Software Patterns

L03: Structural Design Patterns

Marlo Häring & Prof. W. Maalej (@maalejw)

Outline of the talk

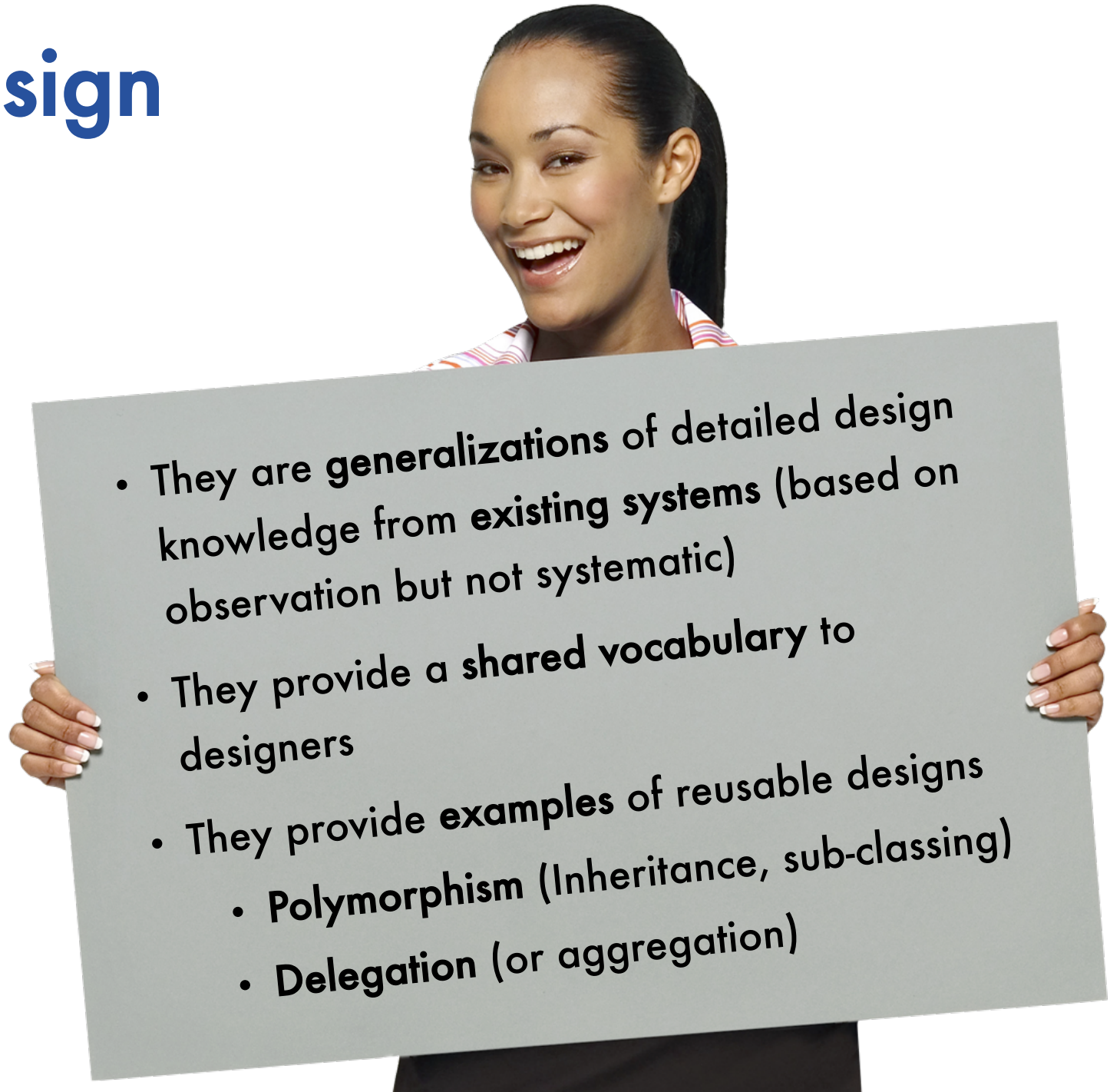
1. Taxonomy of Design Patterns
2. Adapter Pattern
3. Bridge Pattern
4. Proxy Pattern
5. Composite Pattern

Categorization of patterns

- **Architectural Pattern**
  - Expresses a fundamental **structural organization** or schema for software systems
  - Provides a set of predefined **subsystems** and specifies their responsibilities
  - Includes rules and guidelines for organizing the relationships between them
- **Idiom (Coding Pattern)**
  - A low-level pattern specific to a programming language
  - Describes how to implement particular aspects of components or the relationships between them using the features of the given language
- **Design Pattern**
  - Provides a **scheme for refining the subsystems** of a software system or the relationships between them
  - Describes **commonly recurring structure of communicating components** that solves a general design problem within a particular context

[Buschmann et al.]

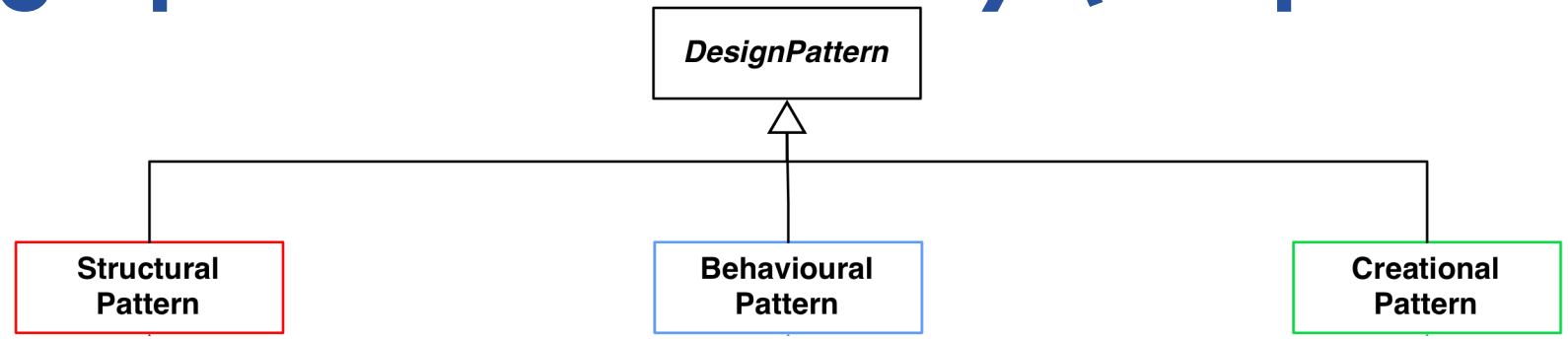
What makes design patterns useful?



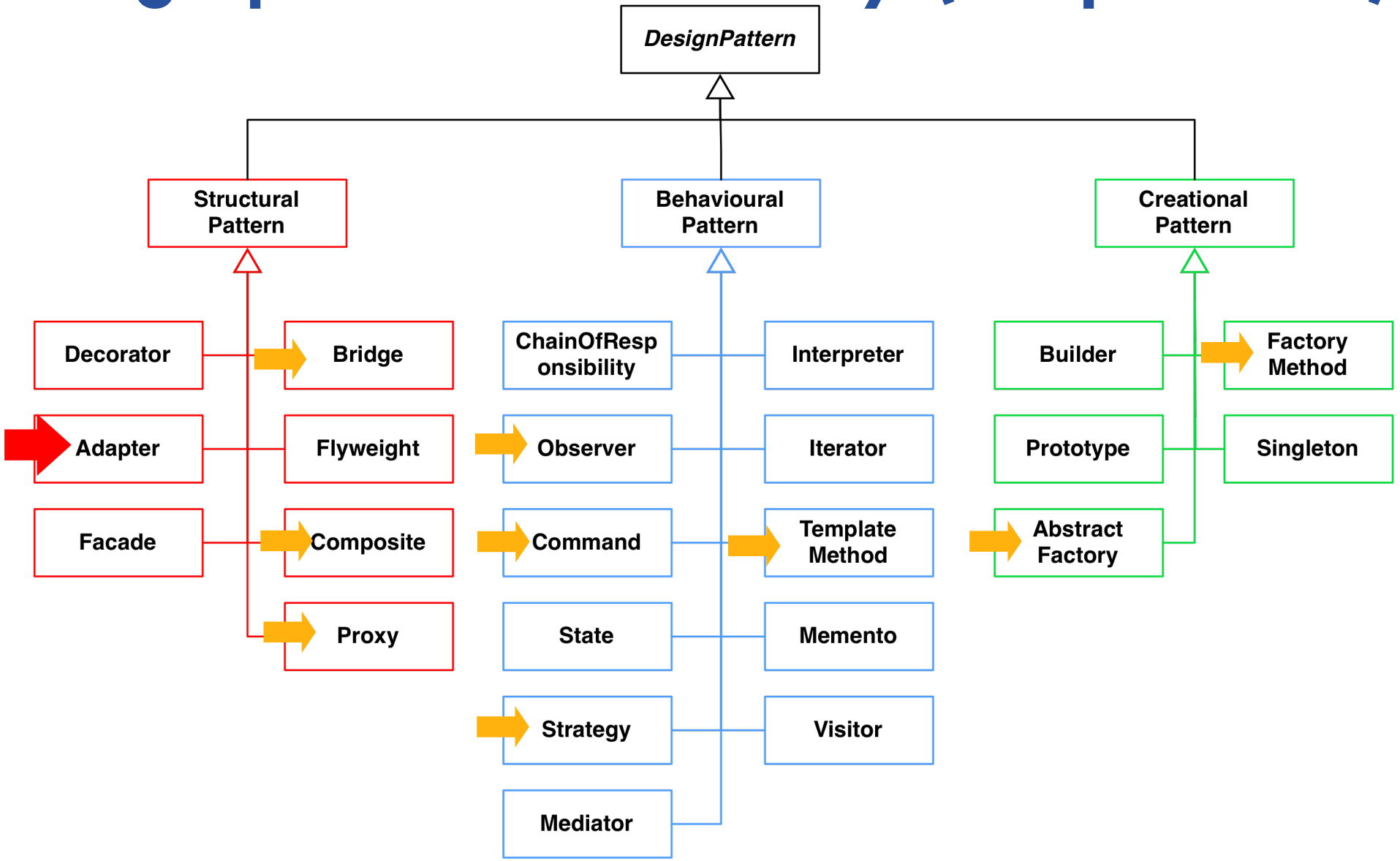
3 types of design patterns ("go4 patterns")

- **Structural Patterns**
  - Reduce coupling between two or more classes
  - Introduce an abstract class to enable future extensions
  - Encapsulate complex structures
- **Behavioral Patterns**
  - Allow a choice between algorithms and the assignment of responsibilities to objects ("Who does what?")
  - Simplify complex control flows that are difficult to follow at runtime
- **Creational Patterns**
  - Allow a simplified view from complex instantiation processes
  - Make the system independent from the way its objects are created, composed and represented

Design patterns taxonomy (23 patterns)



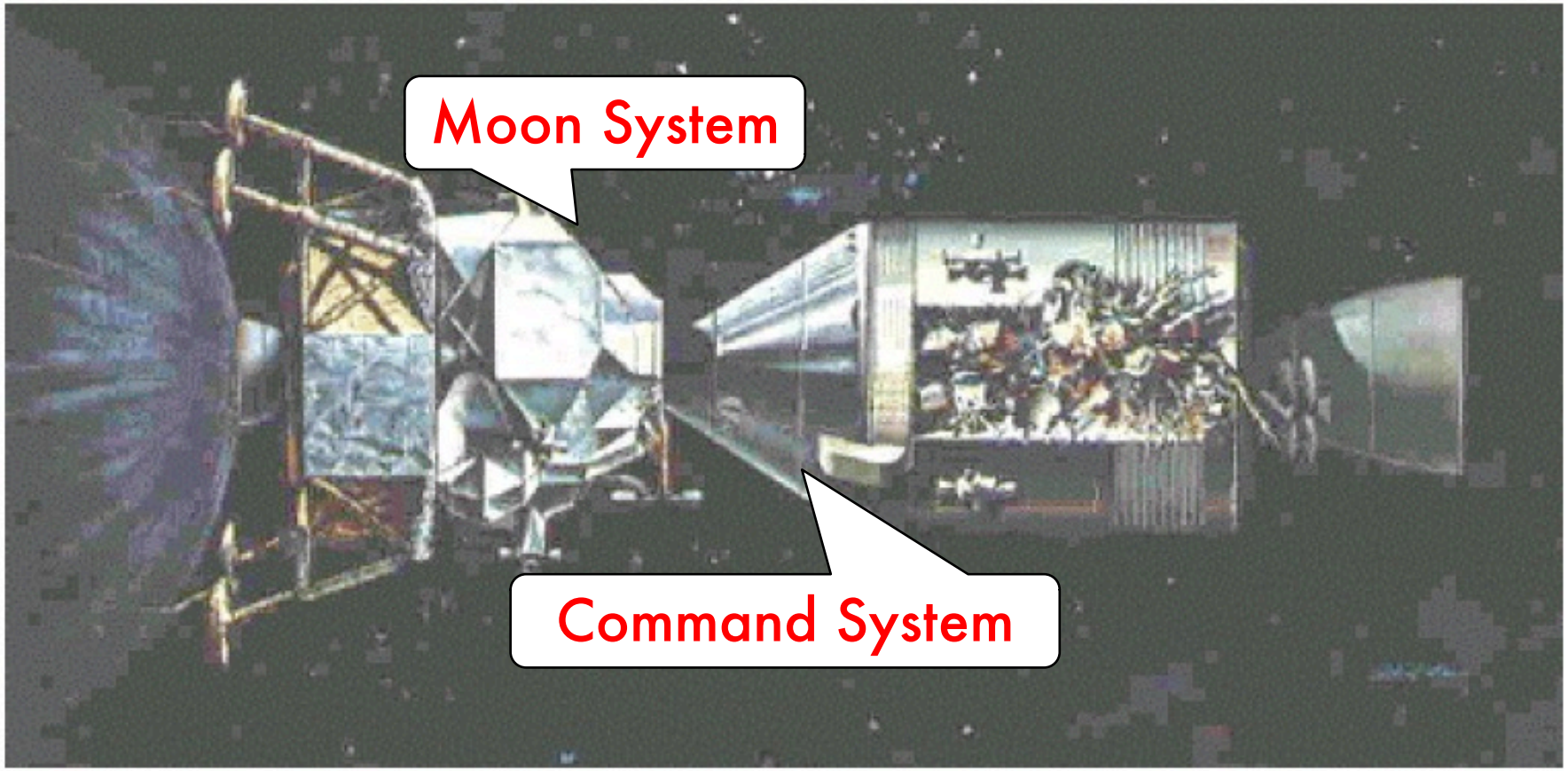
Design patterns taxonomy (23 patterns)



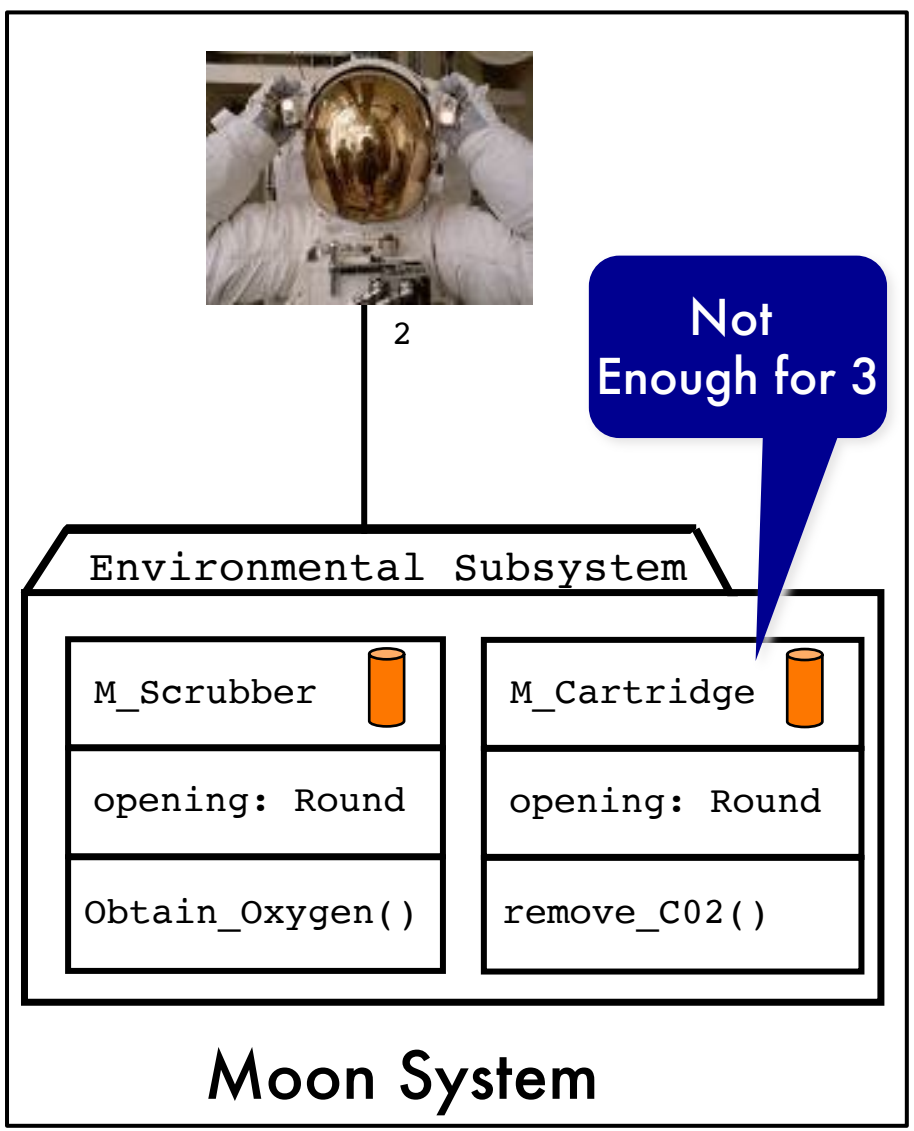
Outline of the talk

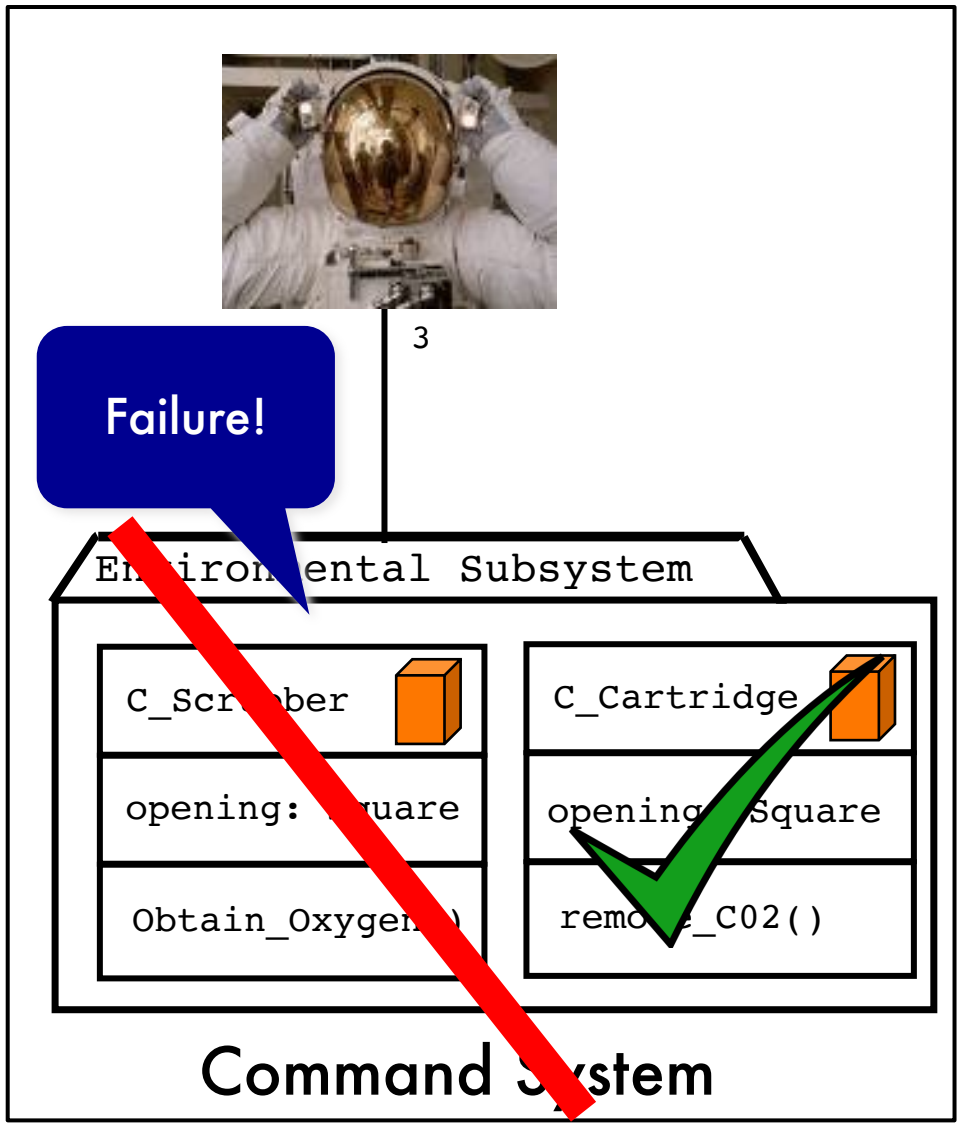
1. Taxonomy of Design Patterns
2. Adapter Pattern
3. Bridge Pattern
4. Proxy Pattern
5. Composite Pattern

Example Apollo 13: "Houston, we've had a problem!"

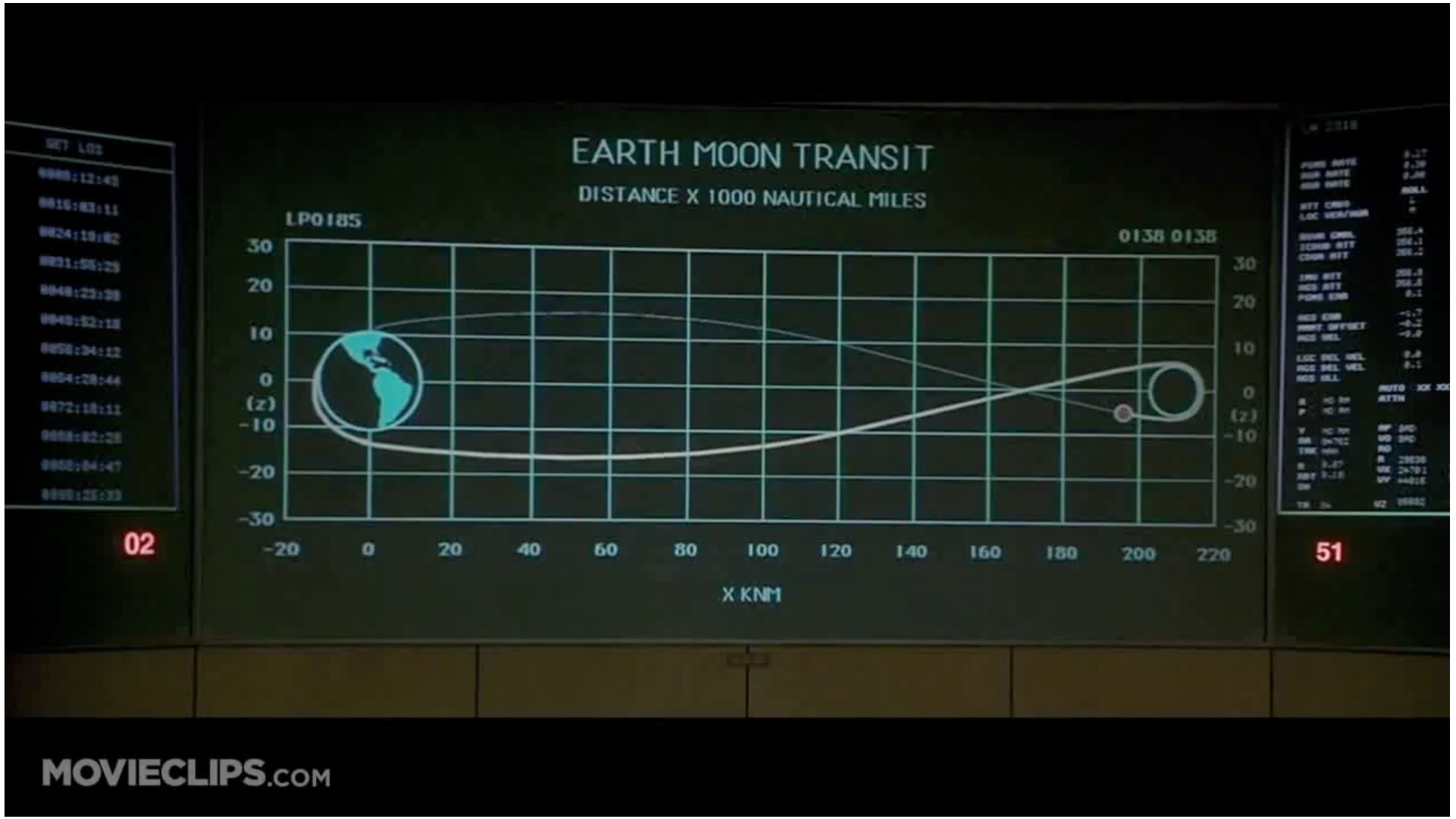


Original design of the Apollo 13 environmental subsystems





"Fitting a square peg in a round hole"



[Source: http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html]

Adapter pattern

- Connects incompatible components
  - Allows the simplified reuse of existing components
  - Converts the interface of an existing component into another interface expected by the calling component
  - Useful in interface engineering projects and in reengineering projects
    - Used to provide a new interface for (the ugly interface of) legacy systems
- Also known as a wrapper

A famous example

Another example

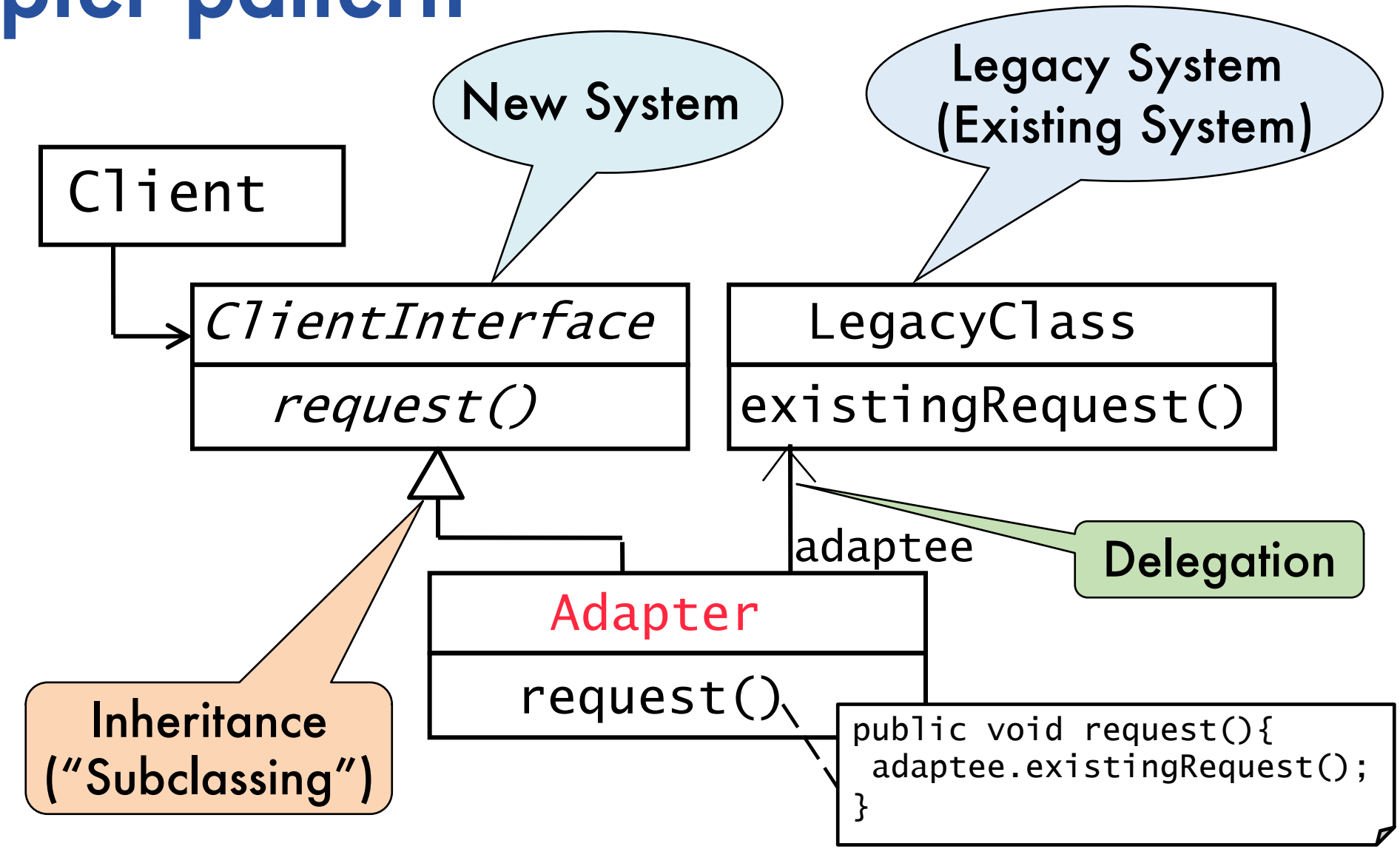
Using the adapter pattern

- Main use: Provide access to legacy systems
- Legacy System:
  - is an old system that continues to be used, even though newer technology or more efficient methods are available
- Why are legacy systems in use?
  - System Cost: System still makes money, the cost for a new system is too high
  - Pragmatism: System is installed and working
  - Poor Engineering: Compiler is no longer available, or source code is lost
  - Availability: System requires high availability, cannot simply be taken out of service to replace it with a new system

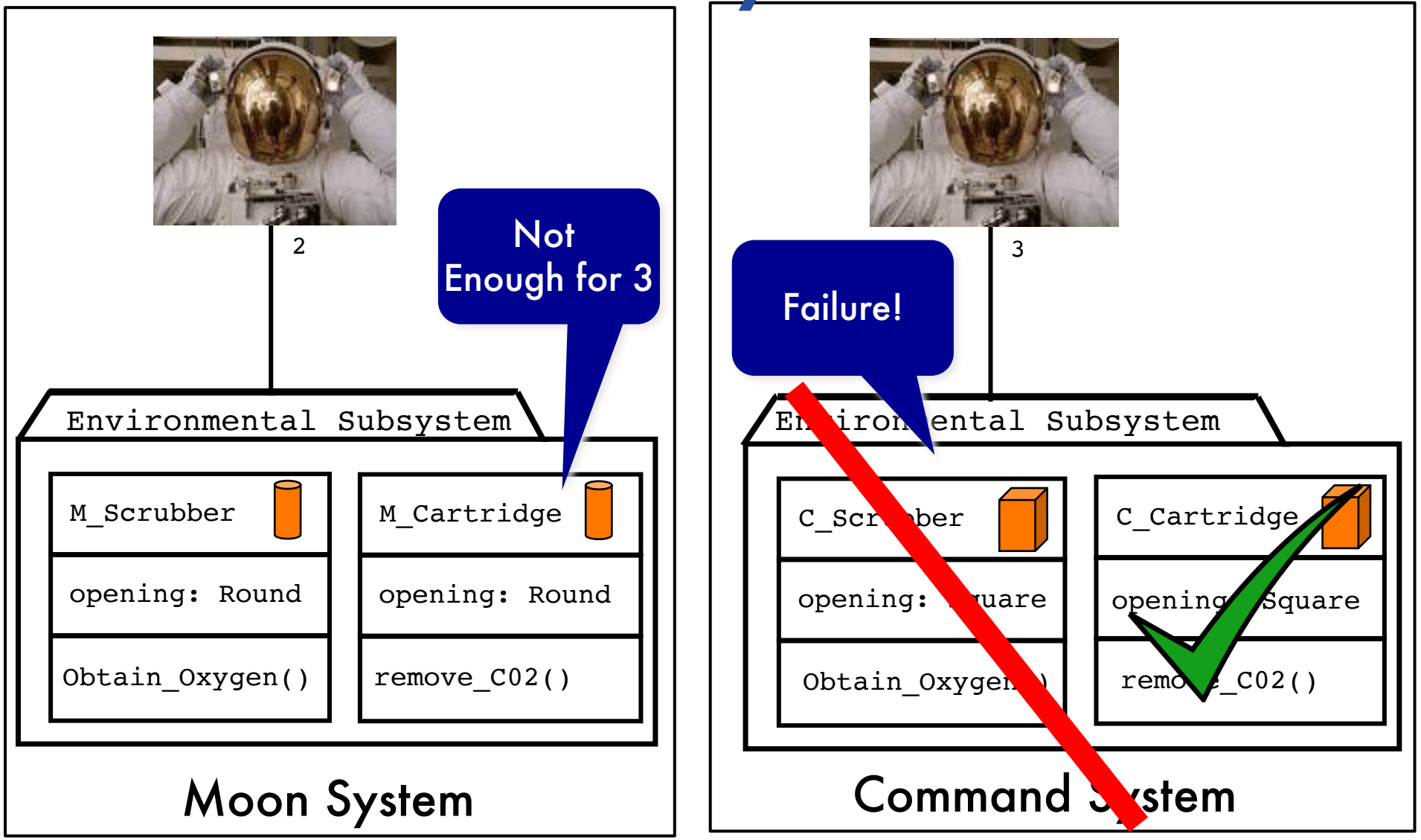
Examples of legacy systems

- Customer relationship management (CRM) systems
- NASA's Space Shuttle program (still uses 1970 technology)
- Air traffic control systems (FAA, Federal Aviation Administration)
- Energy distribution (power grids)
- Nuclear power plants, military defense installations
- SAP R/3 kernel
  - Client-Server Architecture. Last release in 2003
  - The new system is SAP ERP
    - Based on NetWeaver to provide access through Web Services (SOA)
    - SAP ERP is still using legacy SAP R/3 modules

Adapter pattern



Original design of the Apollo 13 environmental subsystems



Original design of the Apollo 13 environmental subsystems

2 3

Not Enough for 3

Environmental Subsystem

M_Scrubber	M_Cartridge
opening: Round	opening: Round
Obtain_Oxygen()	remove_C02()

Moon System

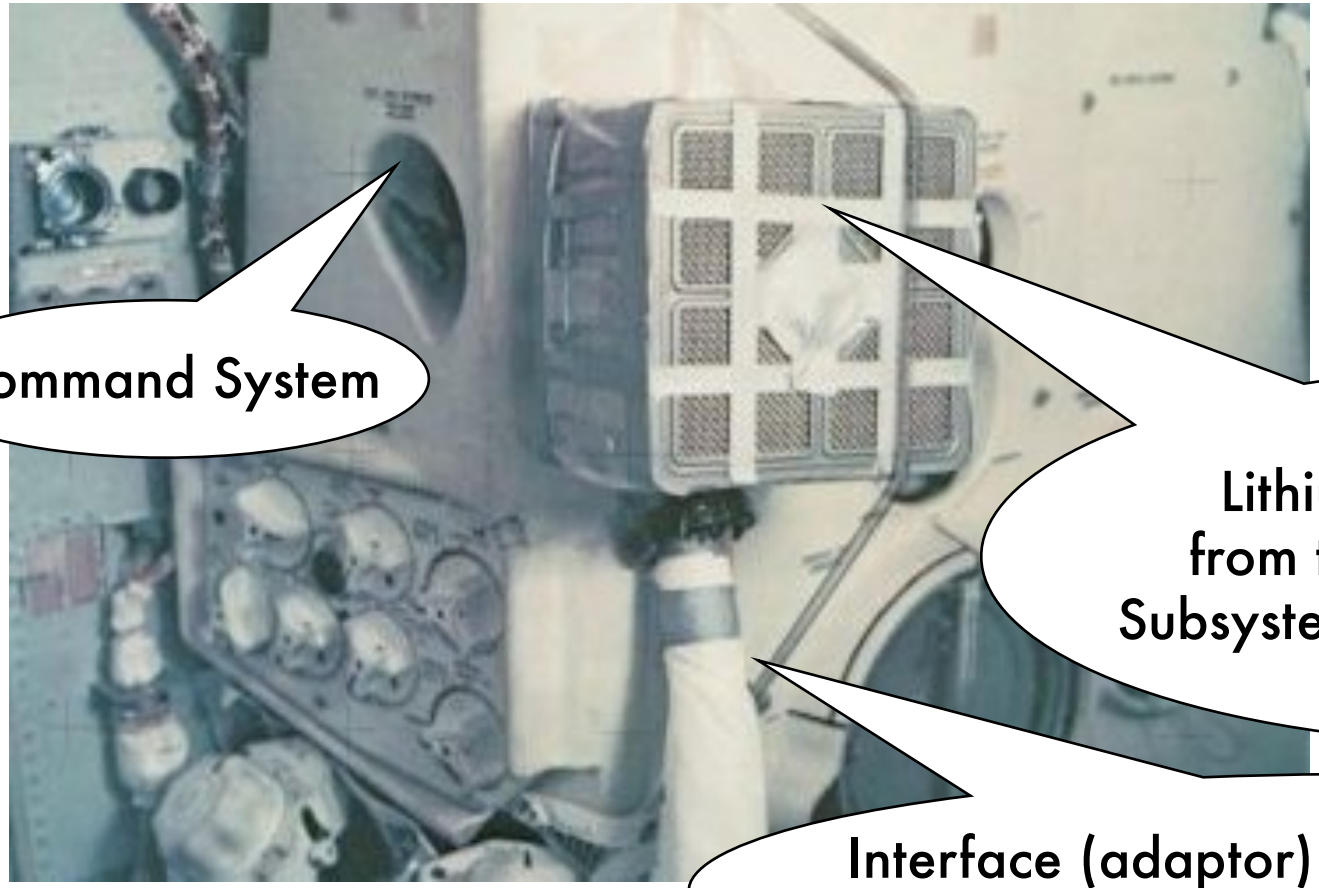
Round_To_Square_Adapter

Obtain_Oxygen()

adaptee

C_Cartridge
opening: Square
remove_C02()

A typical design challenge: connecting incompatible components



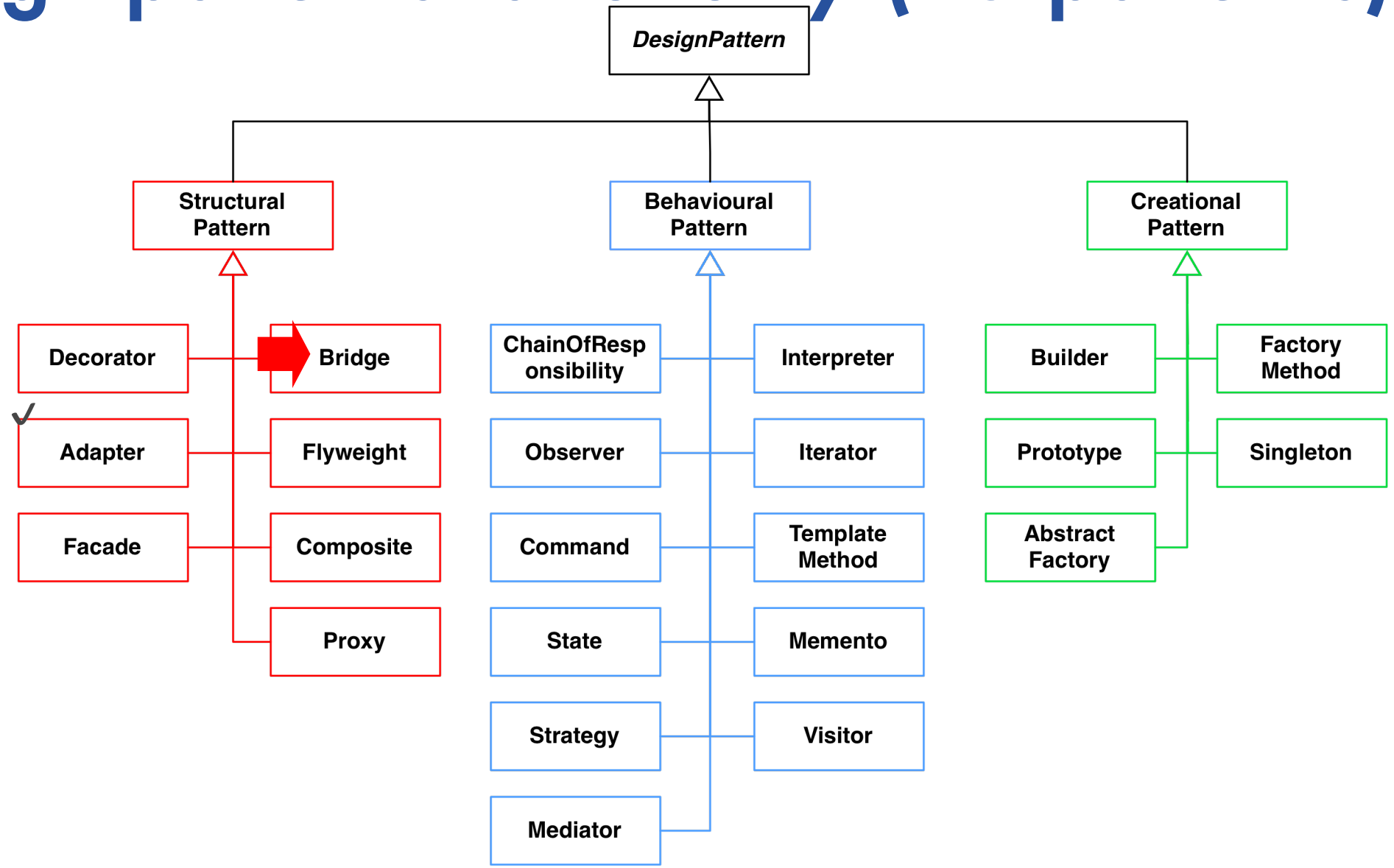
Command System

Lithium hydride canister from the Command System Subsystem using square openings

Interface (adaptor) to the Scrubber in the Lunar System subsystem using round openings

Source: http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html

Design patterns taxonomy (23 patterns)

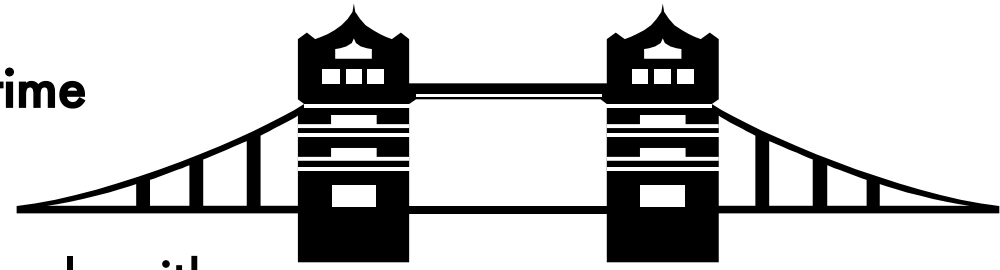


Outline of the talk

1. Taxonomy of Design Patterns
2. Adapter Pattern
3. **Bridge Pattern**
4. Proxy Pattern
5. Composite Pattern

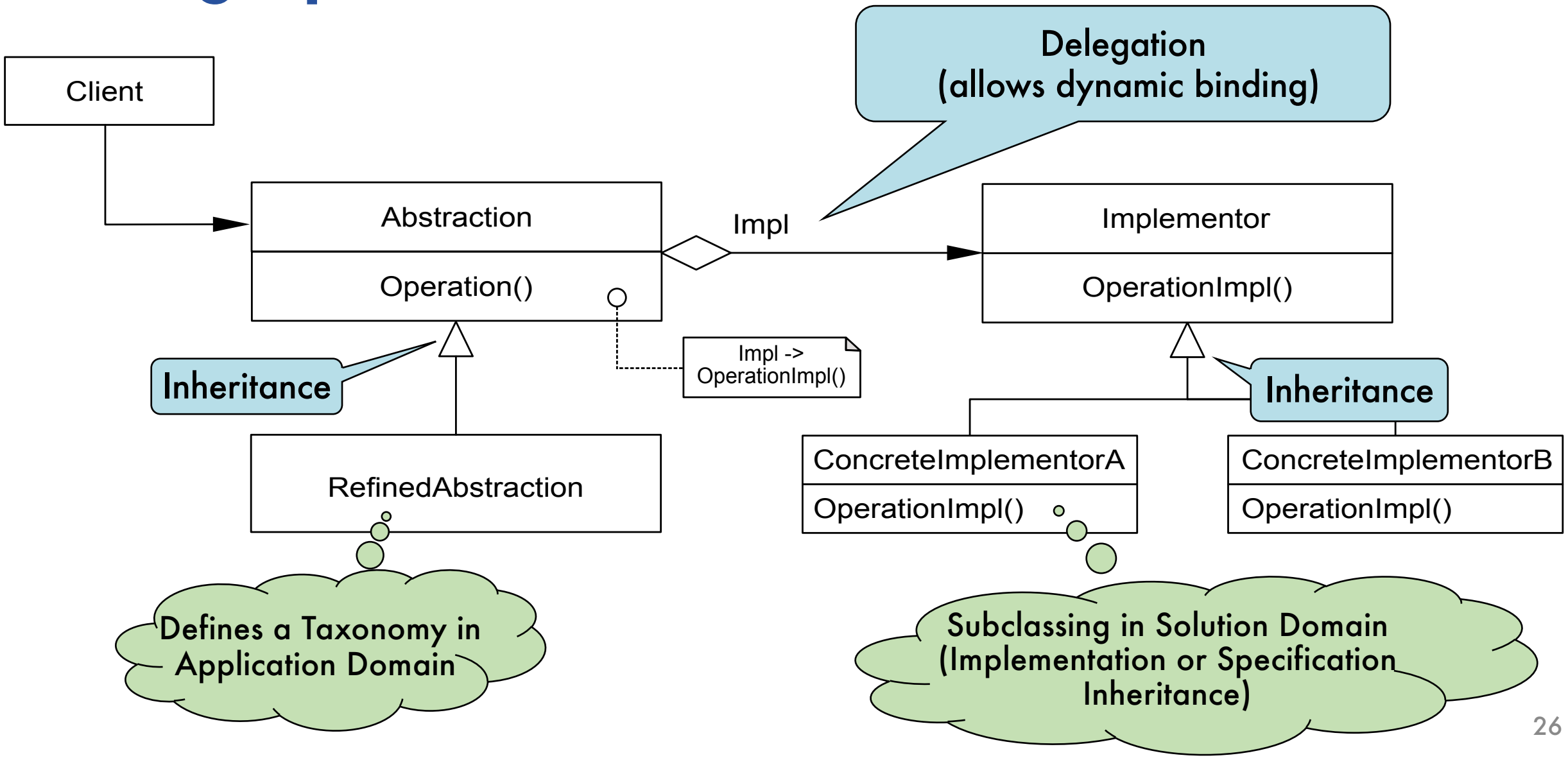
Postpone design decisions to the start time of a system

- **Problem:** Many design decisions have to be made final at **design time** ("design window") or at **compile time**
  - Example: Choosing a specific encryption algorithm
- Often it is desirable to delay design decisions until **run time**
  - Example: We want to support two types of clients
    - One client uses a very old implementation of an algorithm
    - The other client uses a modern implementation of the algorithm



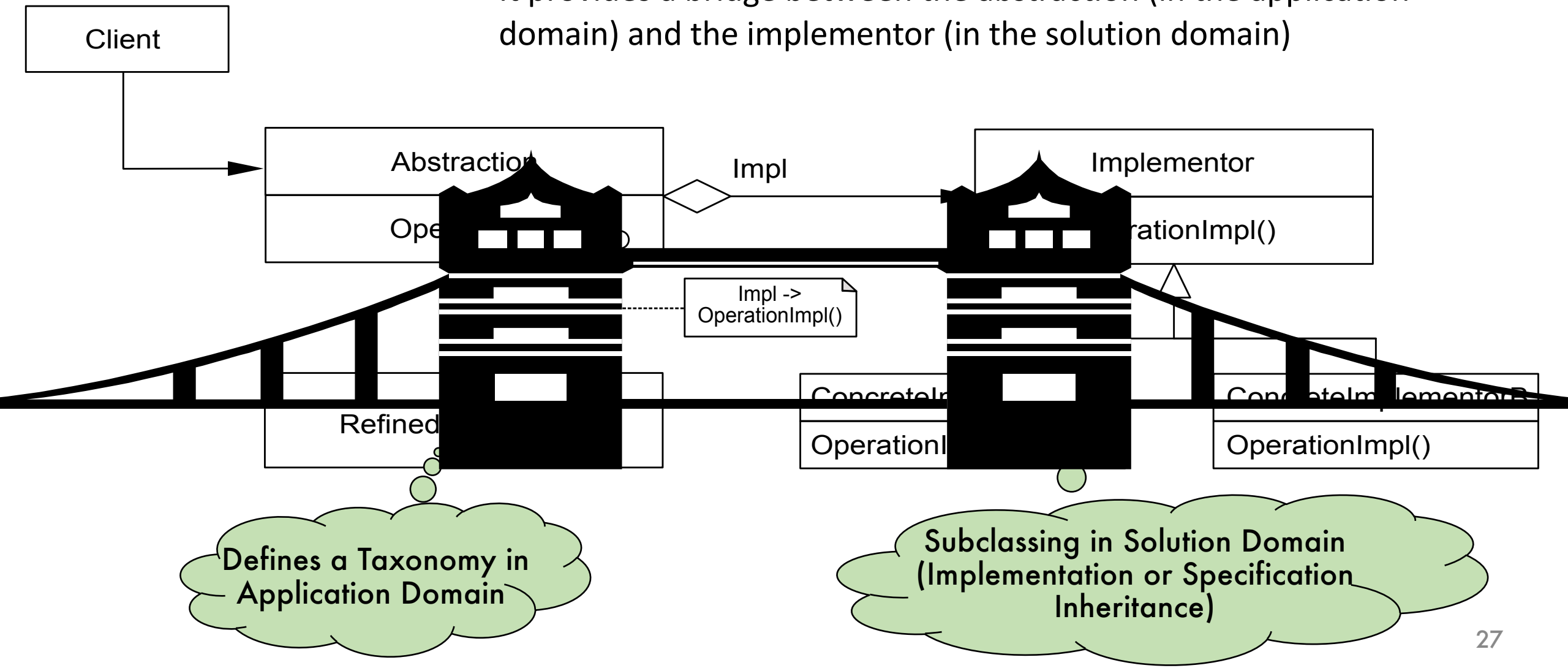
- The **bridge pattern** allows to delay the binding between an interface and its subclass to the start-up time of the system (e.g. in the constructor of the interface class)

Bridge pattern

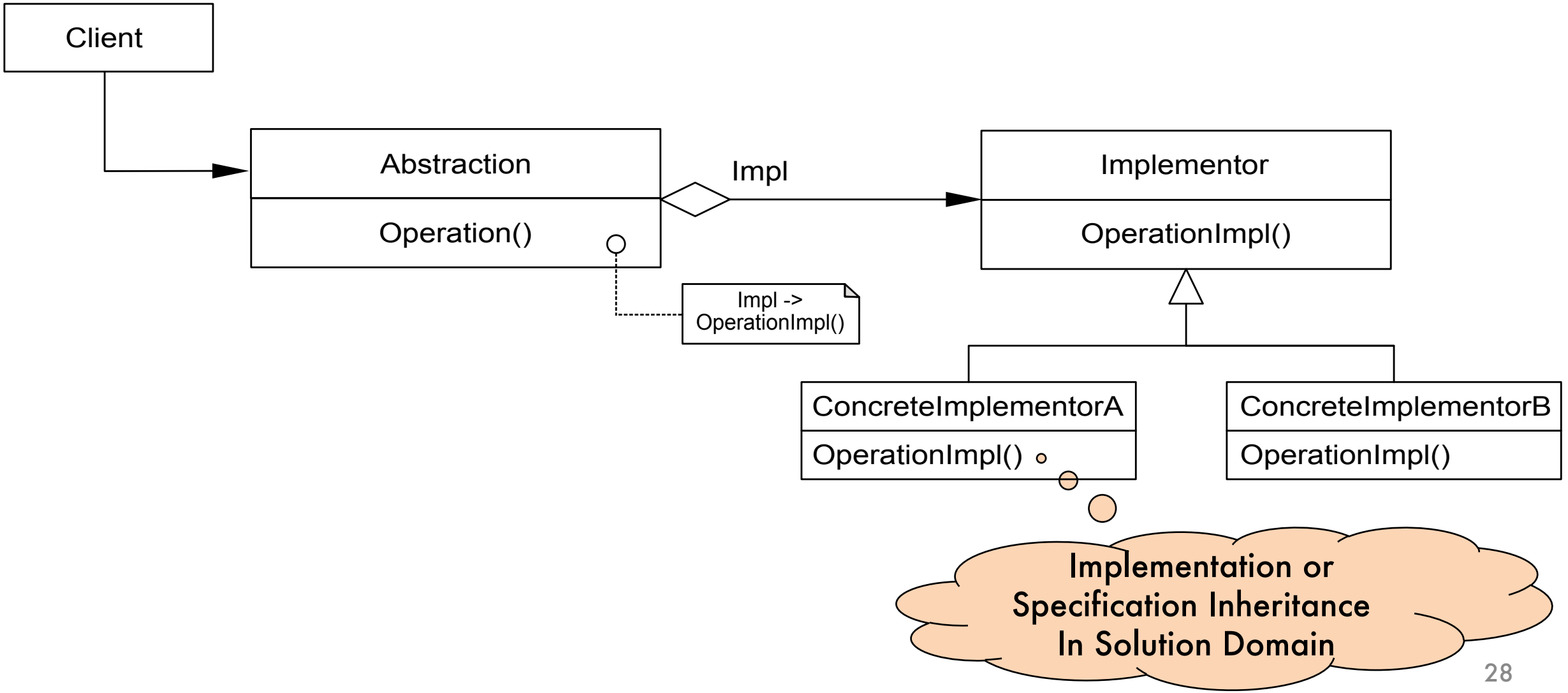


Bridge pattern

- It provides a bridge between the abstraction (in the application domain) and the implementor (in the solution domain)



Often only A "degenerated bridge" is used (no taxonomy)

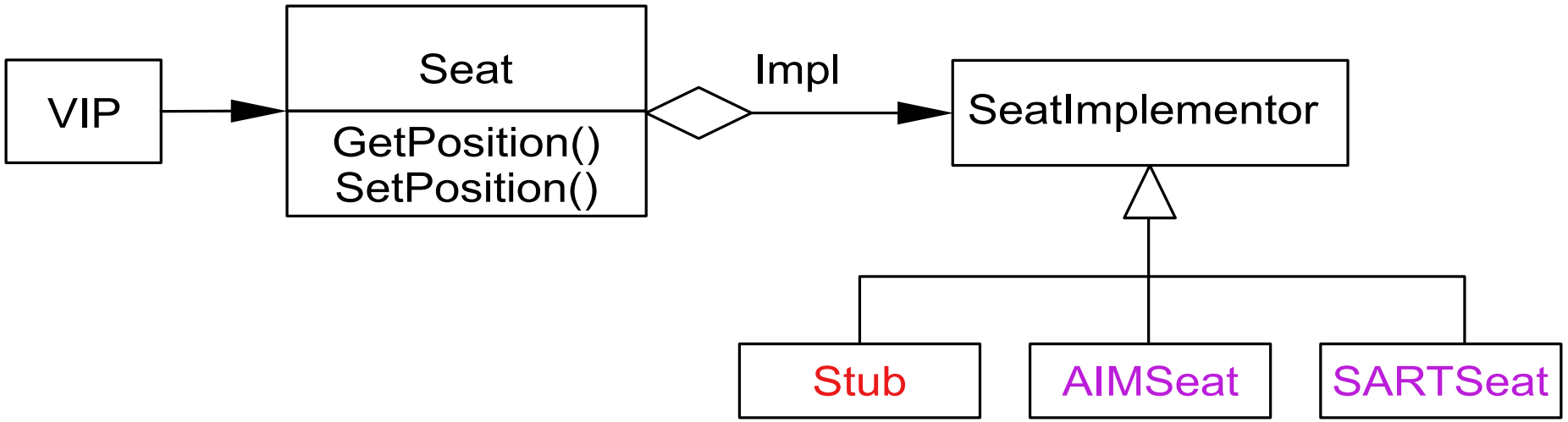


Using the bridge pattern during testing

- Interface to a component that is incomplete, not yet known or unavailable during testing
- Example: The VIP subsystem can adjust the seat position to the preference of the driver

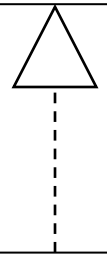
The seat is not yet implemented, so we are using a Stub

Two seat simulators become available: AIM and SART



Seat implementations

```
public interface Seat {
  public int GetPosition();
  public void SetPotision(int newPos);
}
```

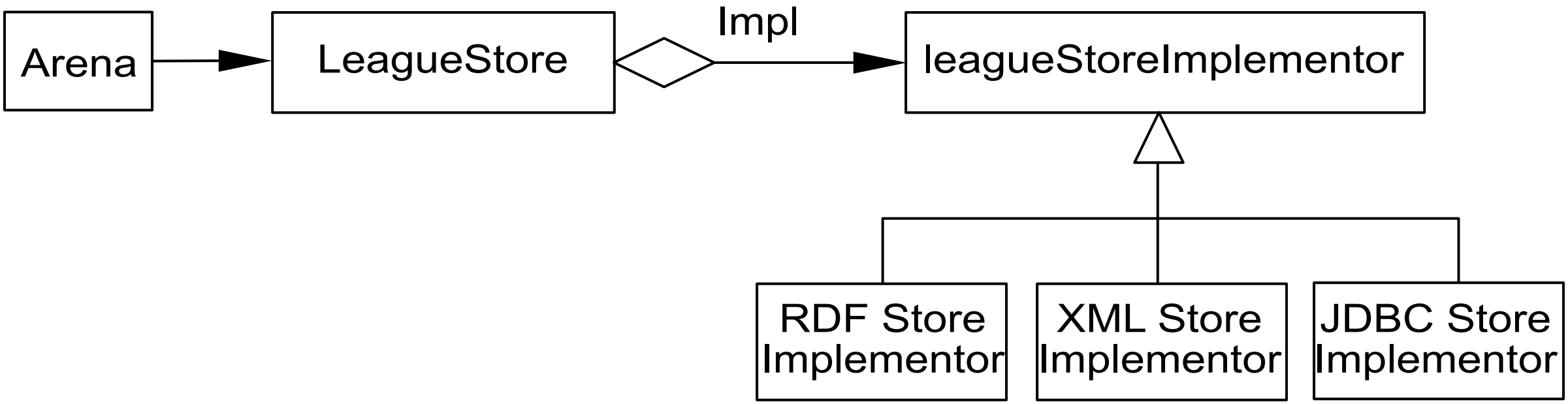


```
public class Stub implements Seat {
  public int GetPosition() {
    return 5.0;
  }
  // ...
}
```

```
public class AimSeat implements SeatImpl {
  public int GetPosition() {
    // call to the AIM simulator
  }
}
```

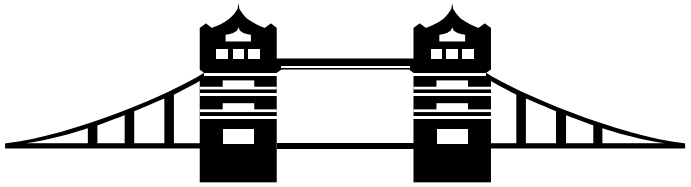
```
public class SARTSeat implements SeatImpl {
  public int GetPosition() {
    // call to the SART simulator
  }
}
```

Another use of the bridge pattern: supporting multiple database vendors



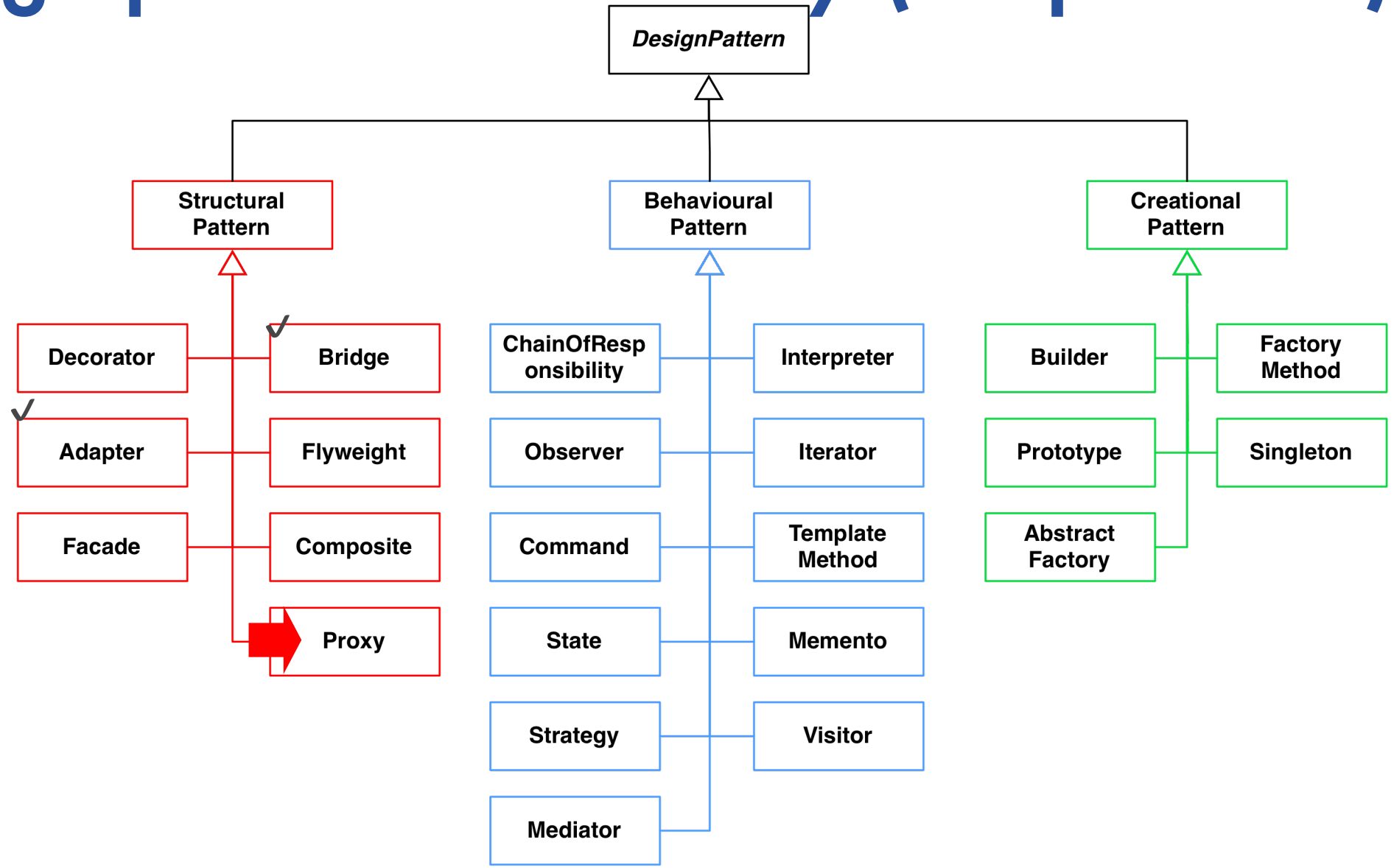
adapter vs bridge

- **Similarities**:
  - Both hide the details of the underlying implementation
- The **adapter** pattern aims at making **unrelated** components work together
  - Applied to systems that are already designed (reengineering, interface engineering projects)
  - "Inheritance followed by delegation"
- A **bridge** is used **up-front in a design** to let abstractions and implementations vary independently
  - Green field engineering of an "**extensible** system"
  - New "beasts" can be added to the "zoo" ("application or solution domain", even if these are not known, when the system is developed
  - "Delegation followed by inheritance"



Is this a bridge or an adapter?

Design patterns taxonomy (23 patterns)

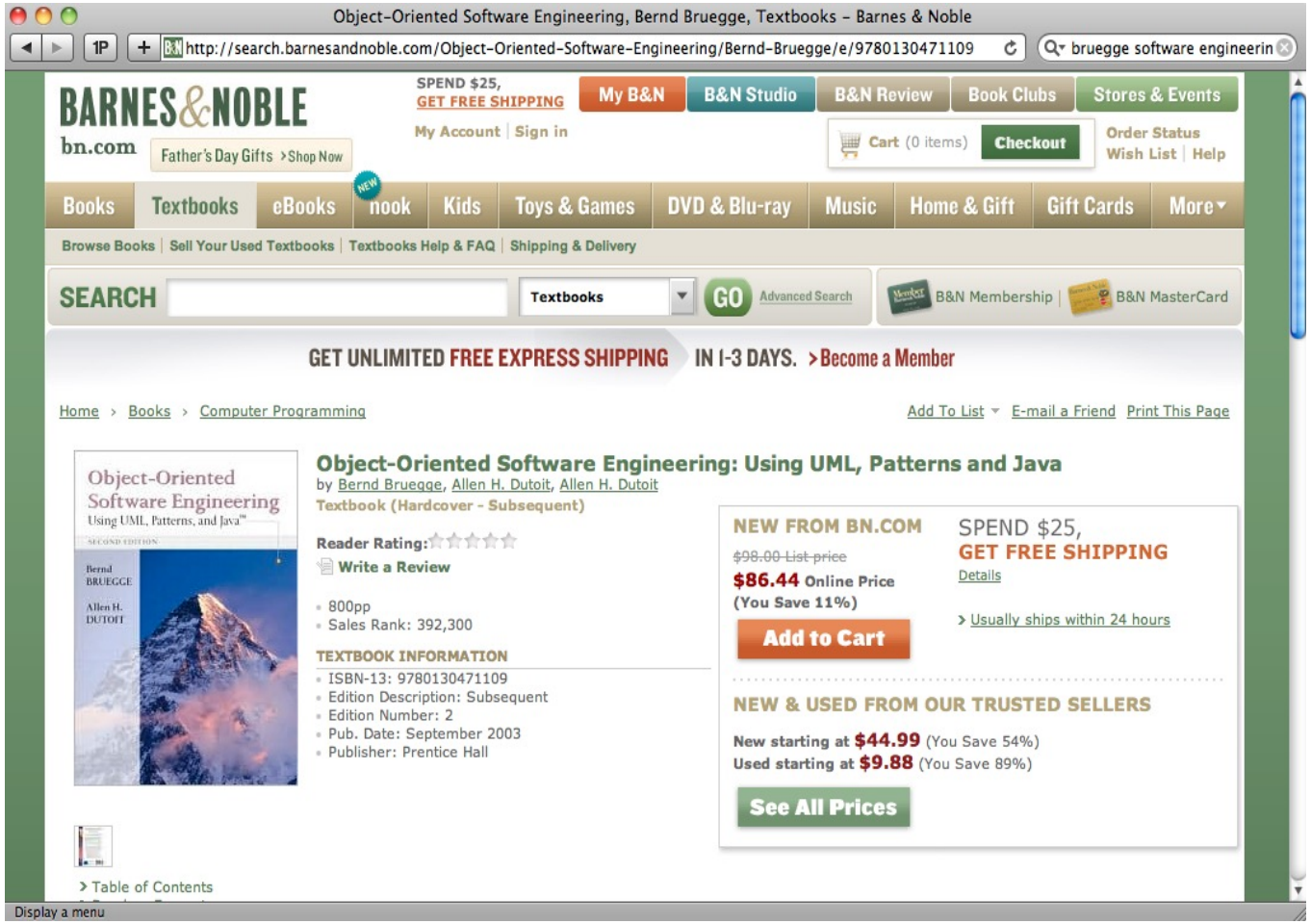


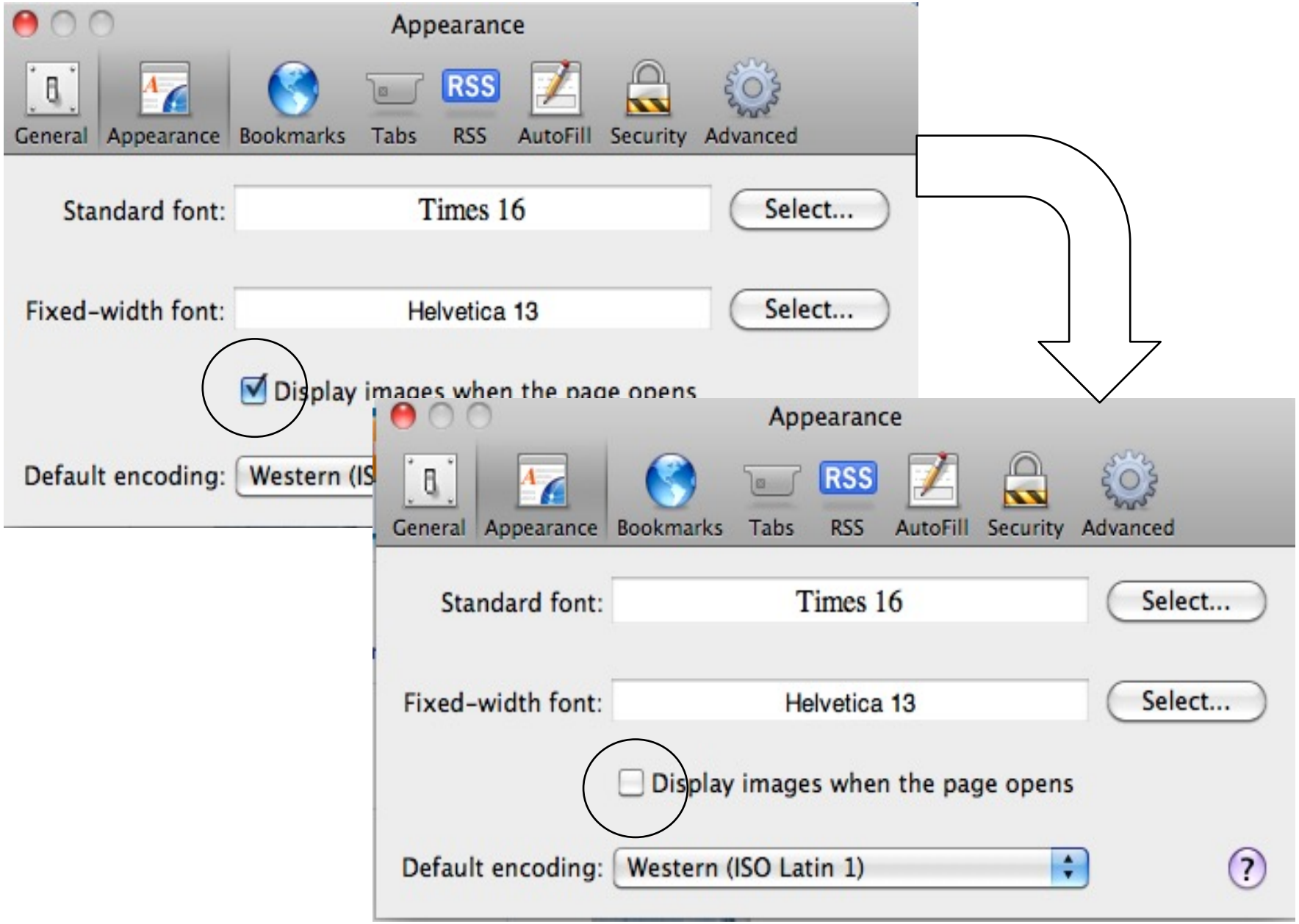
Outline of the talk

1. Taxonomy of Design Patterns
2. Adapter Pattern
3. Bridge Pattern
4. **Proxy Pattern**
5. Composite Pattern

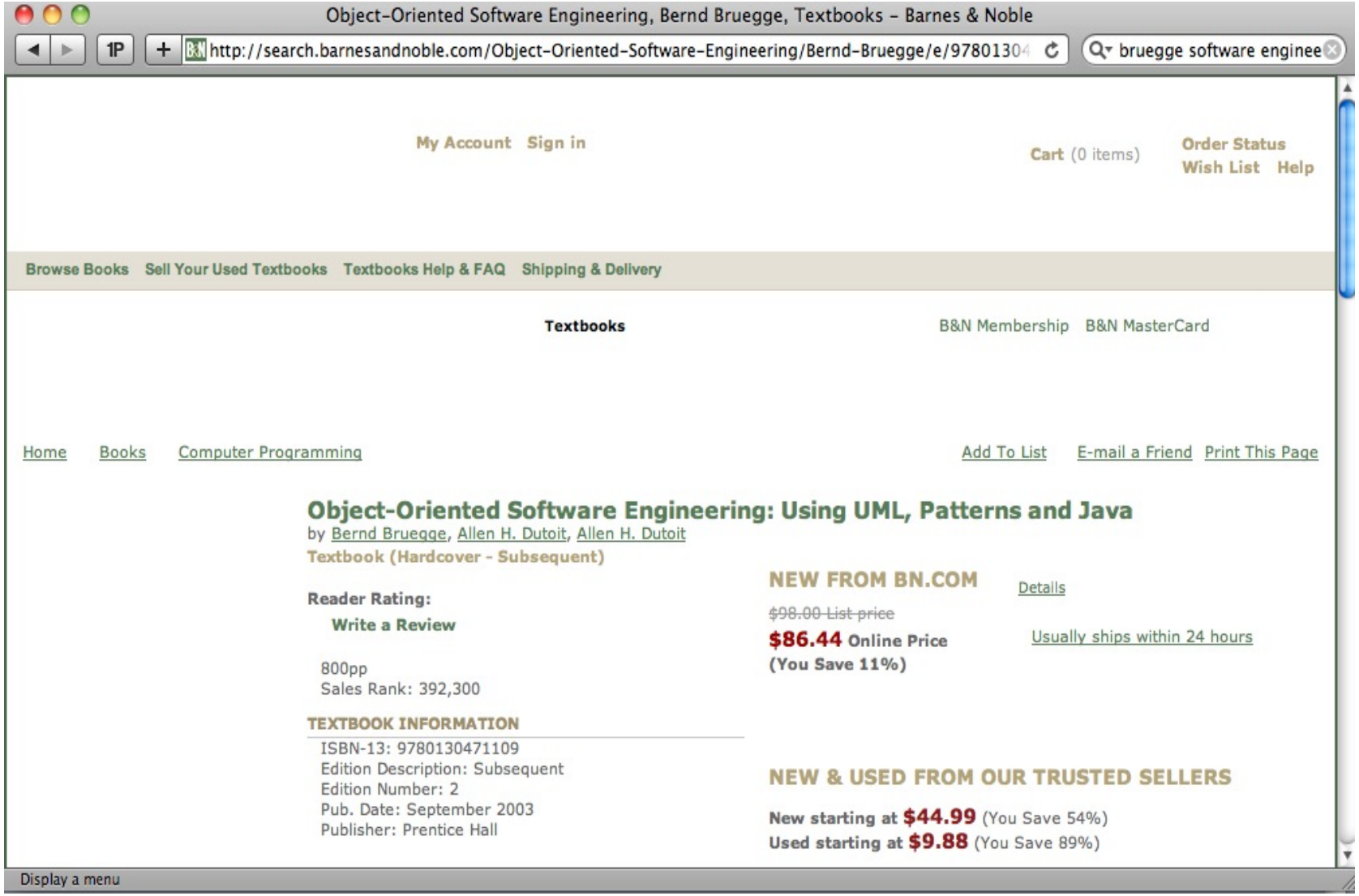
Proxy pattern: motivation

- It is 11:30
- 3 days before the EMSE final exam
- I sit in front of my laptop
- I need to buy Bruegge's book
- I get about 1 kb/sec
- I have 5 minutes before I am meeting my study group
- What I do?
  - Disable the display of images



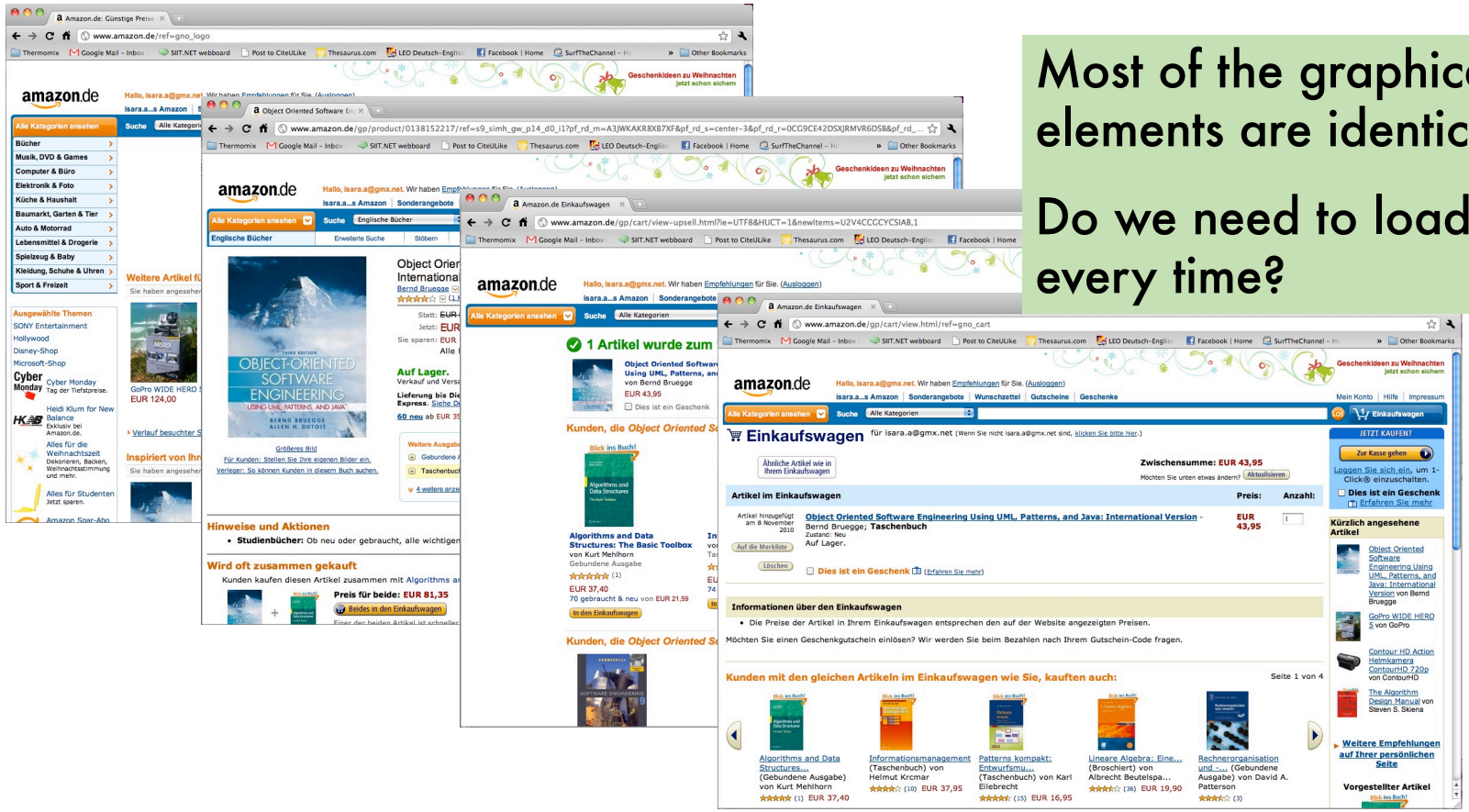

Appearance
General
Appearance
Bookmarks
Tabs
RSS
AutoFill
Security
Advanced
Standard font:
Times 16
Select...
Fixed-width font:
Helvetica 13
Select...
Display images when the page opens
Default encoding:
Western (ISO Latin 1)

Example: web page with dummy images



Another situation: caching

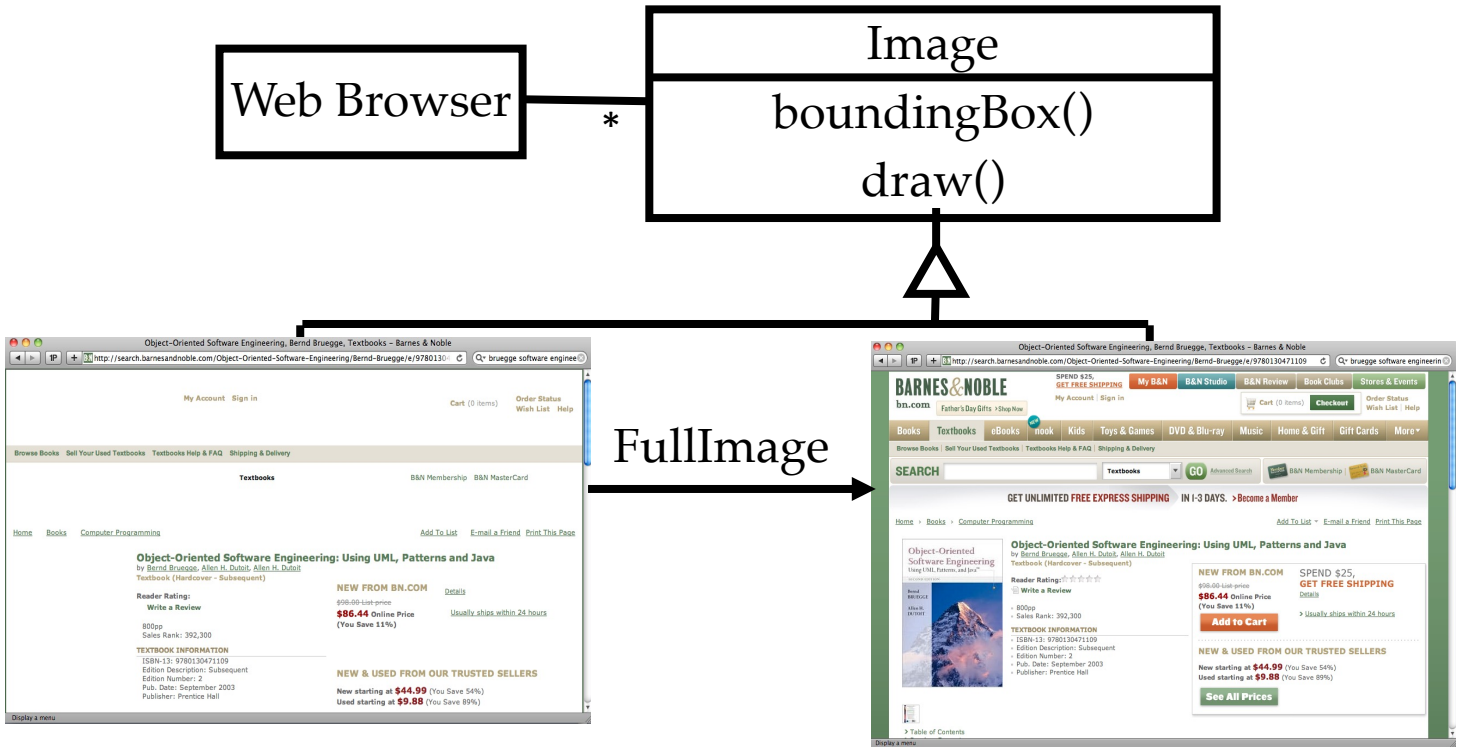
- Sometimes users access the same data repeatedly
- For example, web pages with common elements:



Most of the graphical elements are identical

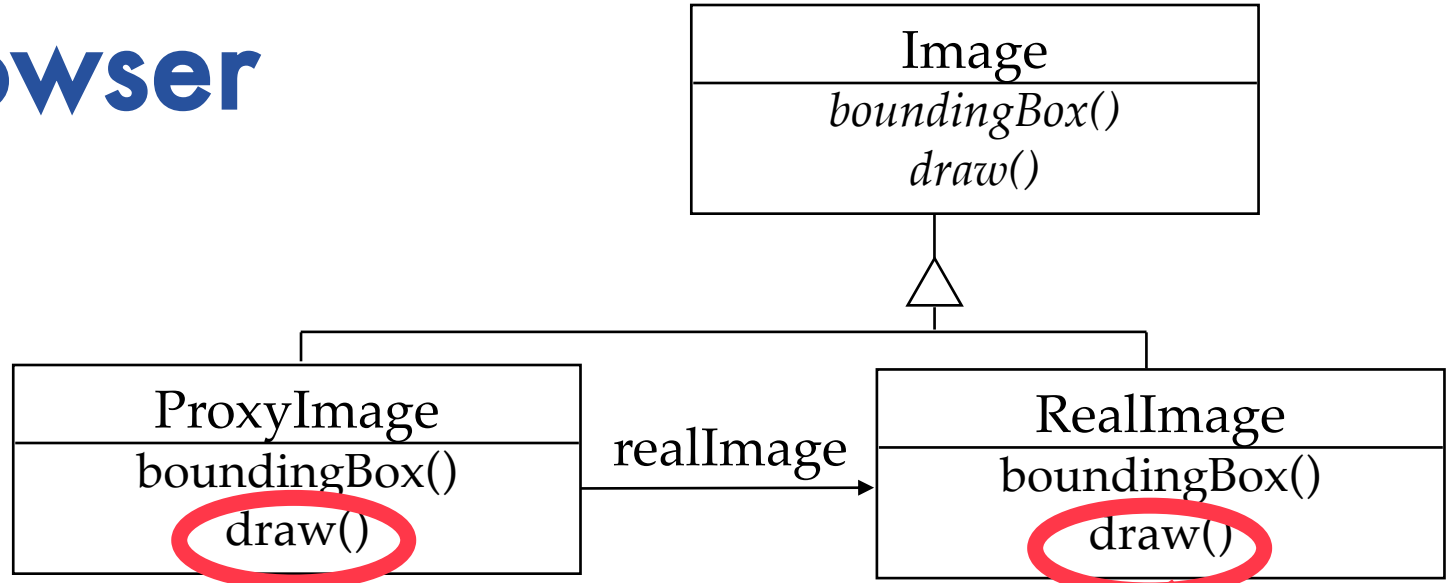
Do we need to load them every time?

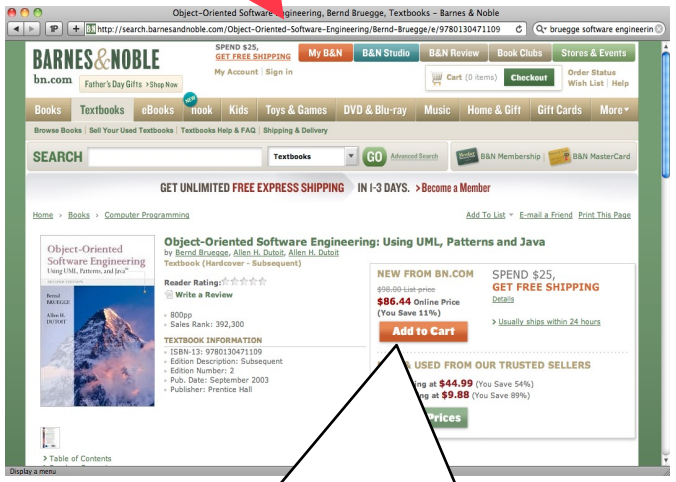
Let's model this!



- ProxyImage and RealImage are both subclasses of Image
- The draw() method is implemented differently in ProxyImage and in RealImage
  - The draw() operation in ProxyImage is executed first. Depending on the browser preferences, it then calls draw() in RealImage
- The Web Browser cannot tell, whether ProxyImage or RealImage is accessed

Web browser example



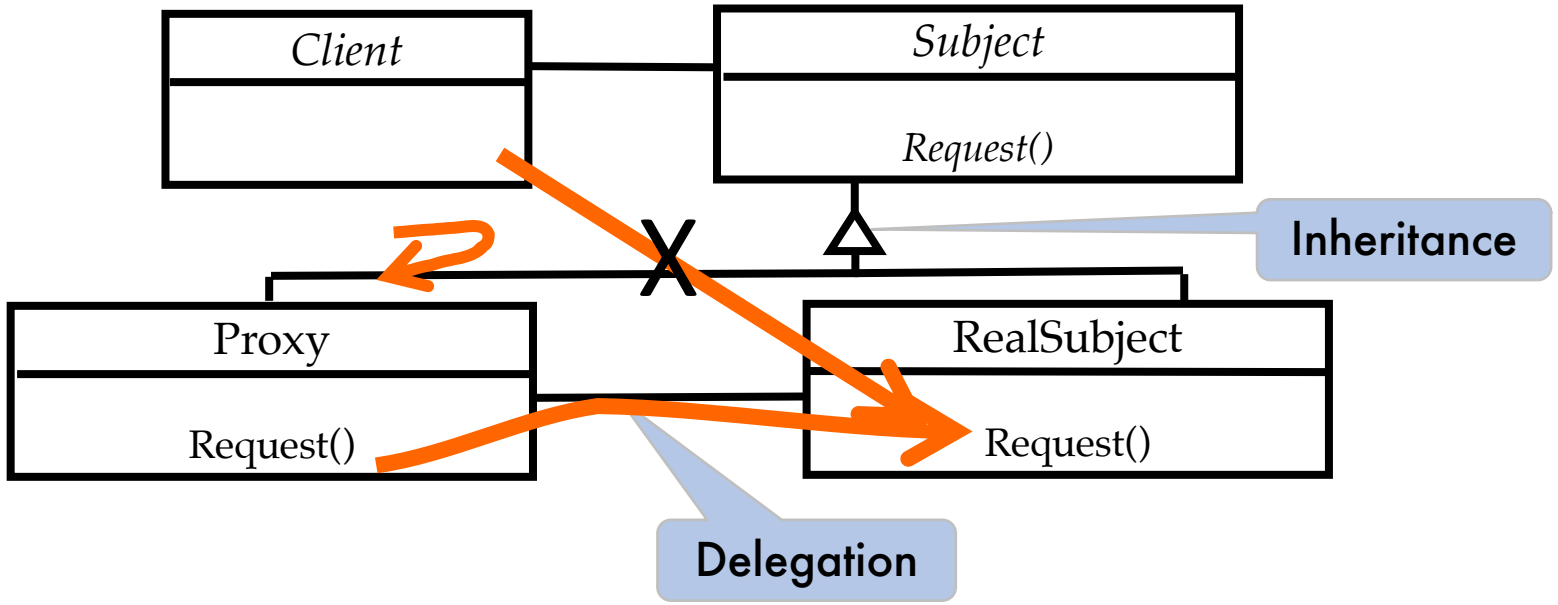


- ProxyImage draws a placeholder
- RealImage draws the image

Small Problem:
Add to Cart is an image ☹

Pro

The Proxy pattern looks so similar to other patterns that use Inheritance and delegation. What is the difference?



- Proxy and RealSubject are subclasses of Subject
- Request() in Proxy calls Request() in RealSubject
- Client never calls Request() in RealSubject

- Client always calls Request() in an instance of type Proxy, which might call Request in RealSubject.
- RealSubject ist also subclass of the abstract class Subject

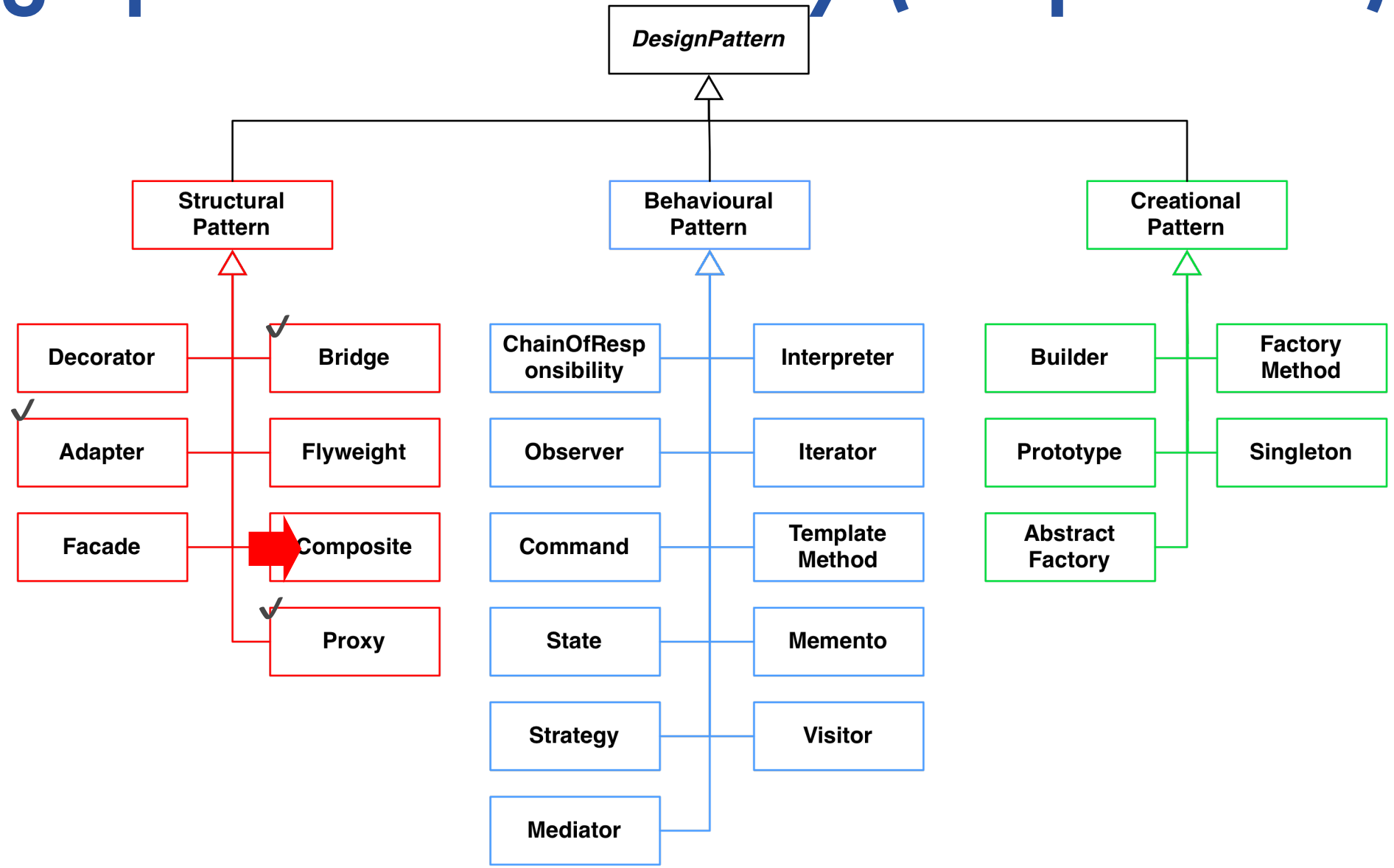
Applicability of the proxy pattern

- What is particularly expensive in object-oriented systems?
  - Object Creation
  - Object Initialization
- Defer object creation and object initialization to the time you need the object
- Proxy pattern:
  - Reduces the cost of accessing objects
    - Creates the real object only if the user asks for it
  - Uses another object ("the proxy") that acts as a stand-in for the real object
  - Provides location transparency

Scenarios where a proxy is appropriate

- Caching (Remote Proxy)
  - The proxy object is a local representative for an object in a different address space (Caching of information)
  - Good if information does not change too often
  - If the information changes, the cache needs to be flushed
- Substitute (Virtual Proxy)
  - The object is quite expensive to create or to download. The Proxy object can act as a stand-in
  - Good for information that is not immediately accessed
  - Good for objects that are not visible (not in line of sight, far away)
  - Example: Google Maps, Fog of War
- Access Control (Protection Proxy)
  - The proxy object provides access control to the real object
  - Good when different objects should have different access and viewing rights for the same document
  - Example: Grade information for a student shared by administrators, teachers and students.

Design patterns taxonomy (23 patterns)

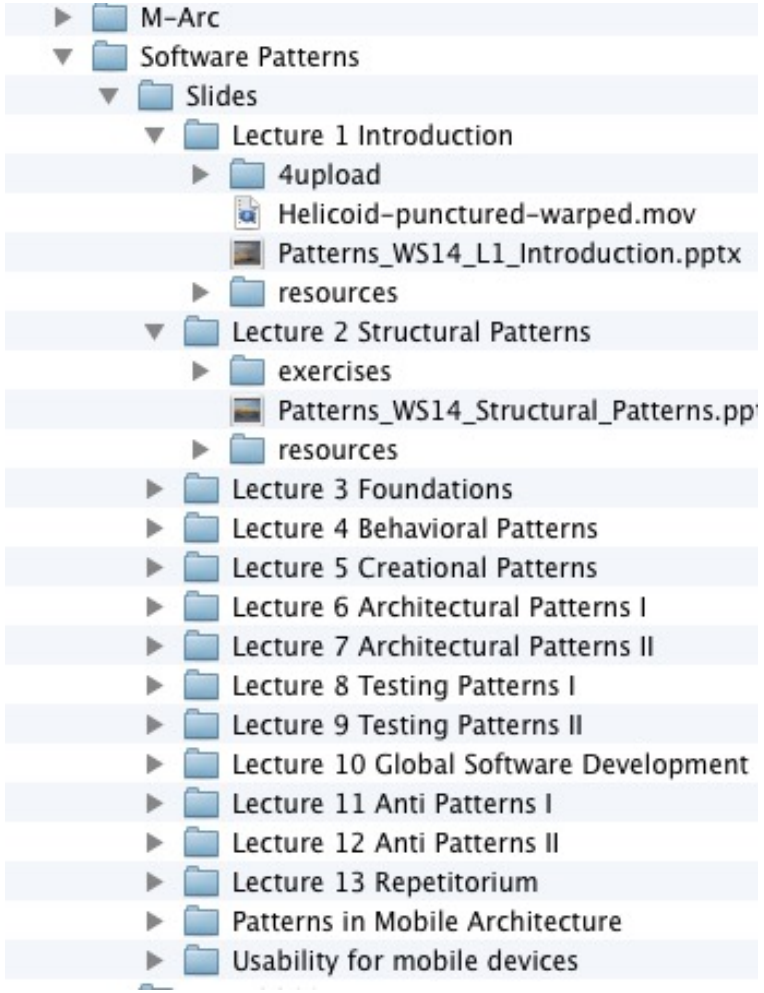


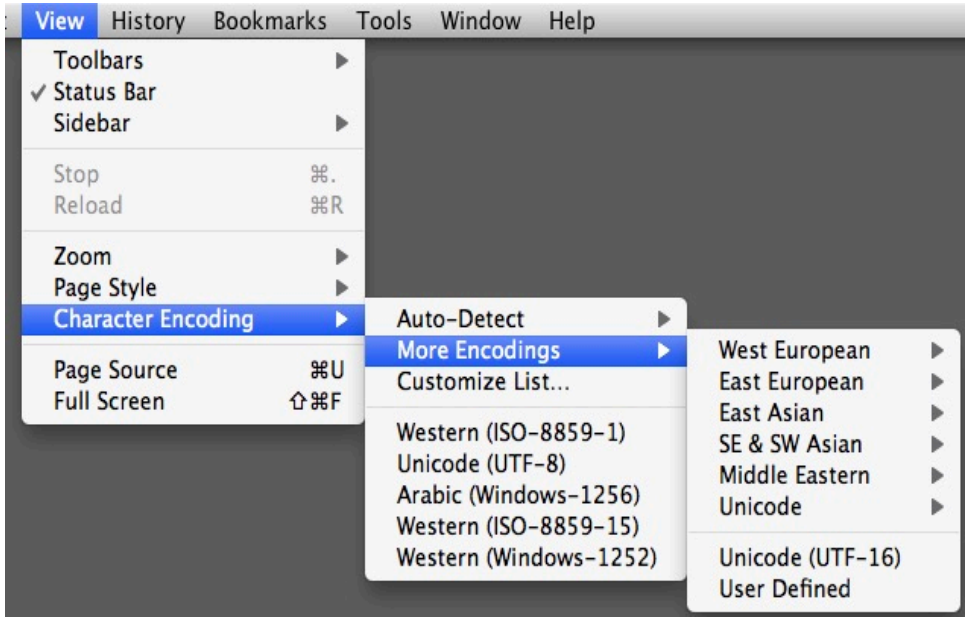
Outline of the talk

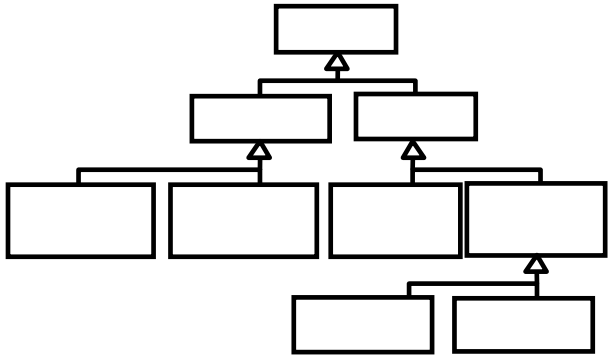
1. Taxonomy of Design Patterns
2. Adapter Pattern
3. Bridge Pattern
4. Proxy Pattern
5. **Composite Pattern**

Problem: manage dynamic structures

- Tree structures representing **part-whole hierarchies** with arbitrary **depth** and **width** can be used in the solution of many problems

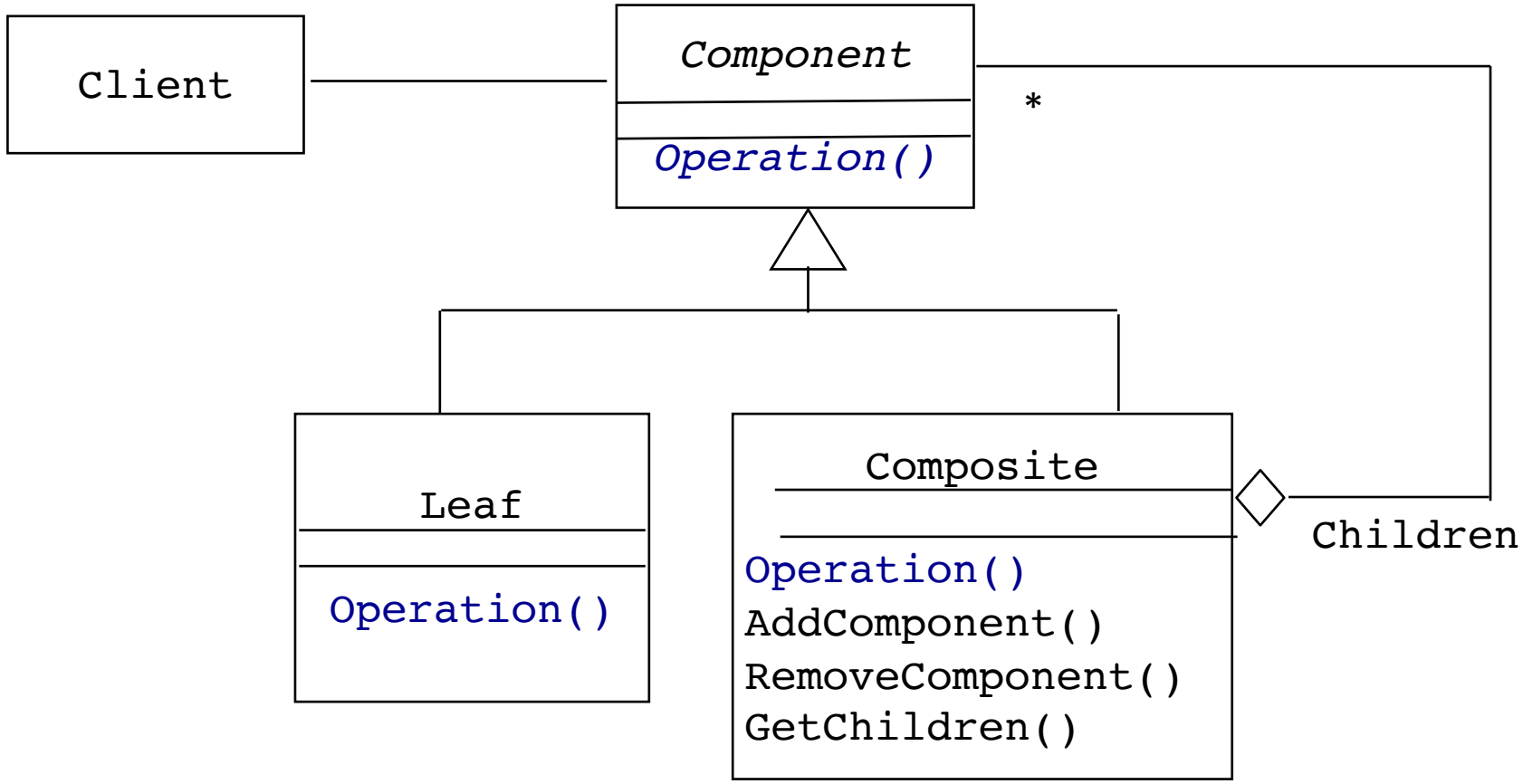






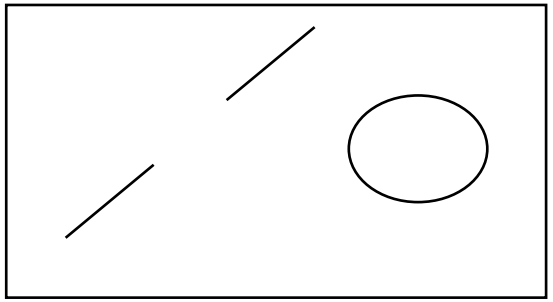
Solution: composite pattern (extract component)

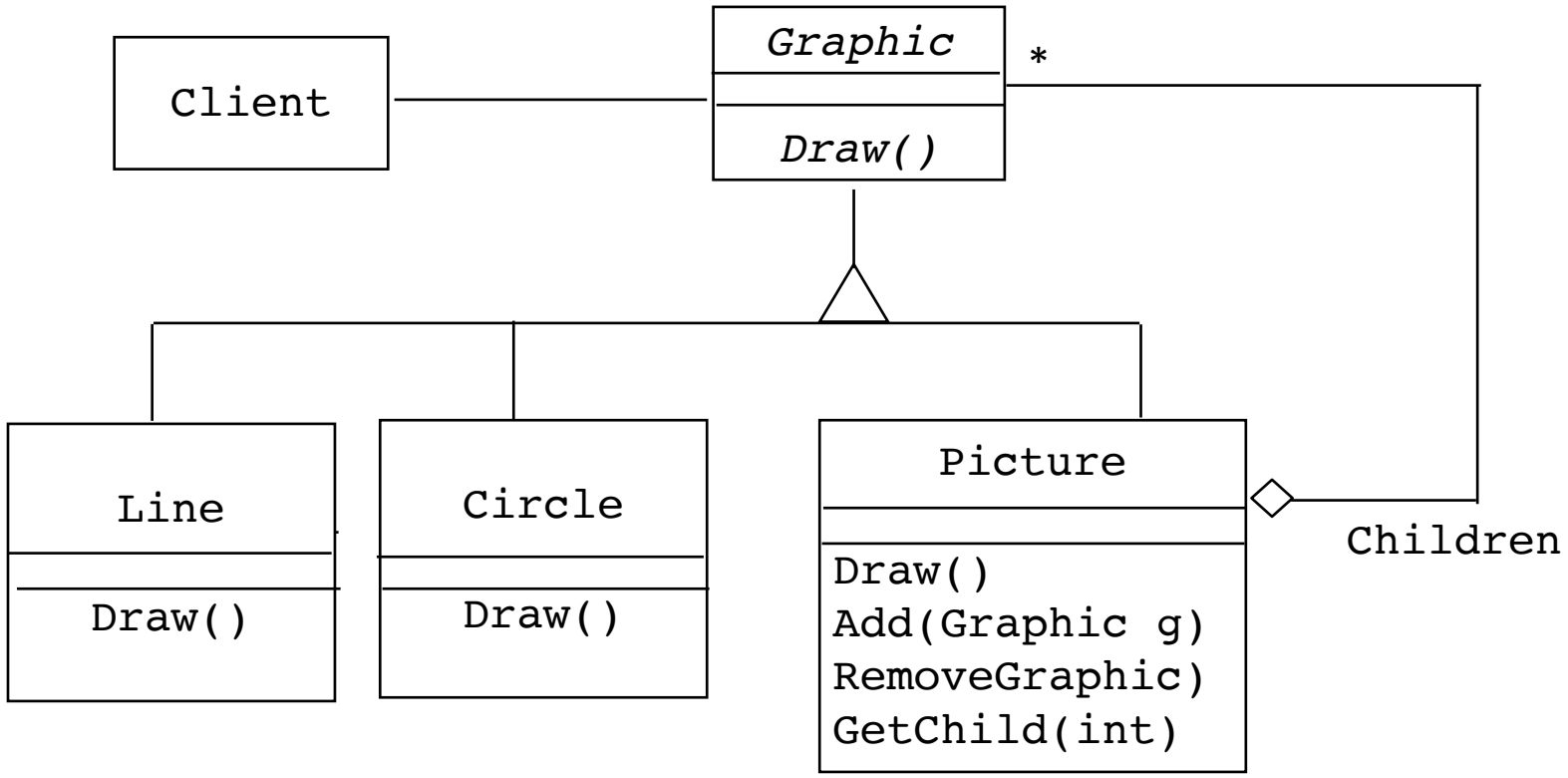
- Let a client treat individual objects and compositions of these objects uniformly



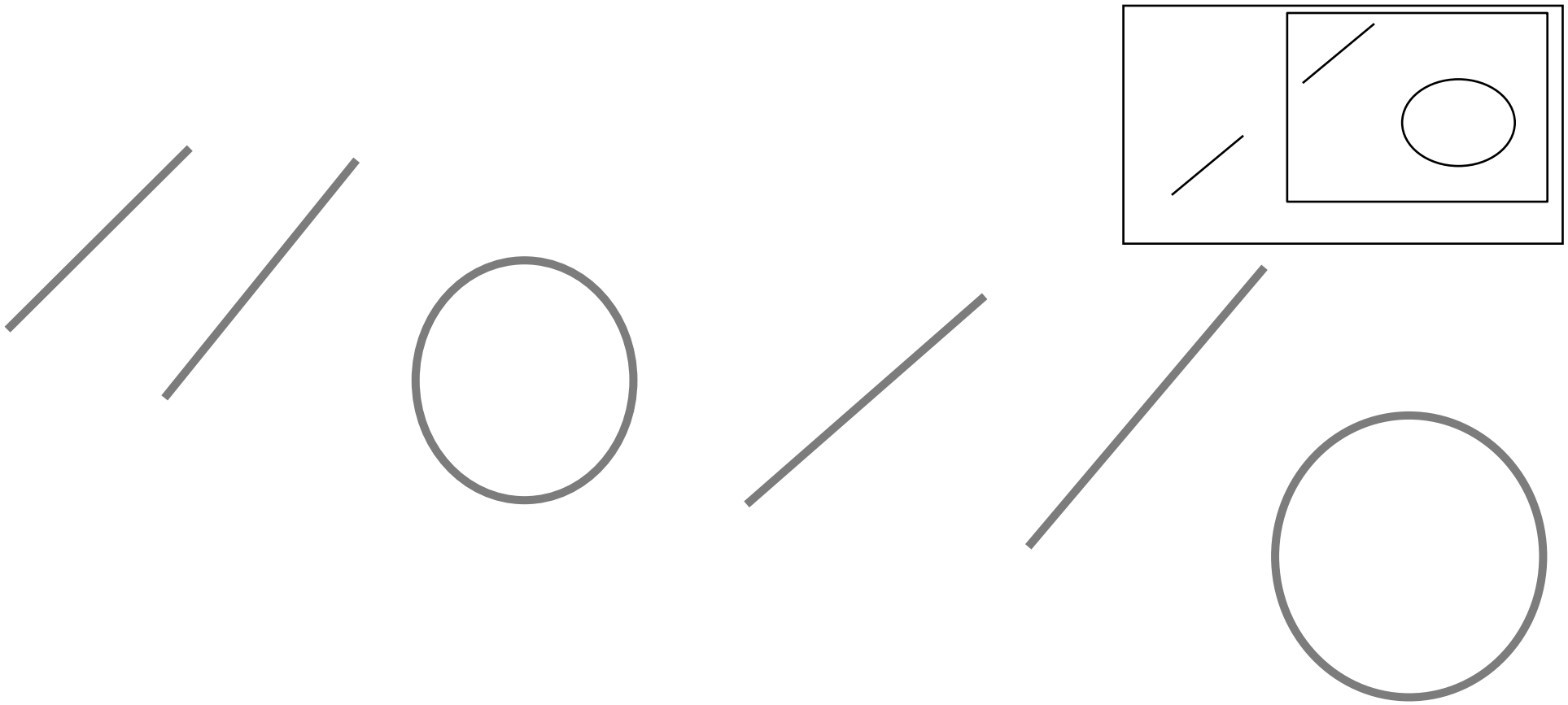
Composite pattern in graphic applications

- The Graphic class represents both primitives (Line, Circle) and containers (Picture).

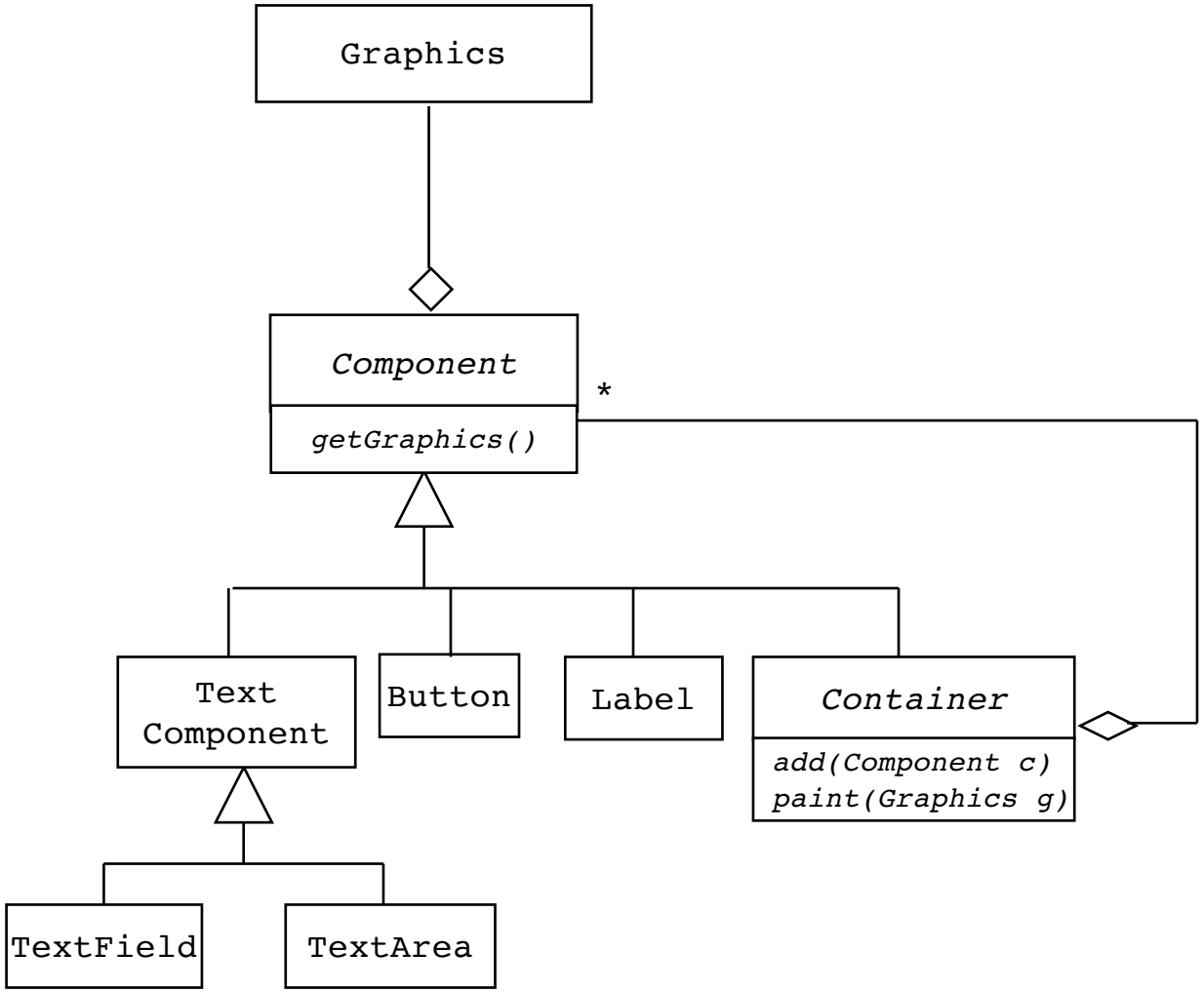




Example: grouping graphic

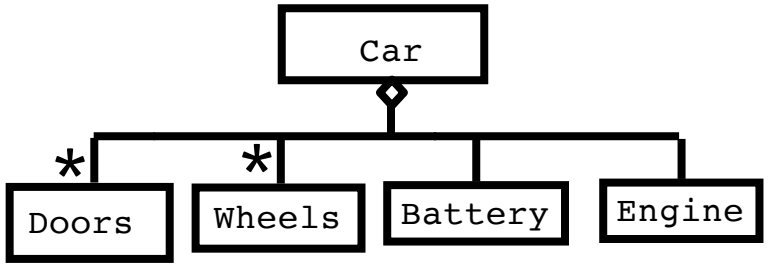


The java's AWT library can be modeled with the composite pattern



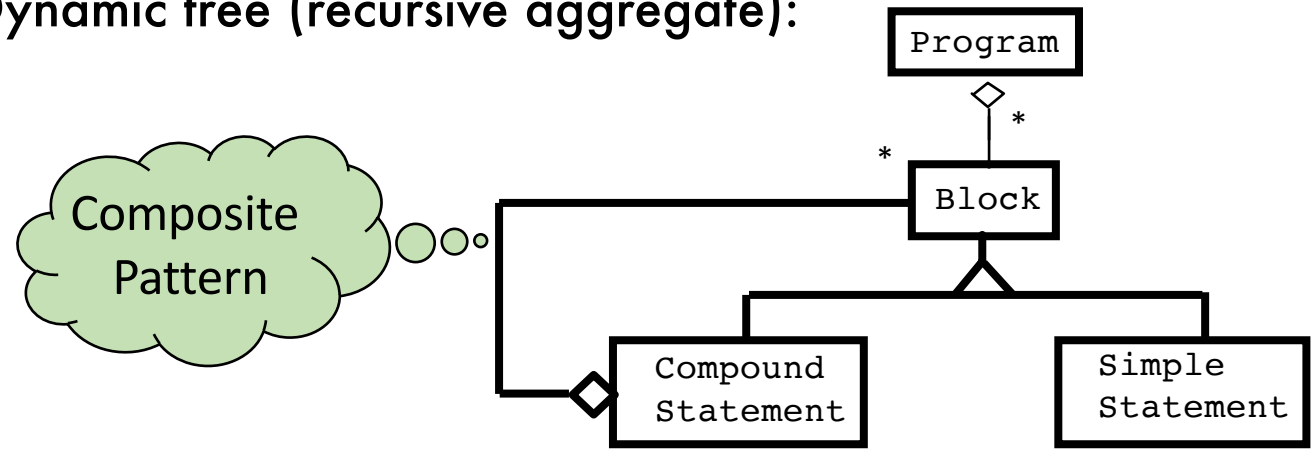
Composite patterns models dynamic aggregates

- Fixed Structure:



- Organization Chart (variable aggregate):

- Dynamic tree (recursive aggregate):



Design patterns taxonomy (23 patterns)