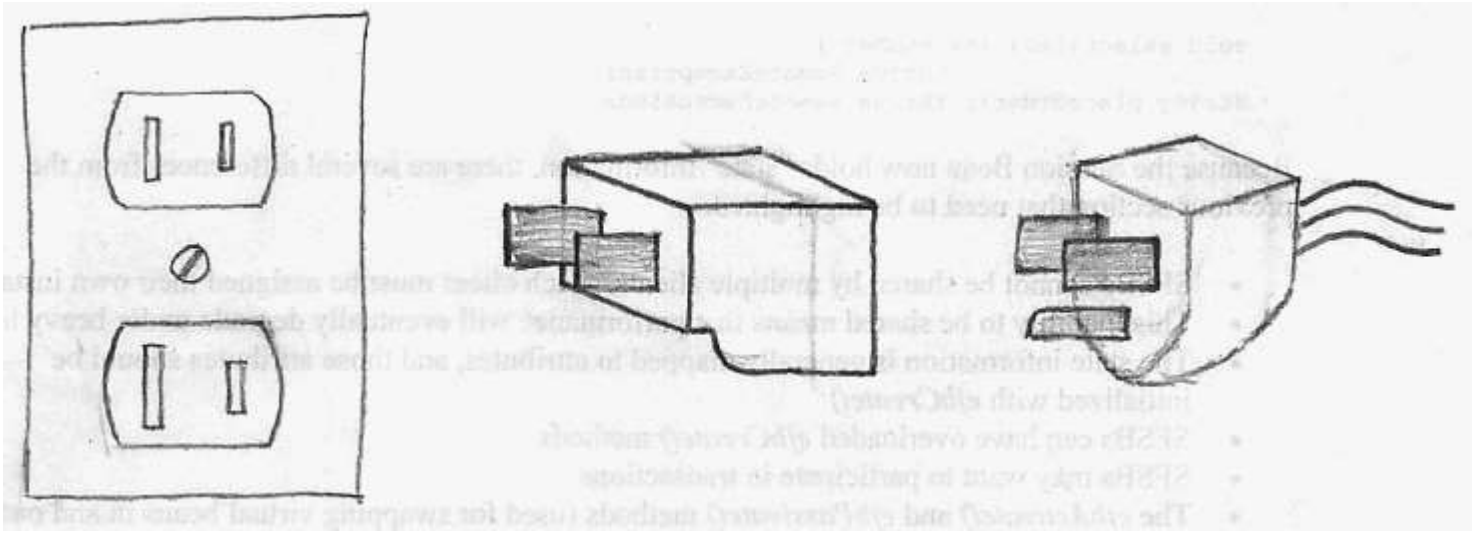
- **Structural patterns** concern common ways of organizing objects of different types so that they can cooperate with each other.

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

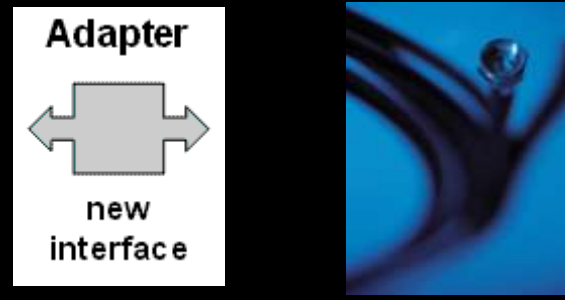
Adapter



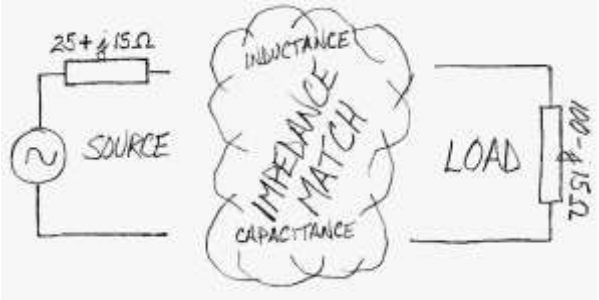
- Convert the interface of a class into another interface clients expect.



Adapter - Problem



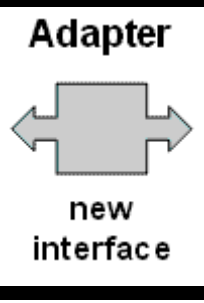
- An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.
  - Incompatible interface
  - Impedance mismatch

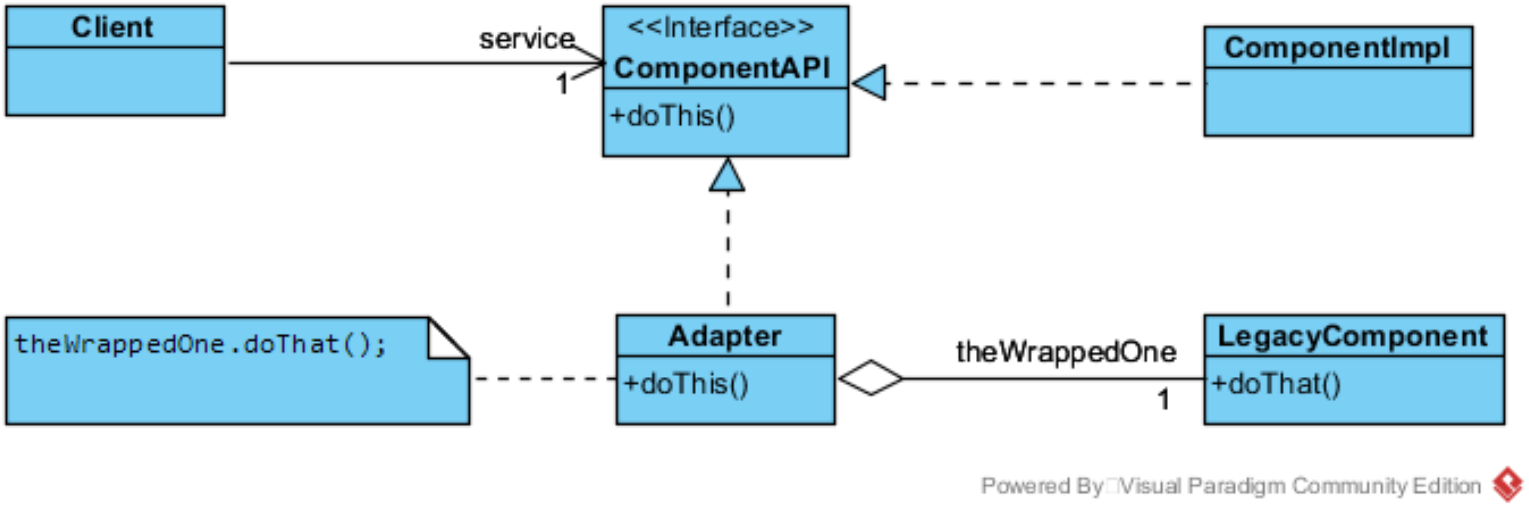


Adapter - Solution

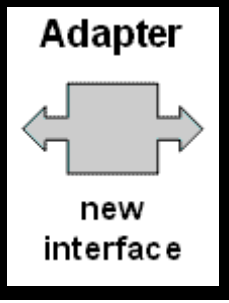
- We wrap the existing code with new interfaces.
- We adjust the impedance of old components to the new system
  - often an apparent solution! –
    i.e., like sewing on an old patch to a new trousers
- Structure: Wrapper / Delegation.

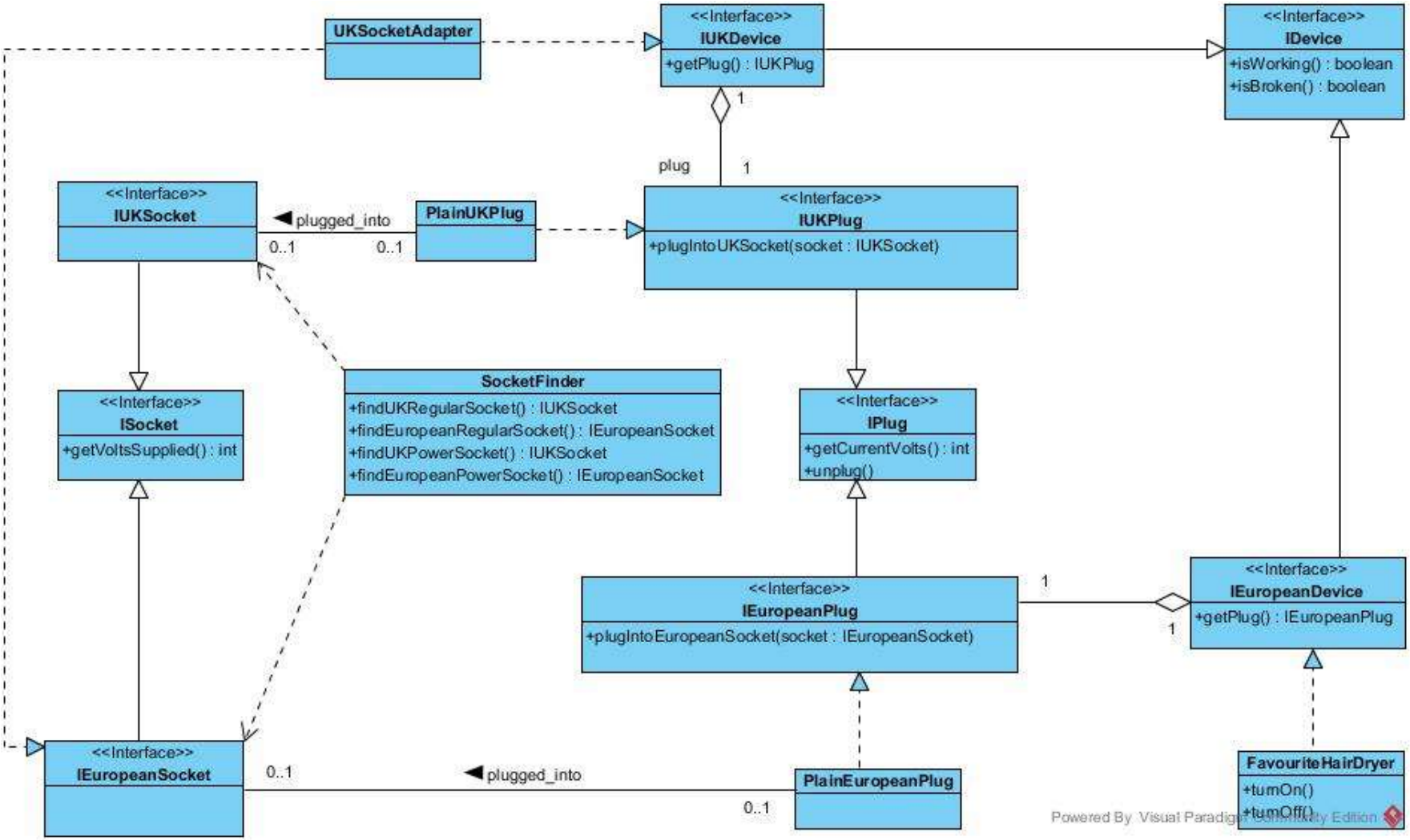
Adapter – Class diagram





Adapter – Example



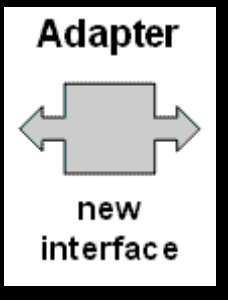


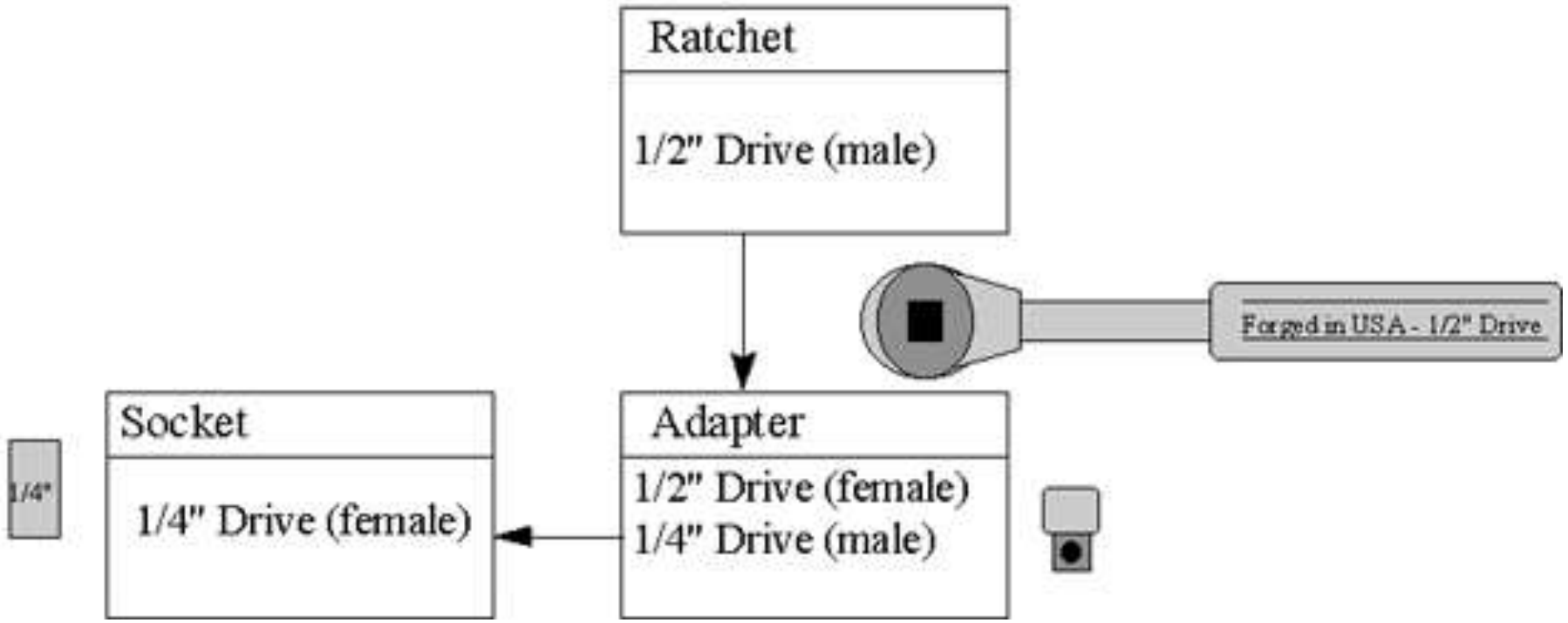
Adapter - Consequences

- 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)

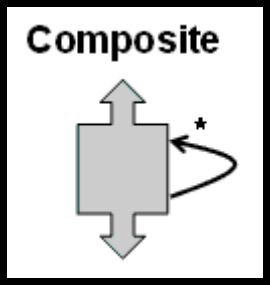
- The adapter adds an intermediate layer in the program:
  - negative impact on performance (for low-level components),
  - difficulty understanding the application (on implementation level).

Adapter – non-software example



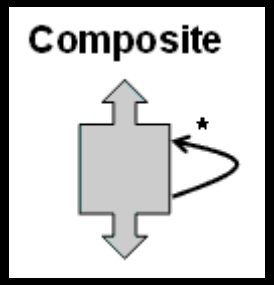


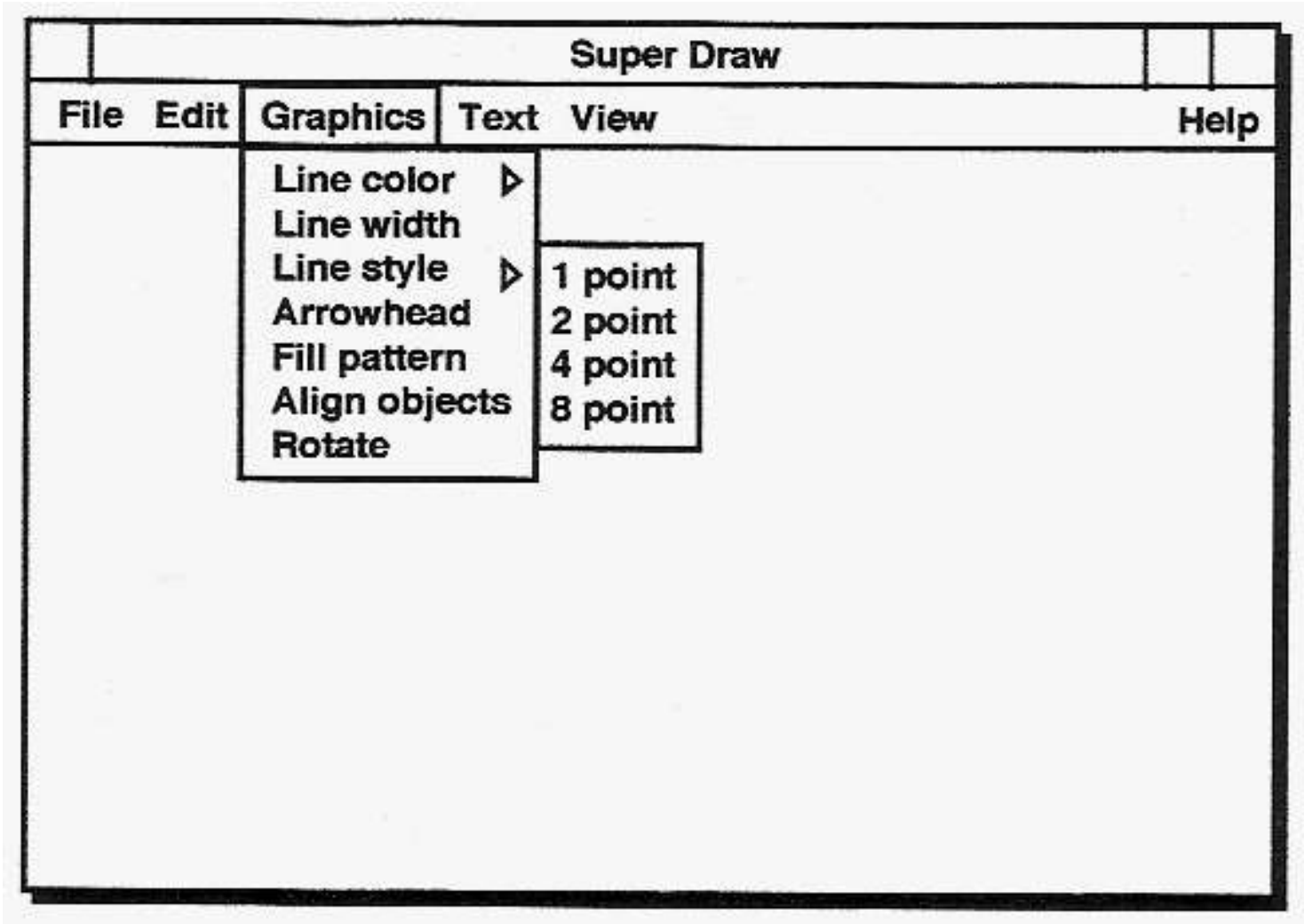
Composite



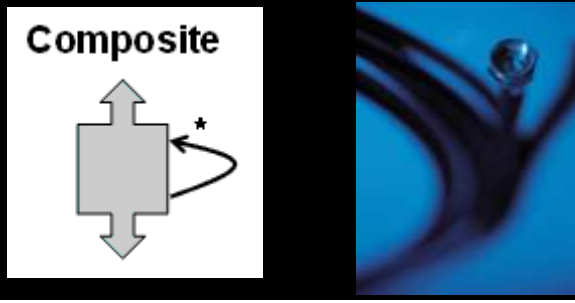
- Compose objects into tree structures to represent whole-part hierarchies.
  - Defining an interface that considers both individual objects and groups of objects.
- Composite lets clients treat individual objects and compositions of objects uniformly.

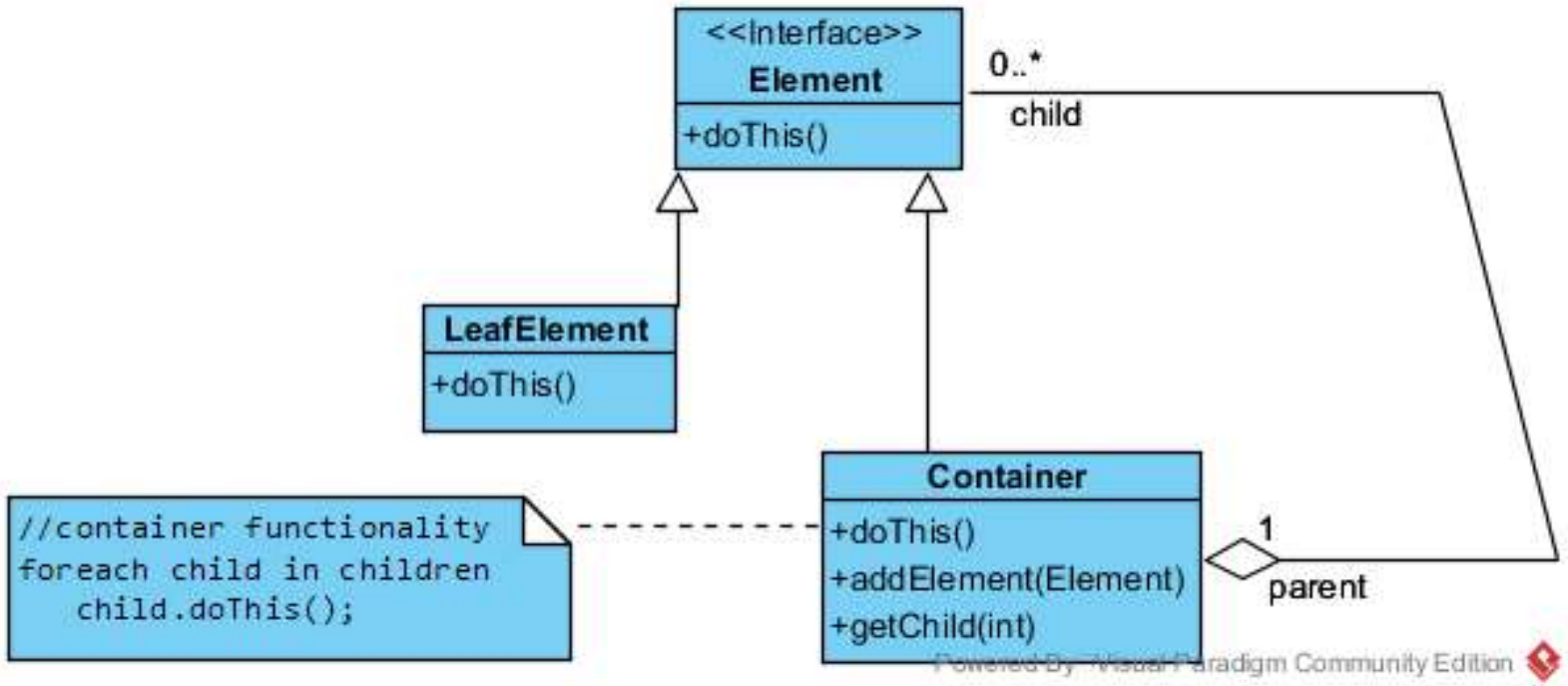
Composite – Example



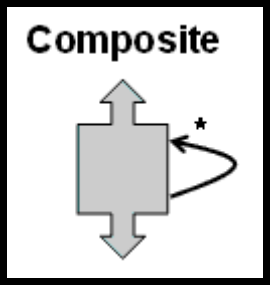


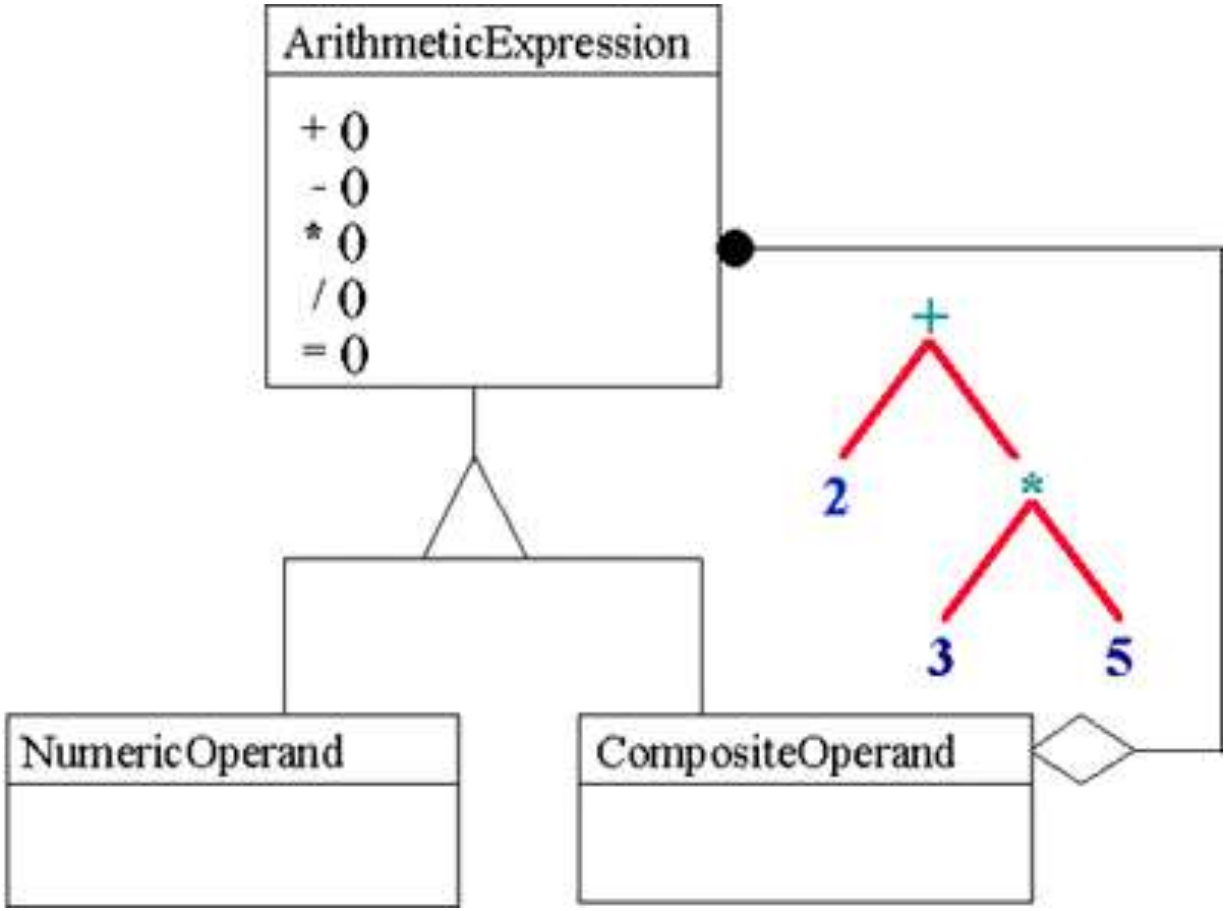
Composite – Class diagram



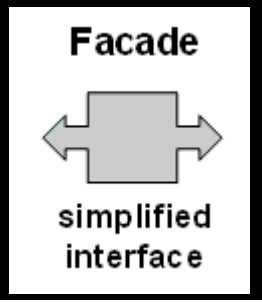


Composite – non-software example

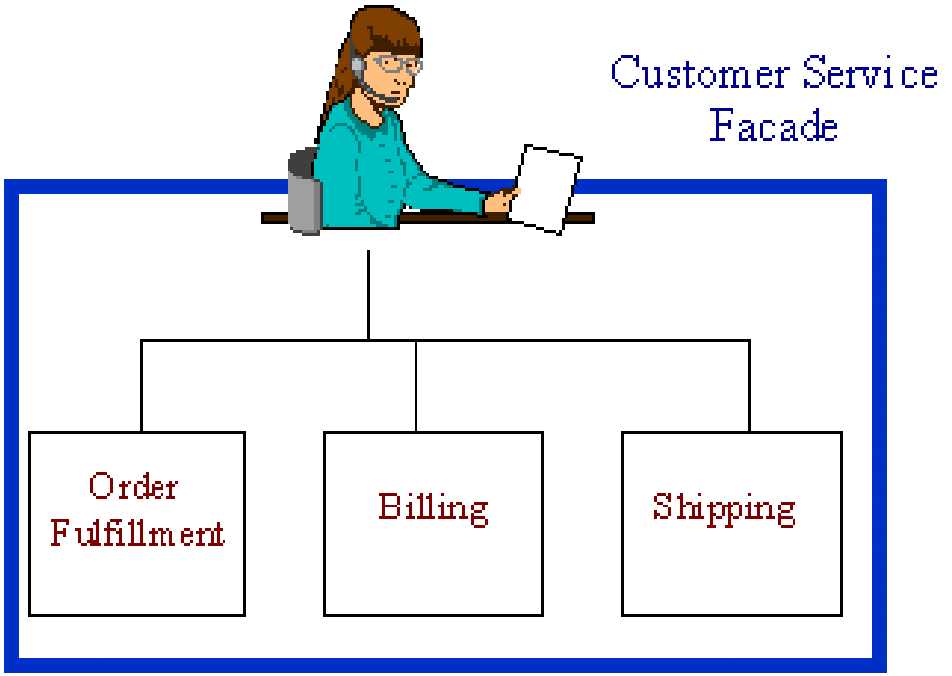




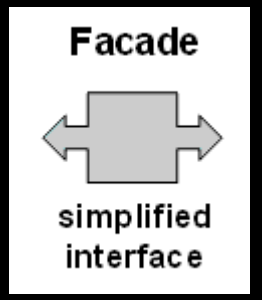
Facade



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

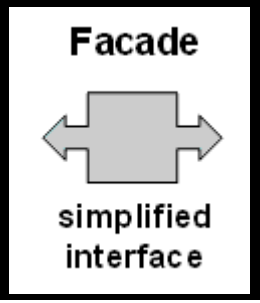


Facade - Problem



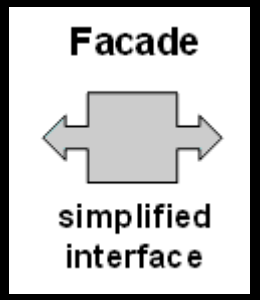
- There are many dependencies between classes implementing abstractions and client classes, noticeably increasing its complexity.
- The need to simplify the client (e.g., reducing the risk of errors).
- Increase the degree of reuse of the system or library.
- Designing classes to work in clearly separated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

Facade - Solution

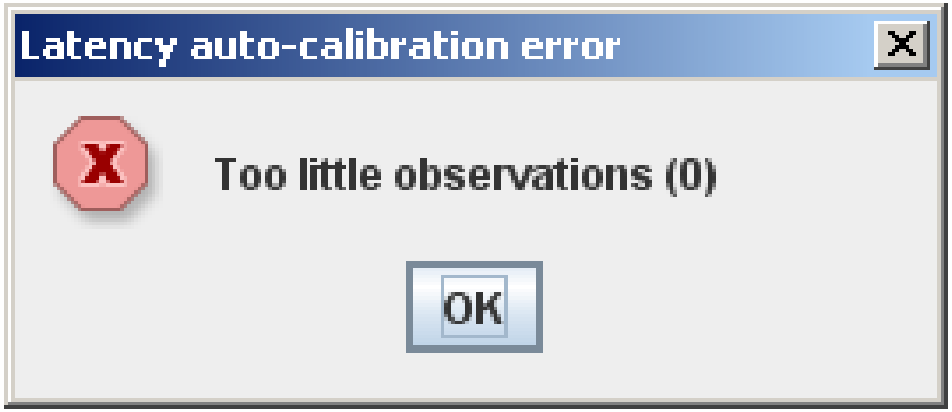


- Wrapping the existing system with a **new** interface.
- A simple entry point for a large subsystem.
- Adding an intermediate layer that hides the complexity of the legacy system

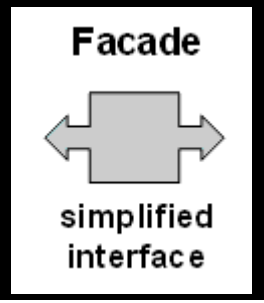
Facade – example



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

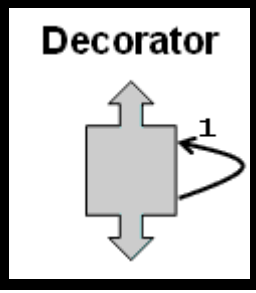


Facade - Consequences



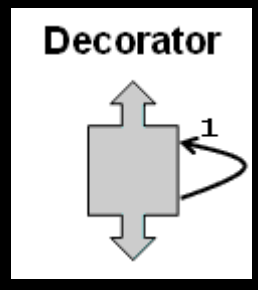
- Inserting the facade classes simplifies the client by shifting client dependencies to the facade.
- The client does not need to know the classes behind the facade.
- Changing the implementation (e.g., fixing) of the classes behind the facade, i.e. those that implement abstractions, is possible without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

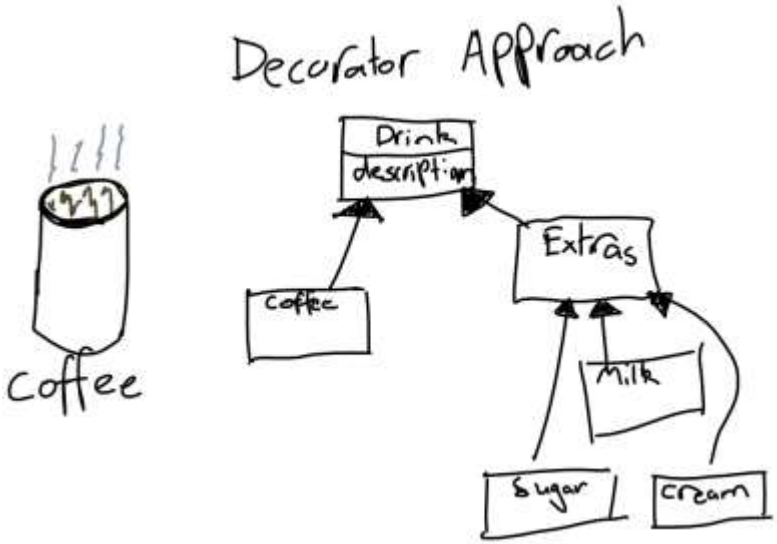
Decorator

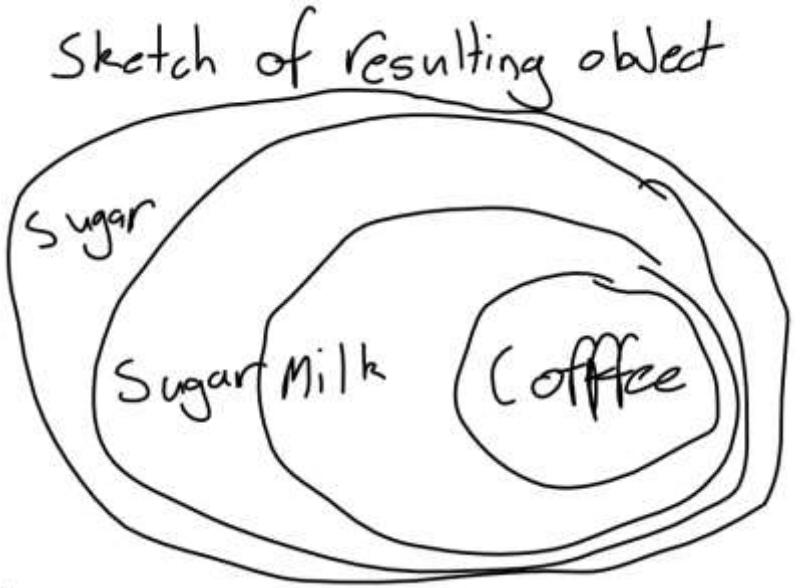


- Attach additional responsibilities to an object dynamically.
- Provide a flexible alternative to subclassing for extending functionality.

Decorator – coffee example

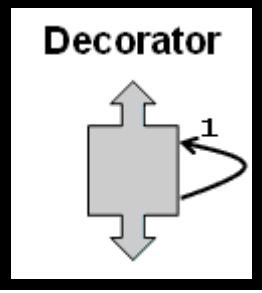


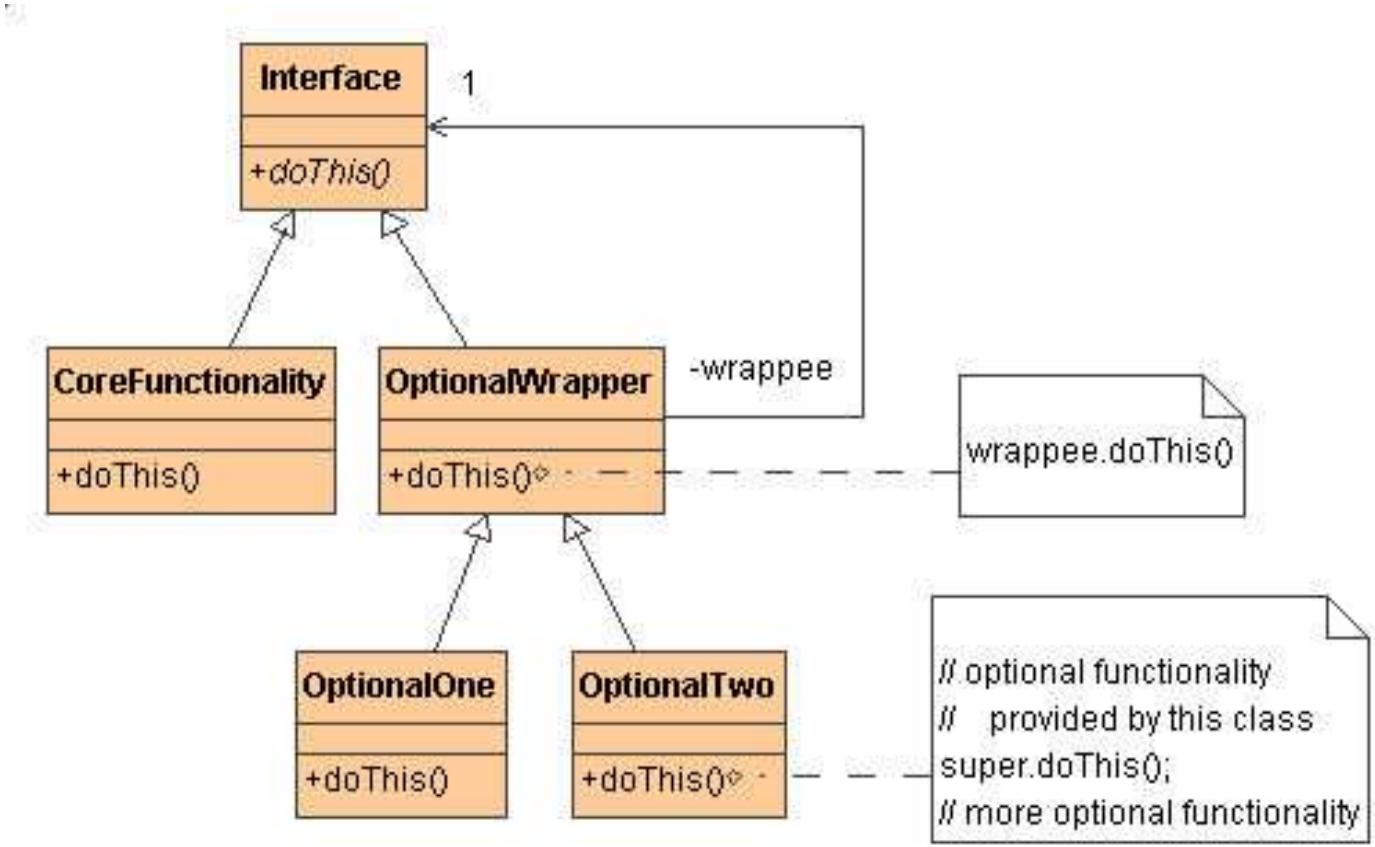




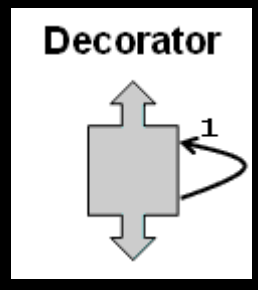
https://tylercash.xyz/post/a-look-at-decorator-patterns

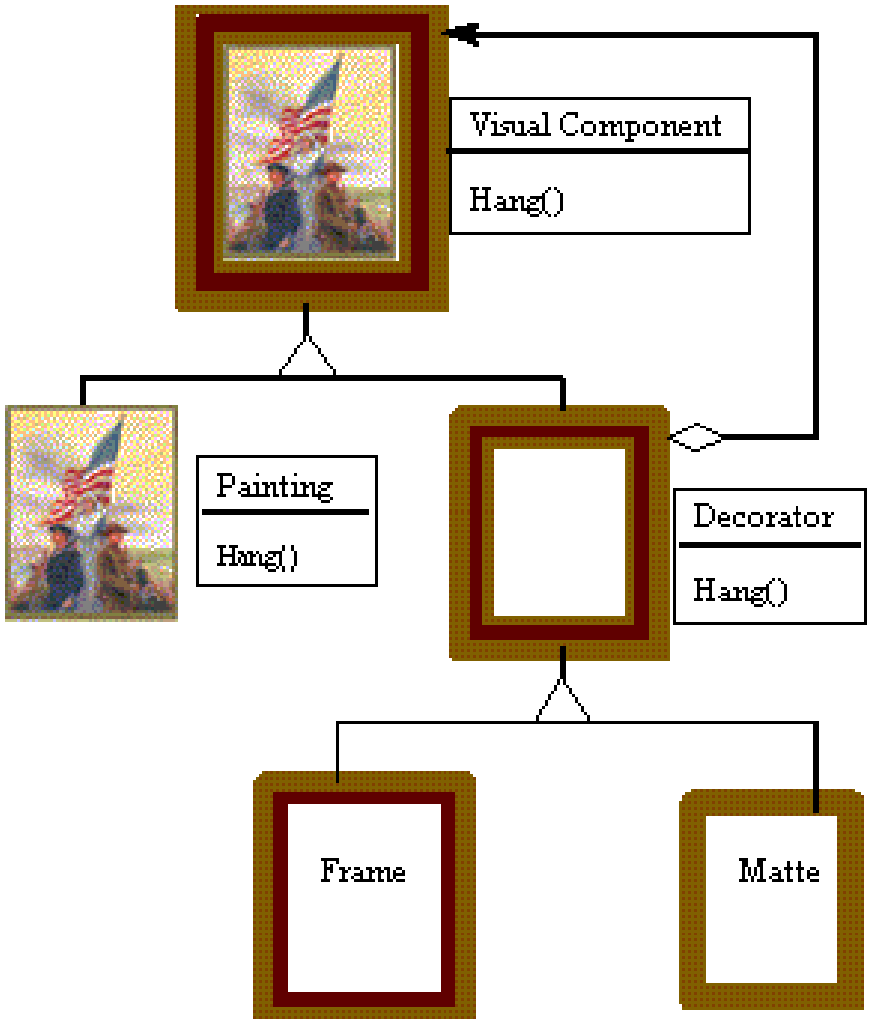
Decorator – Class diagram





Decorator – non-software example



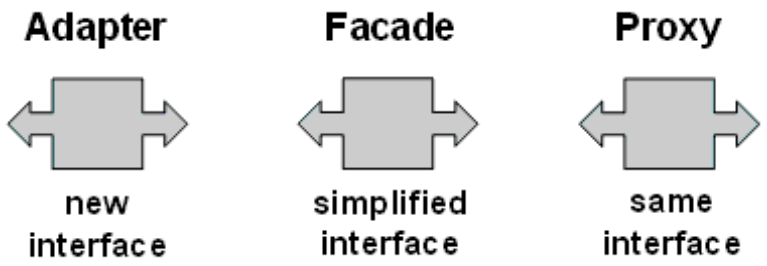


Composite and Decorator

- **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***

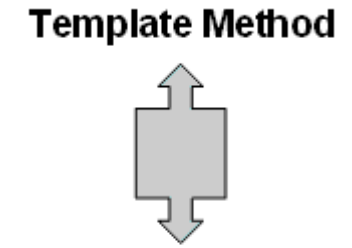


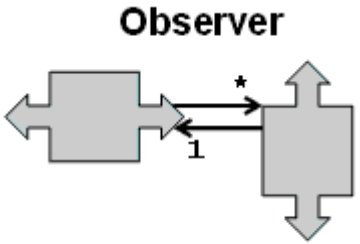
Behavioral patterns

Gang of Four

Roadmap

- Strategy
- Template Method
- Observer
- Iterator



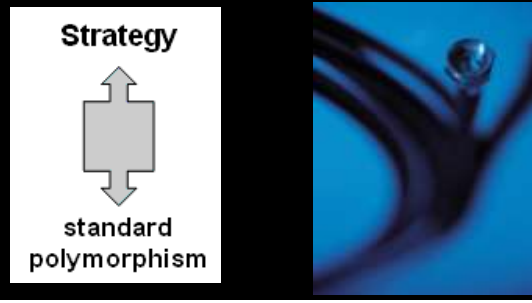


Behavioral patterns

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

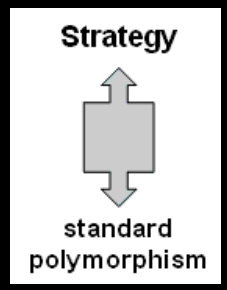
- Organization, management, combining of behavior

Strategy



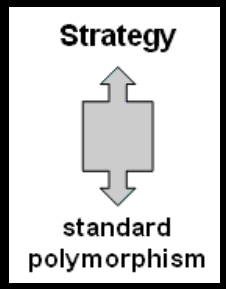
- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from the clients that use it.

Strategy – Problem



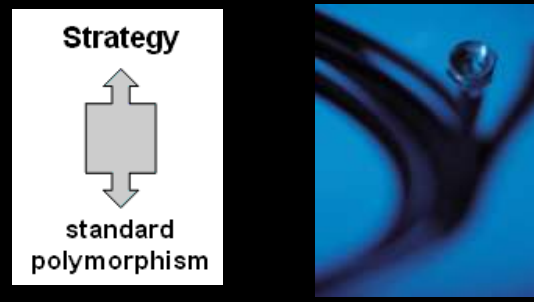
- The complexity of the code resulting from the existence of many strategies for a specific problem.
- The need to build object-oriented software with a minimized number of dependencies.

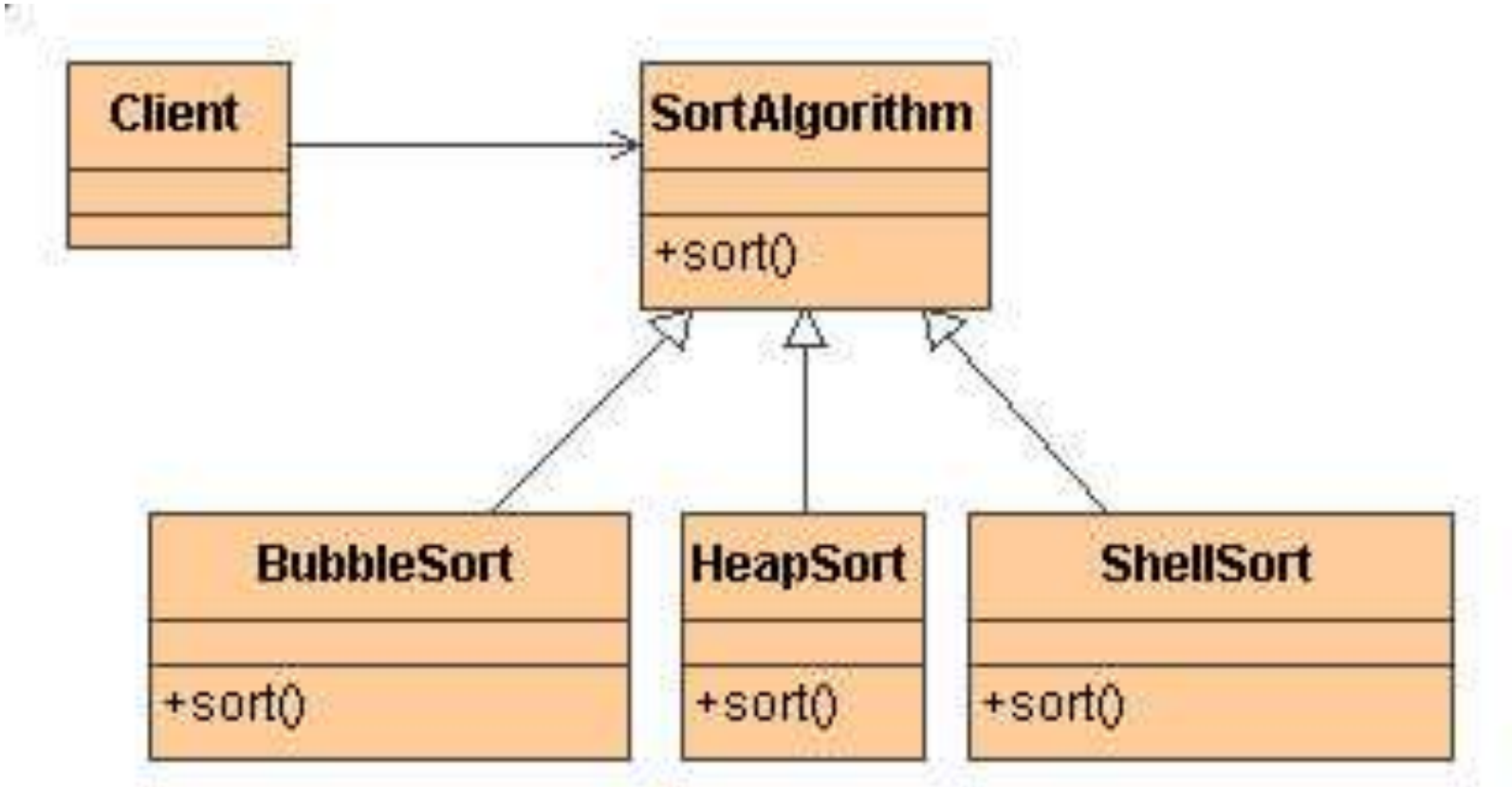
Strategy - Solution



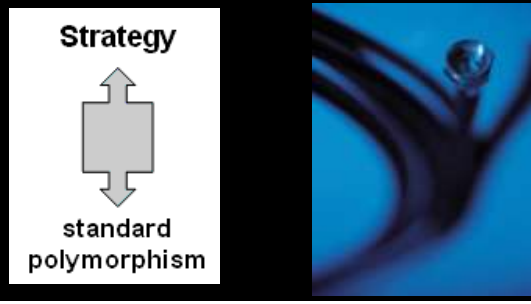
- Provide a way to configure the algorithm selection
- Wrapper / delegation structure
  - the client is a wrapper,
  - the algorithm object is a delegation.
- Adding an intermediate level for the client (couple the client to the interface).

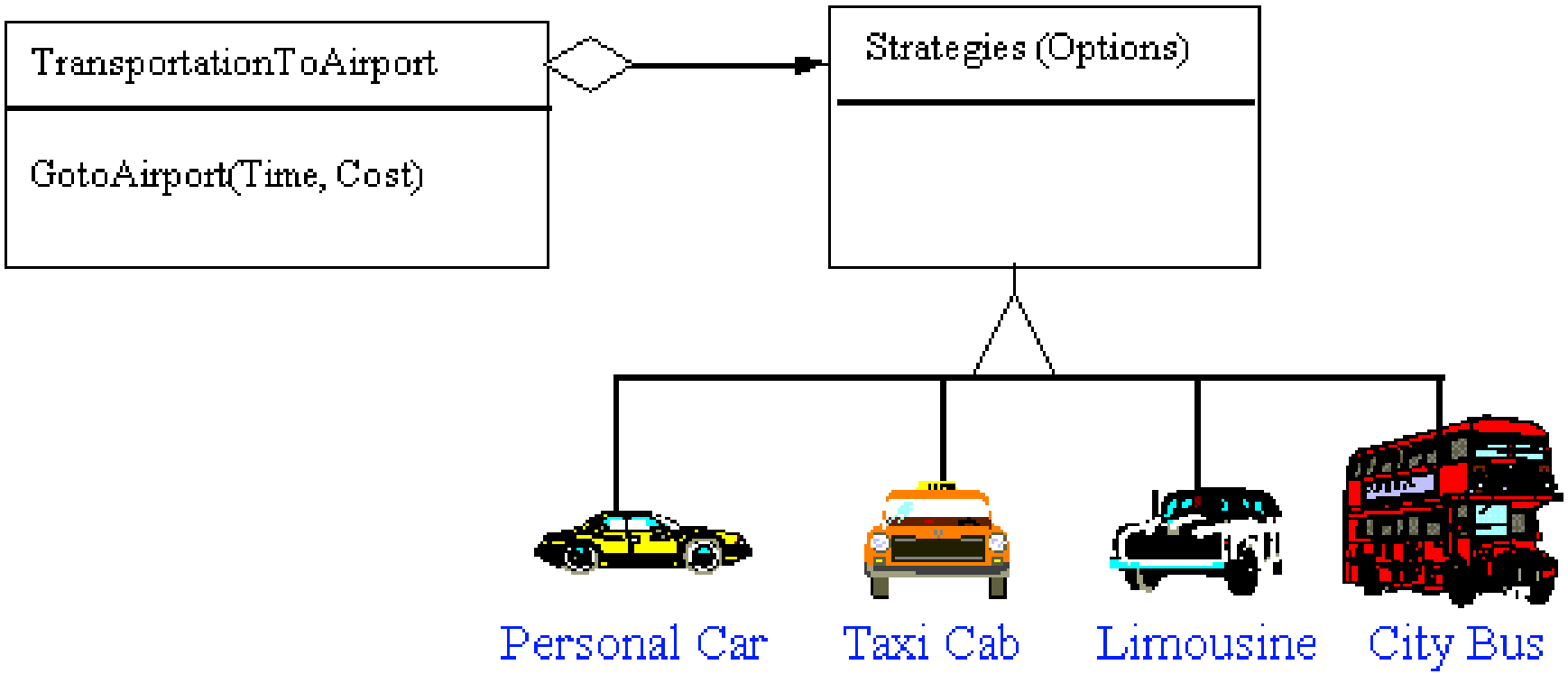
Strategy – Class diagram





Strategy – non-software example

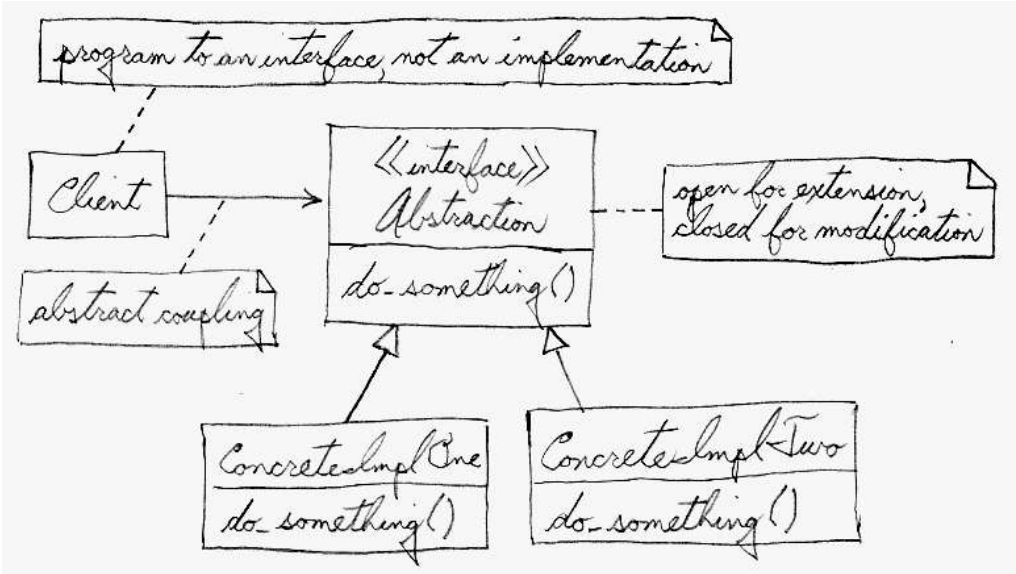




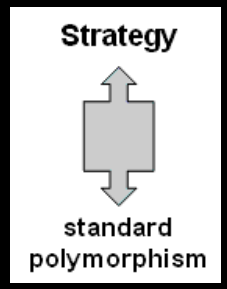
Open-Closed Principle

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

Bertrand Meyer, 1988



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.

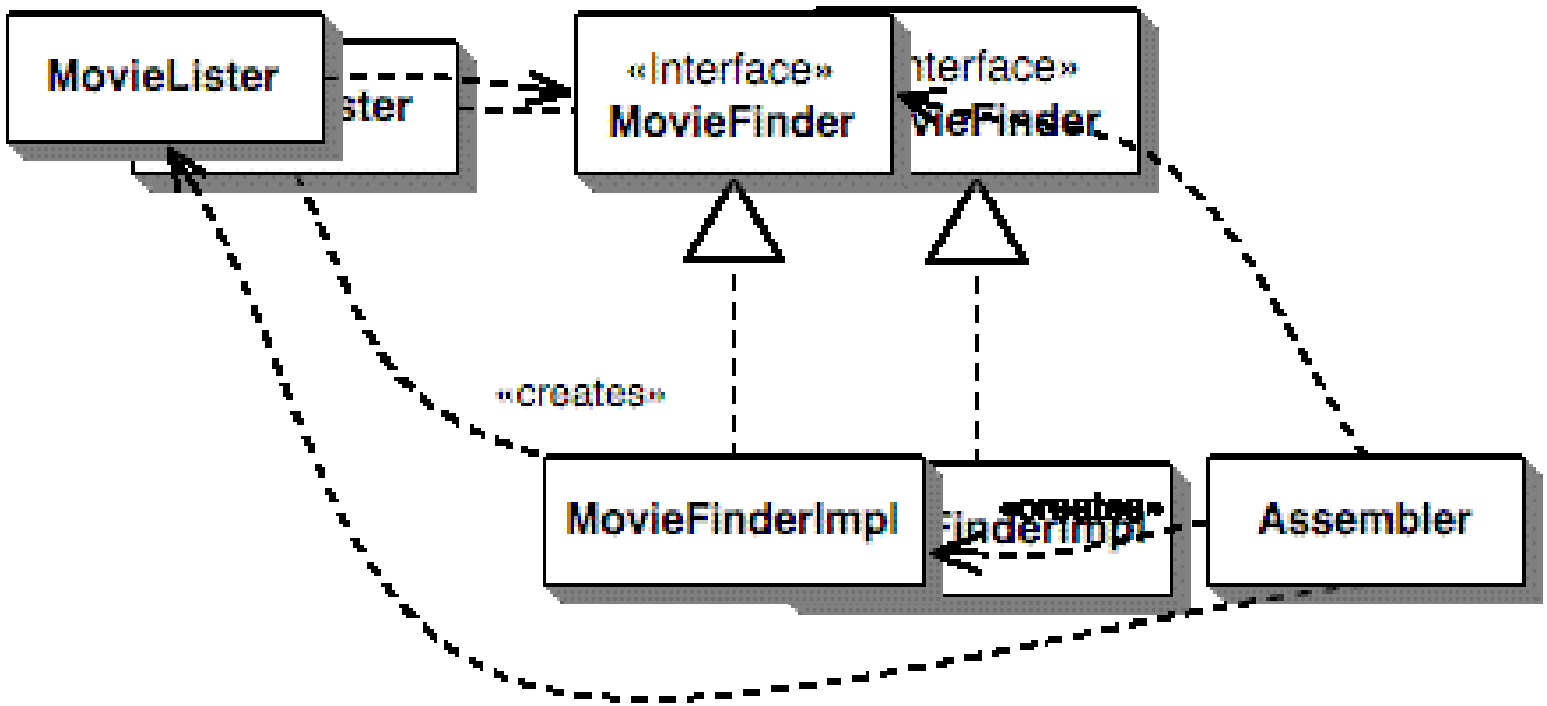
Code-smell?

- 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 …

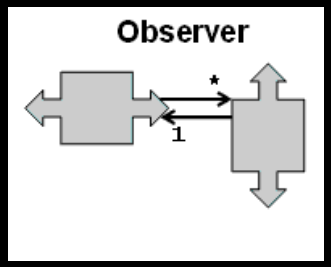
Configuration and Dependency Injection

- 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

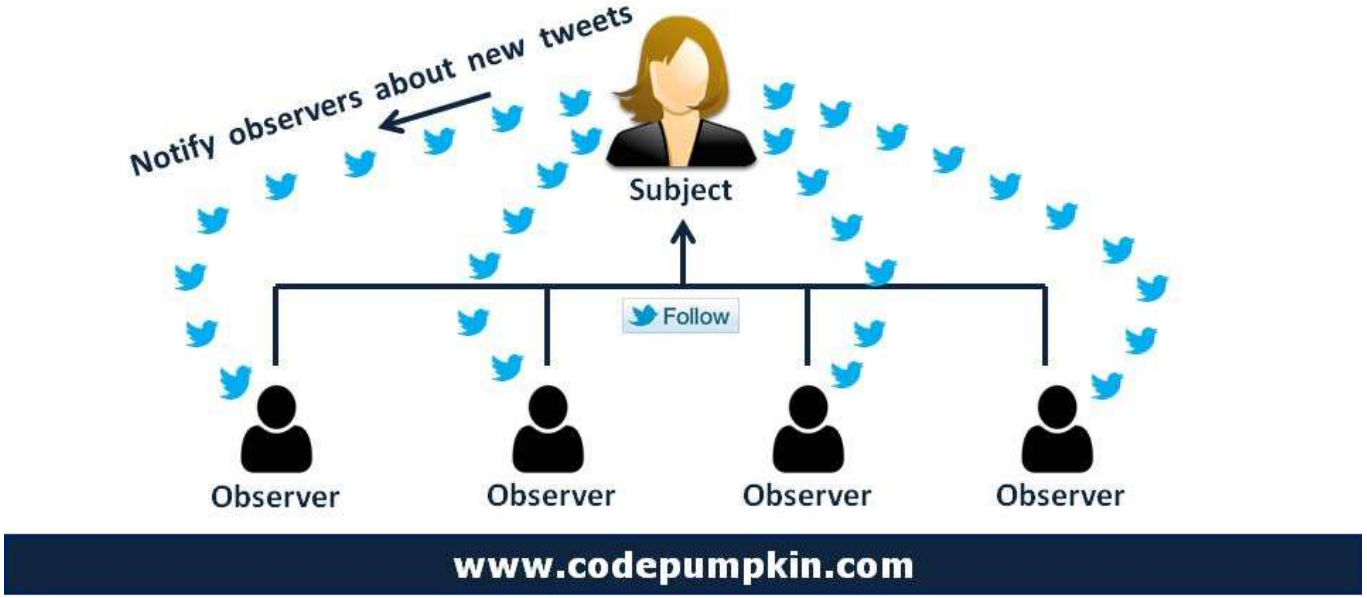
Dependency Injection – Naïve



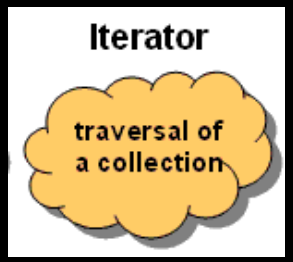
Observer



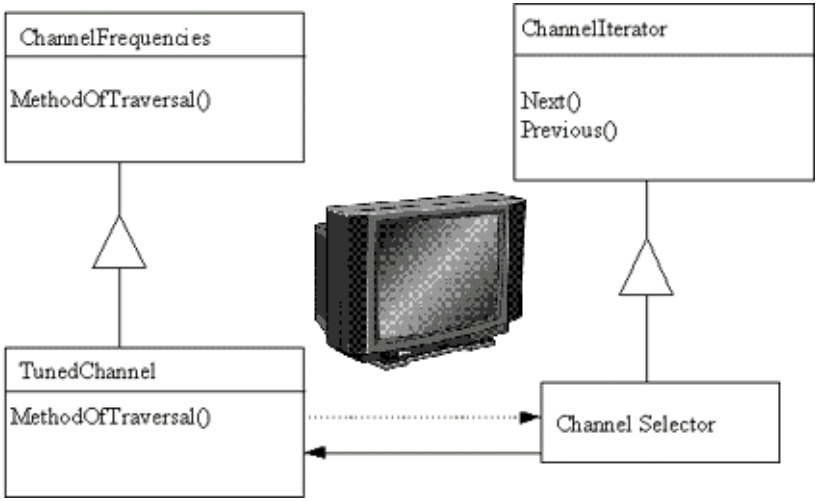
- It enables the separation of the object from the knowledge of objects which dependent on it



Iterator



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

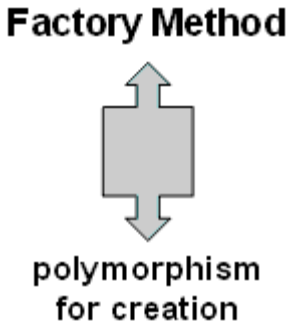


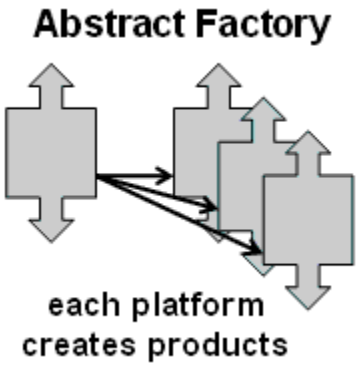
Creational patterns

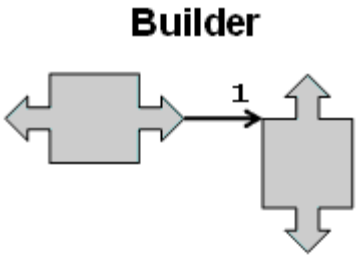
Gang of Four

Roadmap

- Factory Method
- Abstact Factory
- Singleton
- Builder





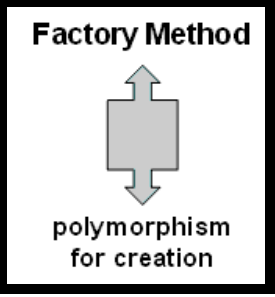


Creational patterns

- **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.

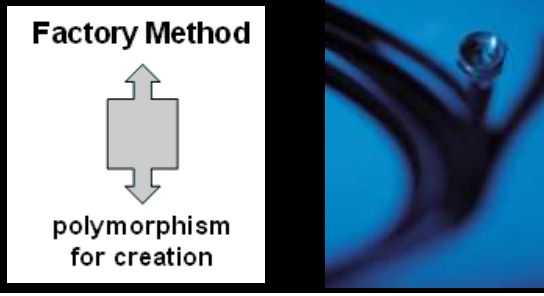
Factory Method



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

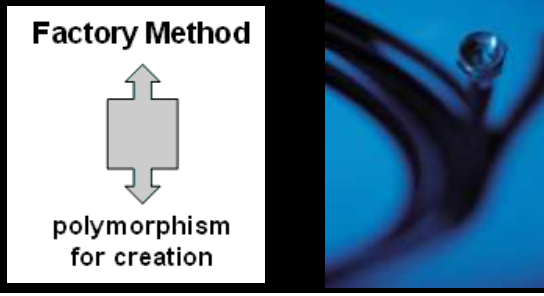
```
Pet p = new Parrot()
```

Factory Method – Problem



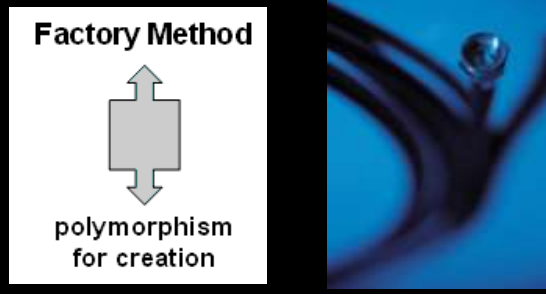
- A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.
- Classes must initiate object creations without having dependencies with the class of the created object.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

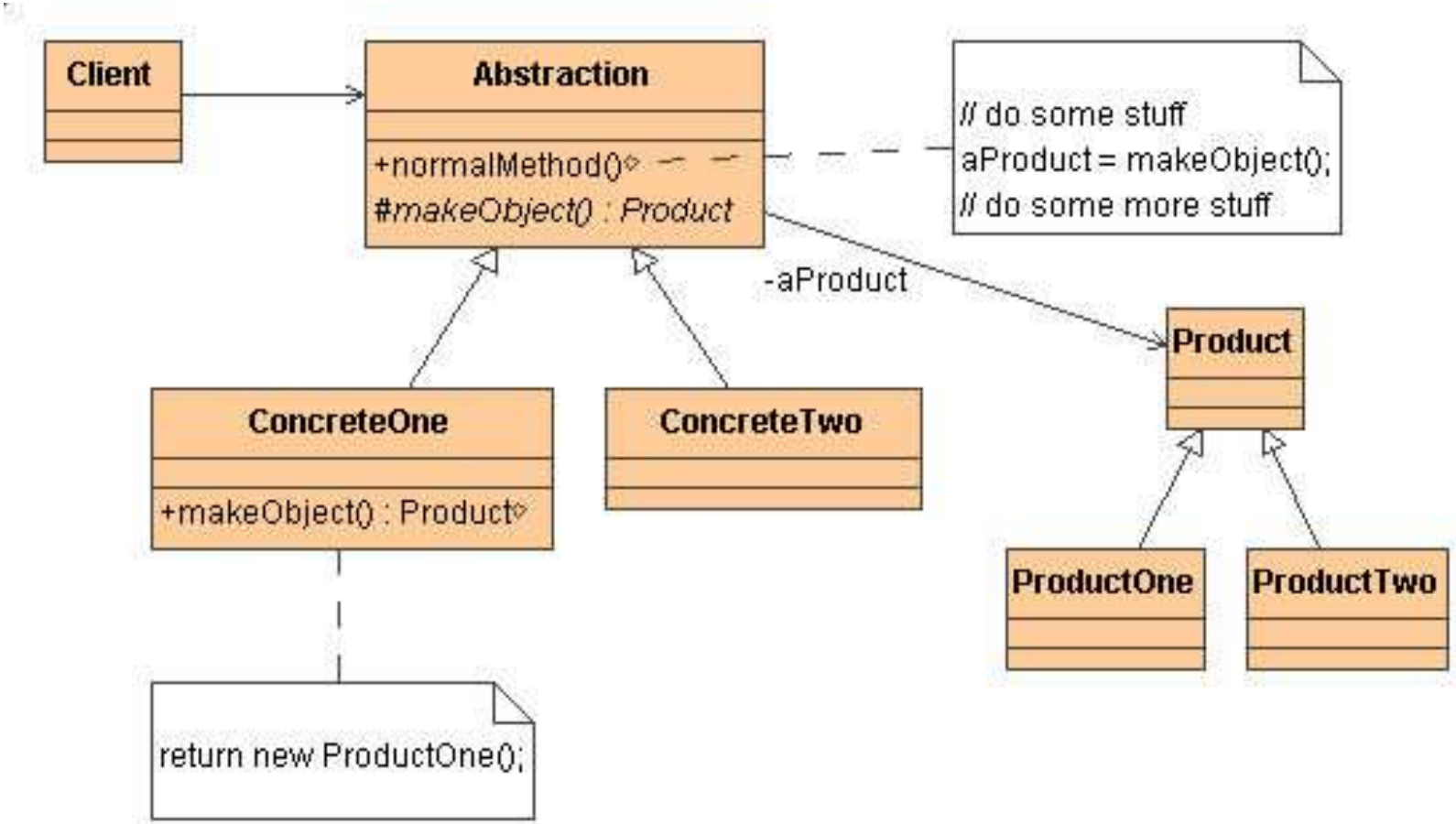
Factory Method – Solution



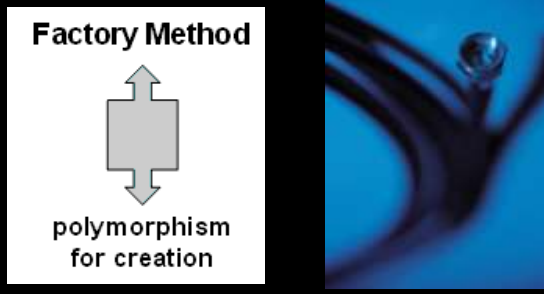
- Indirect creation using inheritance.
- Defining a "virtual" constructor.
- The new operator considered harmful.

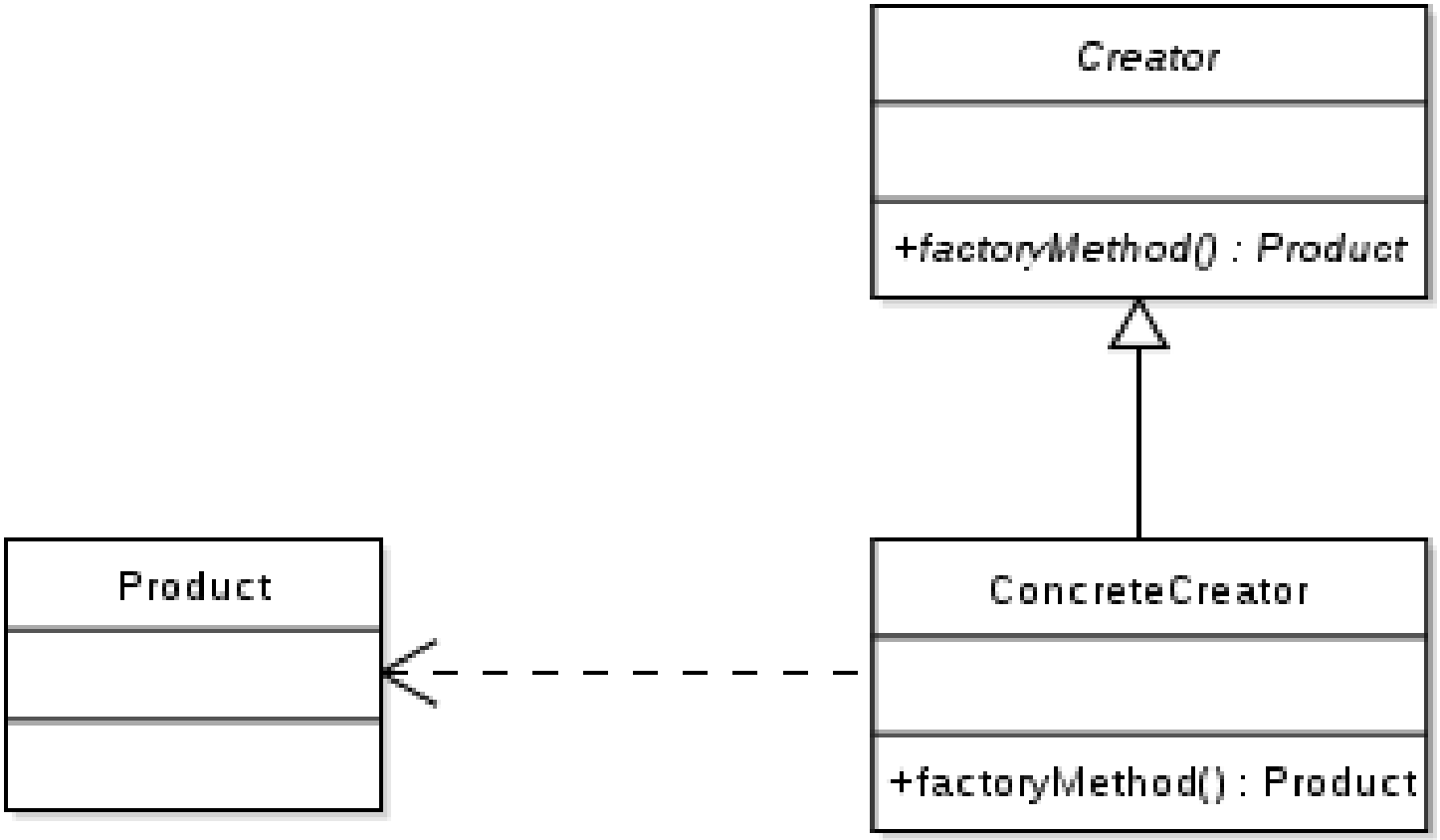
Factory Method – Class Diagram



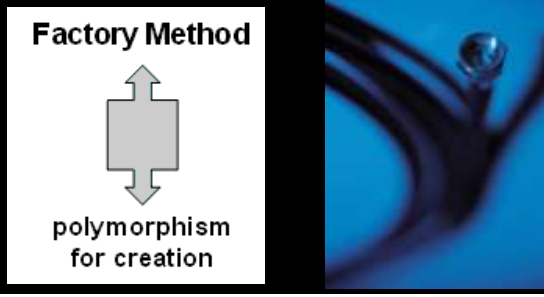


Factory Method – public variant





Factory Method and Iterator

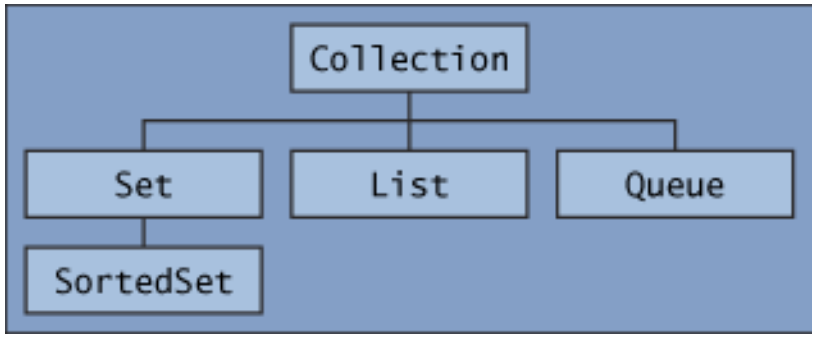


Iterators

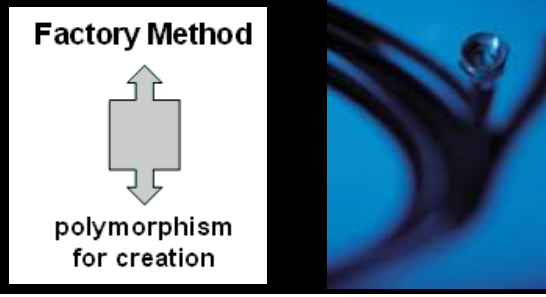
```
static void filter(Collection<?> c) {
        for (Iterator<?> it = c.iterator(); it.hasNext(); )
                if (!cond(it.next()))
                        it.remove();
}
```

for-each Construct

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

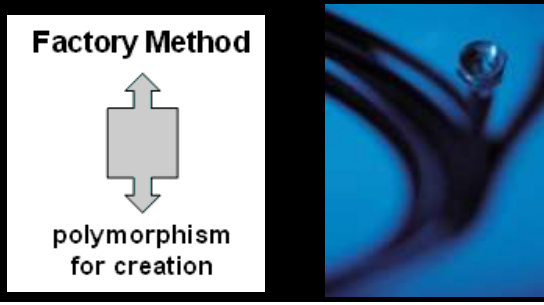


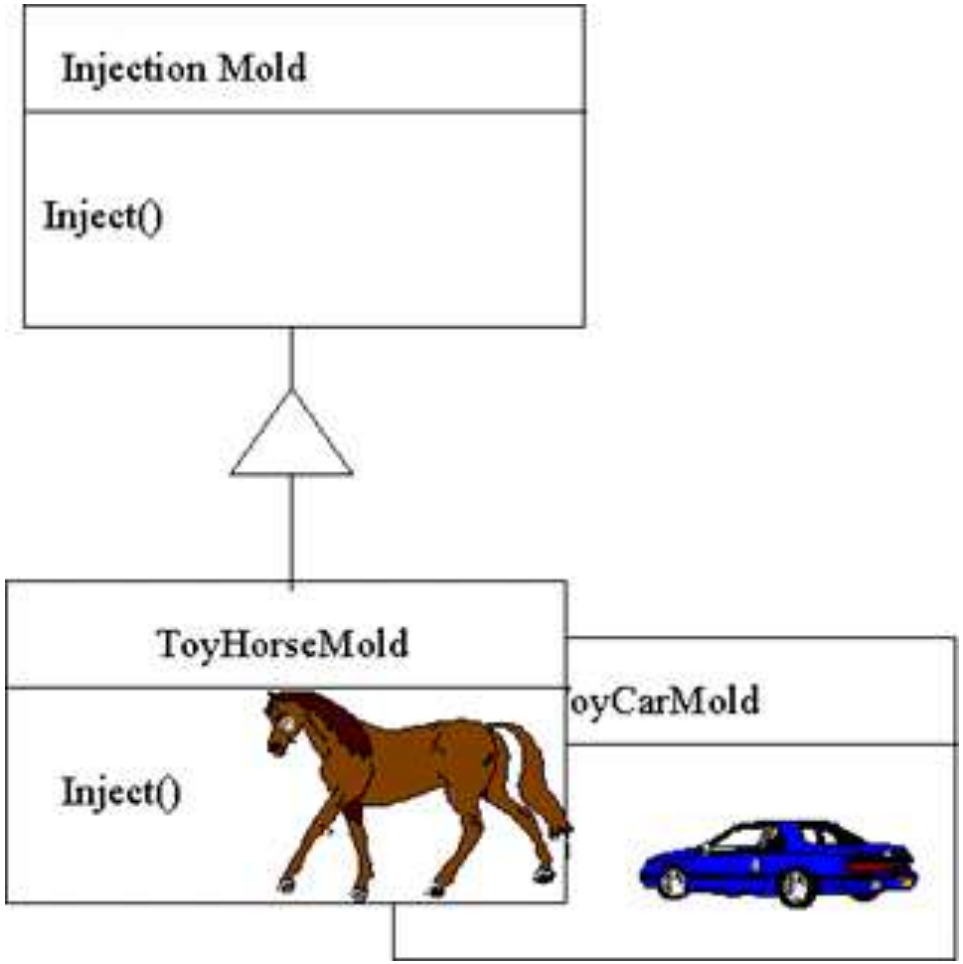
Factory Method – Consequences



- 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.

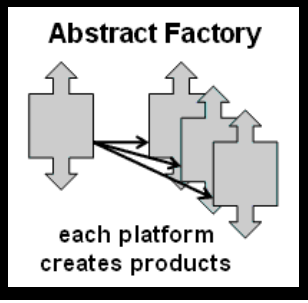
Factory Method – non-software example

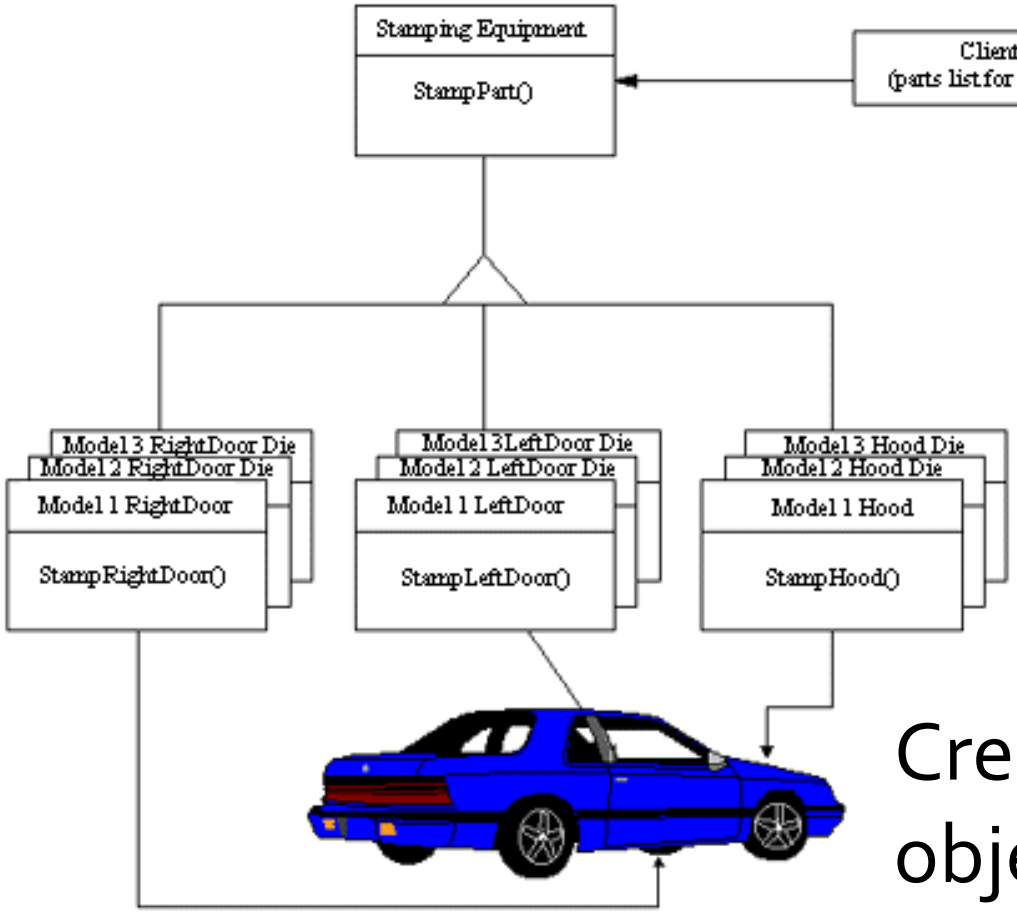




Wilhelmina Barns-Graham Trust

Abstract Factory

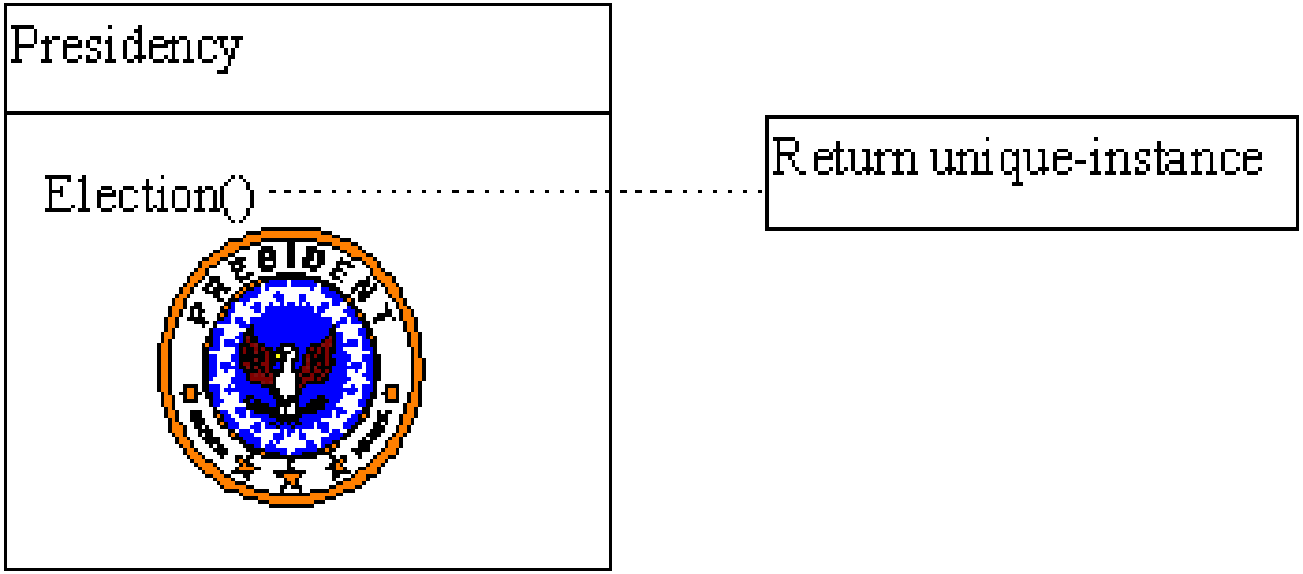




Creating a family of objects with a specific common feature.

Singleton

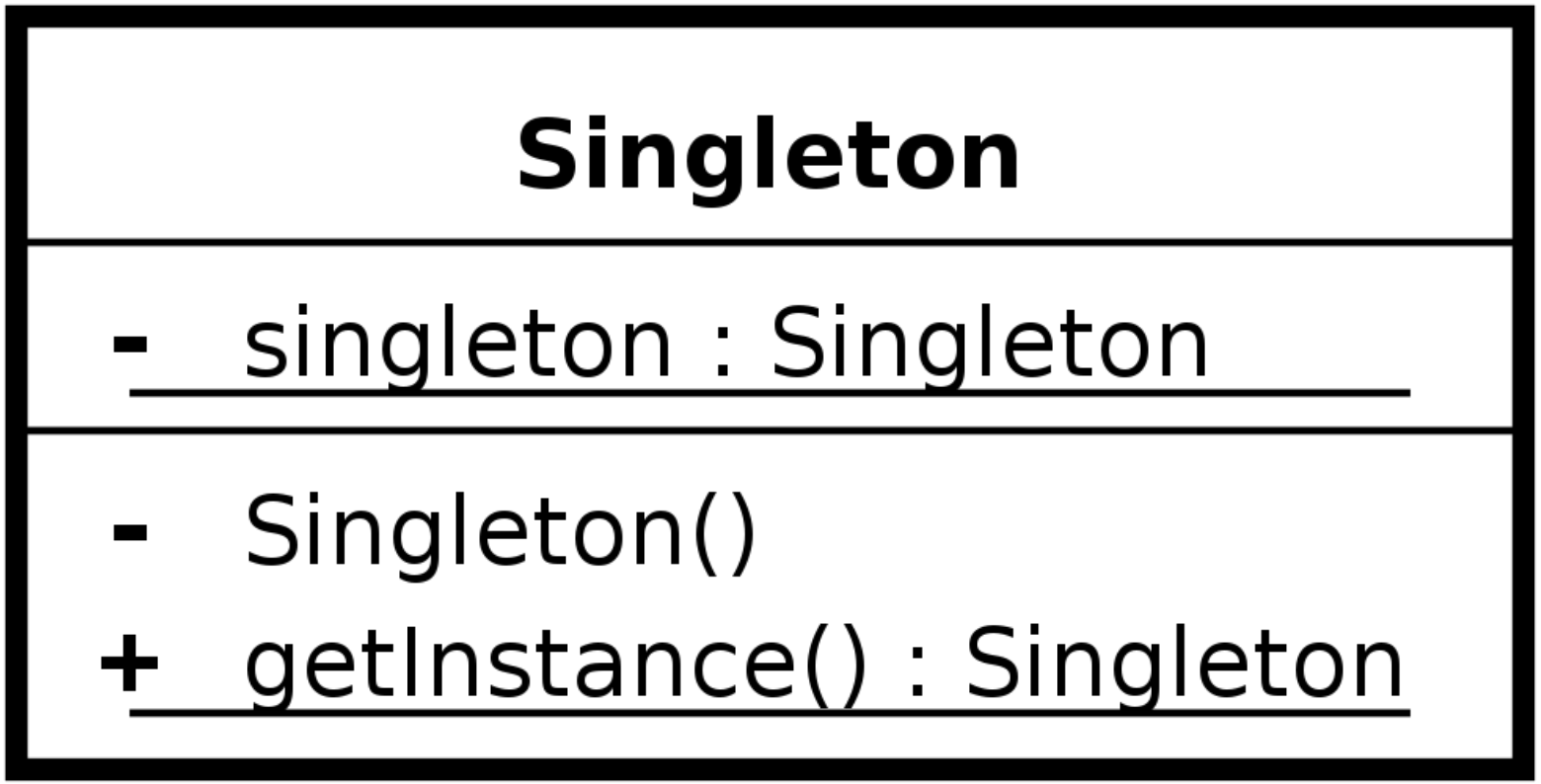
- Provides a way to concentrate **all responsibility** in **one instance** of the class



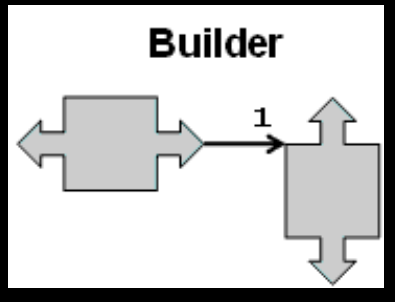
Singleton – Problem

- 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***).

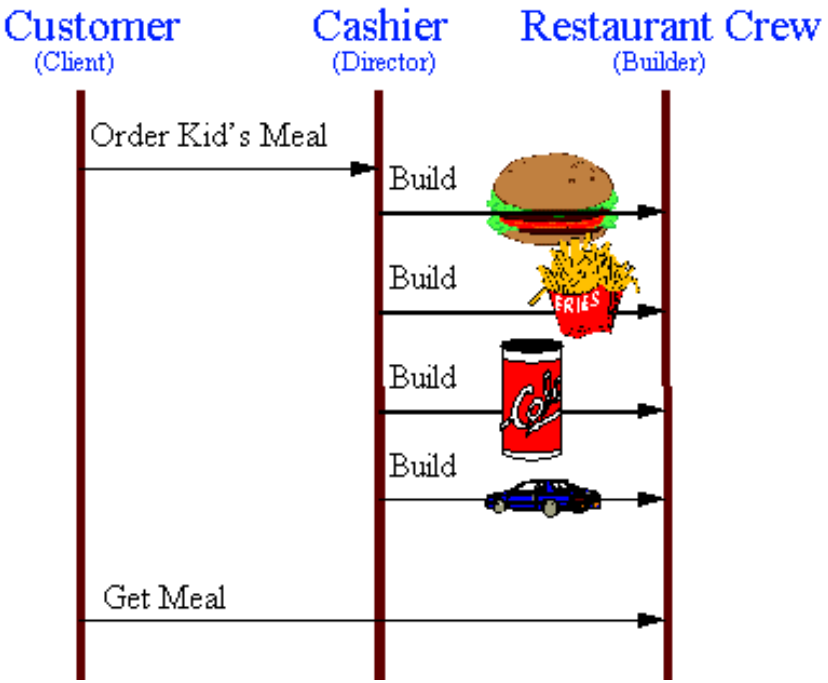
Singleton – Class Diagram



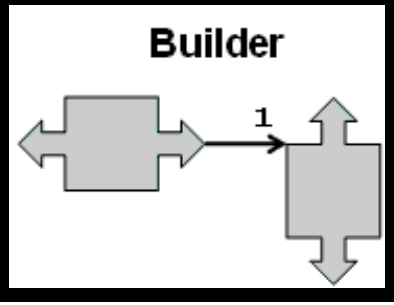
Builder

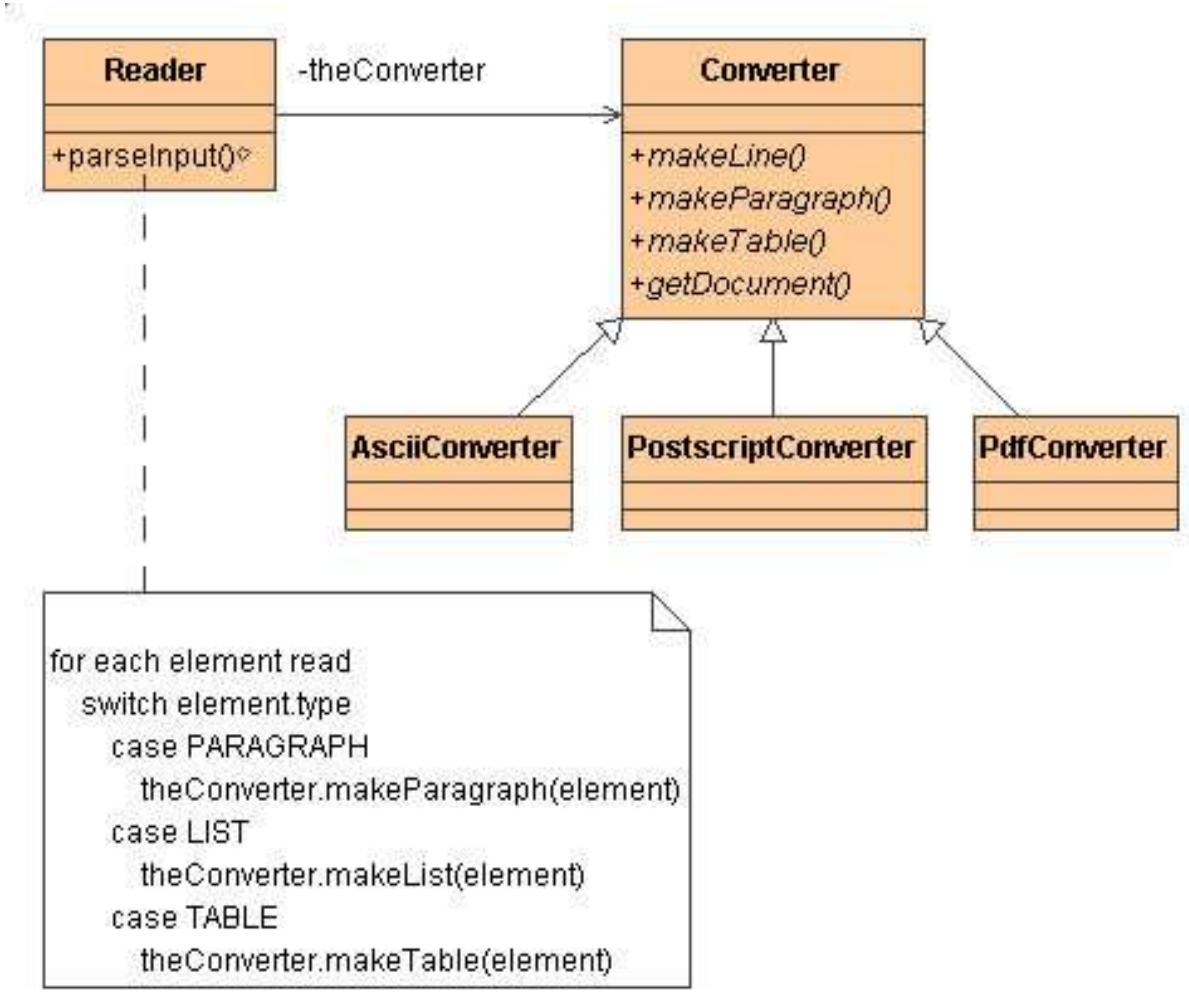


- Gradual gathering information about the object before its construction.



Builder – Class Diagram





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…*