

Course "Softwaretechnik"

Book Chapter 8

Object Design: Reuse and Patterns

Lutz Prechelt, Bernd Bruegge, Allen H. Dutoit
Freie Universität Berlin, Institut für Informatik

- About "difficult" and "simple"
 - Get-15, Tic-Tac-Toe
- Patterns as simplification and reuse
- Design patterns
 - Composite
 - Adapter
 - Bridge
 - Facade

Wo sind wir?: Taxonomie "Die Welt der Softwaretechnik"

Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
 - Anforderungen (Problemraum)
 - **Entwurf (Lösungsraum)**
- Prozess (psycho-soziale P.)
 - Kognitive Beschränkungen
 - Mängel der Urteilskraft
 - Kommunikation, Koordination
 - Gruppendynamik
 - Verborgene Ziele
 - Fehler

Welt der Lösungsansätze:

- Technische Ansätze ("hart")
 - **Abstraktion**
 - **Wiederverwendung**
 - Automatisierung
- Methodische Ansätze ("weich")
 - Anforderungsermittlung
 - **Entwurf**
 - Qualitätssicherung
 - Projektmanagement

- Einsicht: Man sollte *vor* dem Kodieren über eine günstige Struktur der Software nachdenken
 - und diese als Koordinationsgrundlage schriftlich festhalten
- Prinzipien:
 - **Trennung von Belangen**
 - **Architektur**: Globale Struktur festlegen (Grobentwurf), insbes. für das Erreichen der nichtfunktionalen Anforderungen
 - **Modularisierung**: Trennung von Belangen durch Modularisierung, Kombination der Teile durch Schnittstellen (information hiding, Lokalität)
 - **Wiederverwendung**: Erfinde Architekturen und Entwurfsmuster nicht immer wieder neu
 - **Dokumentation**: Halte sowohl Schnittstellen als auch zu Grunde liegende Entwurfsentscheidungen und deren Begründungen fest

A game: Get-15

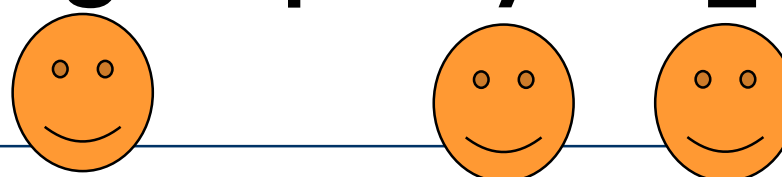
- Start with the nine numbers 1, 2, 3, 4, 5, 6, 7, 8, 9
- You and your opponent take alternate turns, each taking a number
 - Each number can be taken only once: If your opponent has selected a number, you cannot also take it
- The first person to have any three numbers that sum up to 15 wins the game
- Example:

You:

1 5 3 8

Opponent:

6 4 7 2



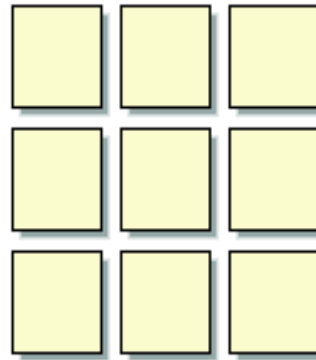
Opponent Wins!

Get-15 is "difficult"

- Hard to play
- The game is especially hard if you are not allowed to write anything down
- Why?
 - All the numbers need to be scanned to see if you have won/lost
 - It is hard to see what the opponent will take if you take a certain number
 - The choice of the number depends on all the previous numbers
 - Not easy to devise a simple strategy

Another game: Tic-Tac-Toe

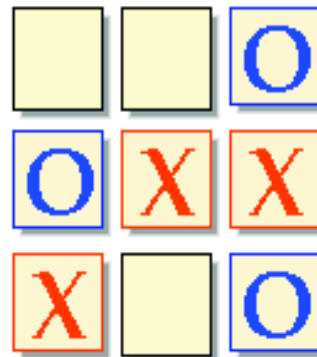
Players take turns signing a field with their mark.
 The first player to get three of his marks in a row, column,
 or diagonal wins.



YOU ARE 

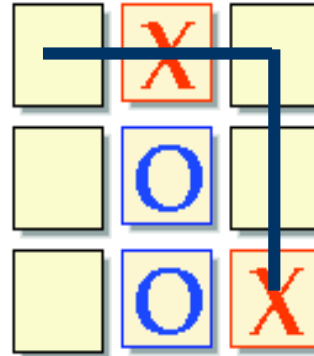
Source: <http://boulter.com/ttt/index.cgi>

A draw situation



YOU ARE 

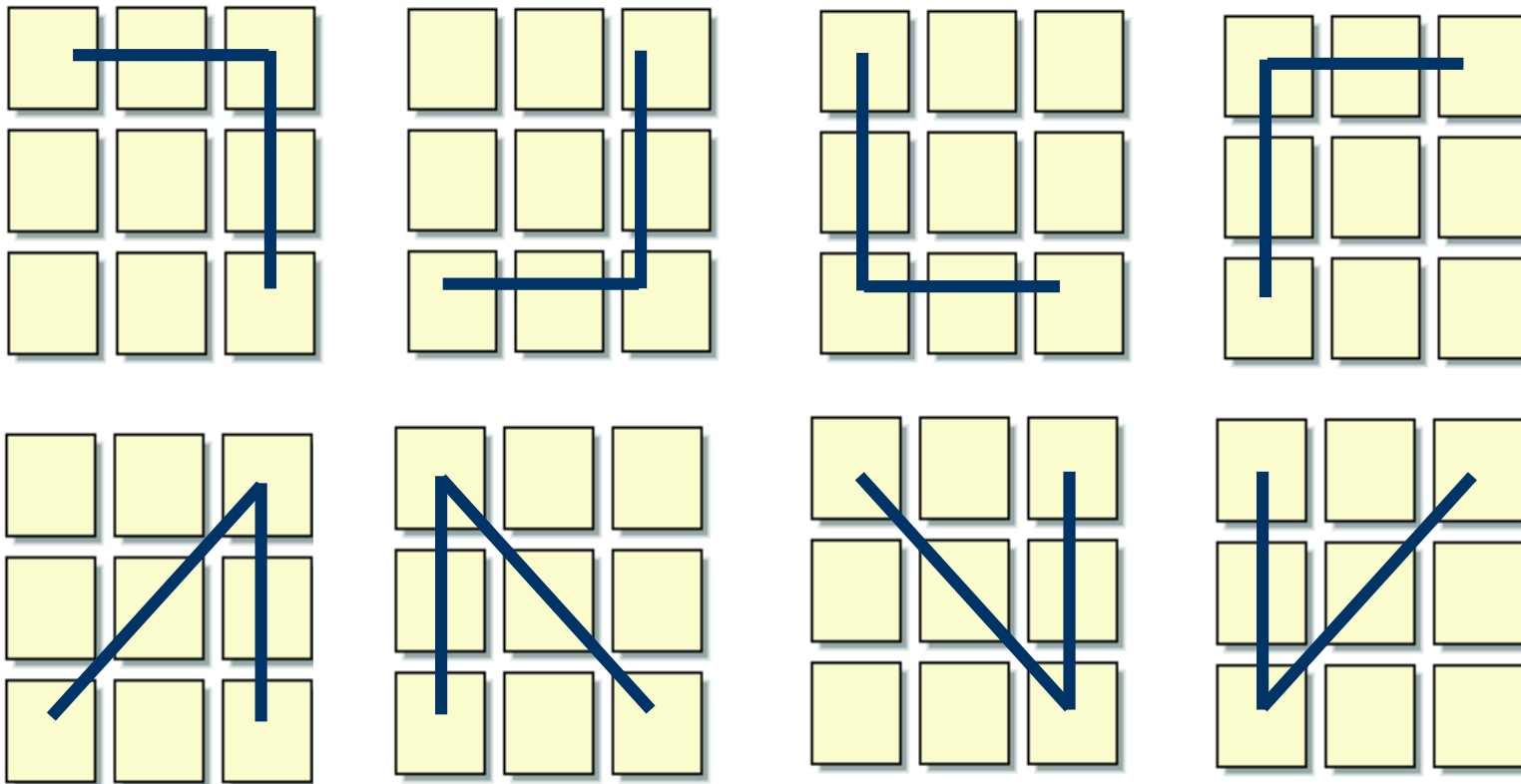
Strategy for determining a winning move



You win if

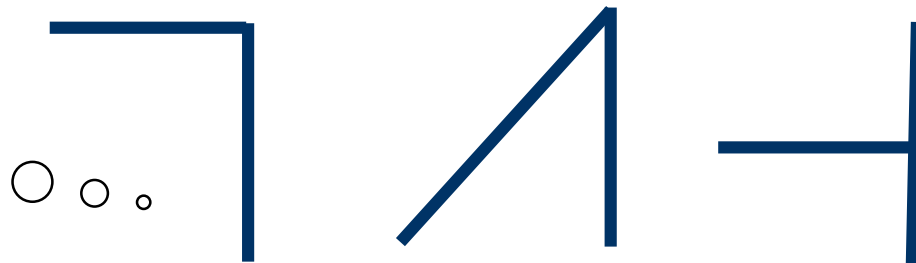
- you hold three fields on the two-segment line
- your opponent has none
- and yours include the corner

Winning situations for Tic-Tac-Toe



or likewise
with the
middle
row or
column

or likewise
with a
horizontal
and
diagonal



Tic-Tac-Toe is "Easy"

- Why? Reduction of complexity through patterns and symmetries
- Knowing the following **patterns**, the player can anticipate the opponents move:



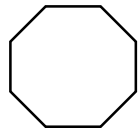
- The player needs to remember only these three patterns (and use symmetries) to deal with all different game situations
- **Such patterns are useful for problem solving**
 - **and even better than that:**

Get-15 and Tic-Tac-Toe are identical problems!

- Any three numbers that solve the Get-15 problem also solve tic-tac-toe
- Any tic-tac-toe solution is also a solution of Get-15
- To see the relationship between the two games, we simply arrange the 9 digits into the following pattern

8	1	6
3	5	7
4	9	2

Get-15 as Tic-Tac-Toe



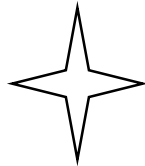
You:

1

5

3

8



Opponent:

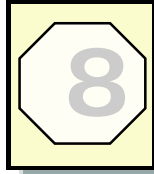
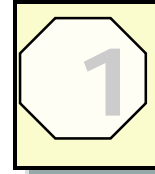
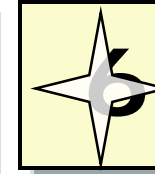

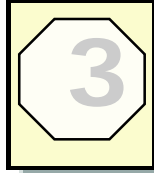
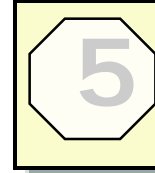
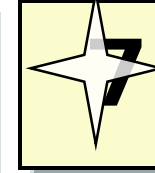

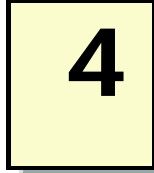
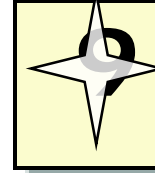
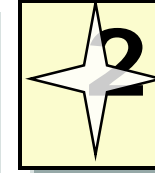

6

9

7

2

8	1	6
3	5	7
4	9	2

Why patterns are helpful

1. Patterns are abstractions

- Understanding a pattern reduces a number of elements to a single idea
- This saves mental resources and simplifies understanding
 - and communication

2. Patterns provide reuse

- If I know the patterns solution previously,
I do not have to invent my own solution: Reuse of ideas!

In the next two lectures we show how to use design patterns

Modeling heuristics

- Modeling must address our mental limitations:
 - Our short-term memory has only limited capacity (7 ± 2)
- Good models deal with this limitation, because they
 - reduce complexity
 - Turn complex tasks into easy ones (by good choice of representation)
 - Use symmetries or other regularities
 - Use helpful abstractions
 - "Obvious" taxonomies
 - Memory limitations are overcome with an appropriate representation ("natural model")
 - and therefore do not tax the mind
 - A good model requires only little mental effort to understand

Design patterns have these properties.

Outline of the lecture

- Design Patterns
 - Usefulness of design patterns
 - Design Pattern Categories
- Patterns covered in this lecture
 - **Composite**: Model dynamic aggregates
 - **Facade**: Interfacing to subsystems
 - **Adapter**: Interfacing to existing systems (legacy systems)
 - **Bridge**: Interfacing to existing and future systems
- Patterns covered in the next lecture
 - Abstract Factory
 - Builder
 - Command
 - Observer
 - Proxy
 - Strategy

Patterns help finding objects

- The possibly hardest problems in object-oriented system development are:
 - Identifying objects
 - Decomposing the system appropriately into objects
- 1. Requirements Analysis focuses on application domain:
 - Identify application objects
- 2. System Design addresses both, application and implementation domain:
 - Identify architecture
 - Partition into subsystems and modules
- 3. Object Design focuses on implementation domain:
 - Transform application objects into solution objects
 - Identify technical solution objects

Design patterns help with Object Design

Definition: Design Patterns

What are Design Patterns?

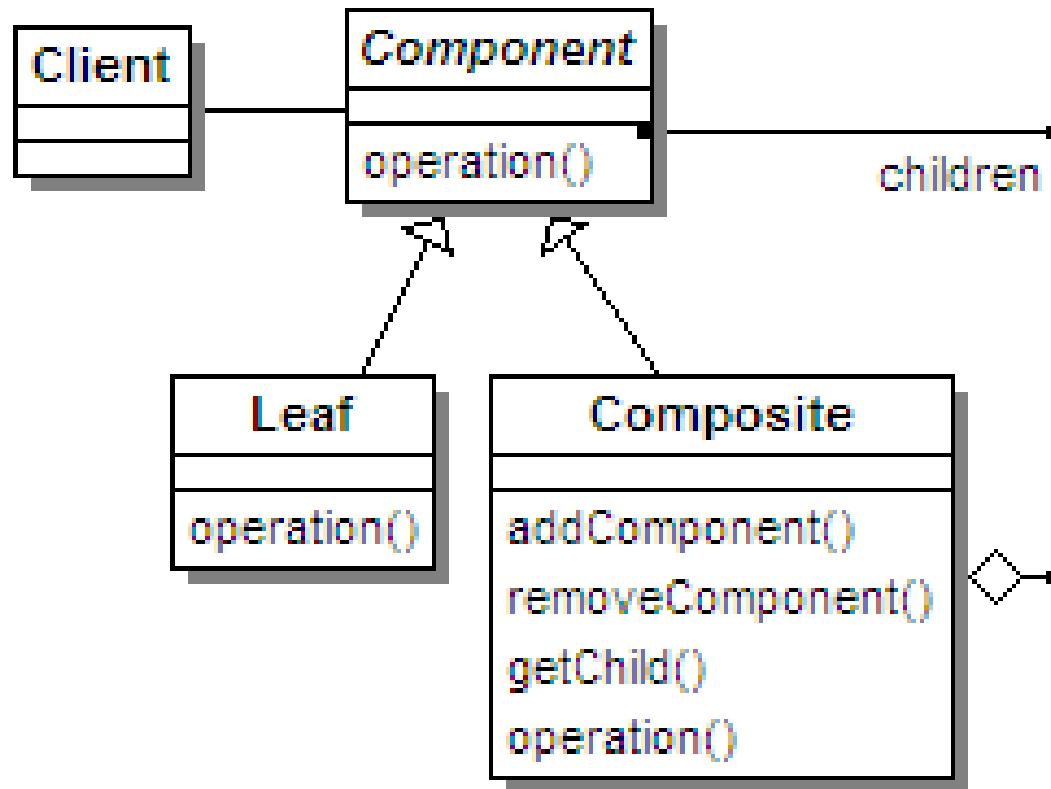
- A design pattern describes a **problem** which occurs over and over again in our environment
- Then it describes the **idea of a solution** to that problem
 - in such a way that you can use the pattern many times, without ever doing it the exact same way twice:
 - The solution idea will always be **adapted** to the specific context in which the pattern is being used

What is common between these two definitions?

- Definition Software System:
 - "A software system consists of parts which are either themselves systems (called subsystems) or individual classes"
- Definition Software Lifecycle:
 - "A software development process consists of steps which are either smaller processes (called activities) or elementary tasks"

The Composite Pattern

- Models tree structures that represent *part/whole hierarchies* with arbitrary depth and width.
- The Composite Pattern lets a client treat individual objects and *compositions* of these objects uniformly

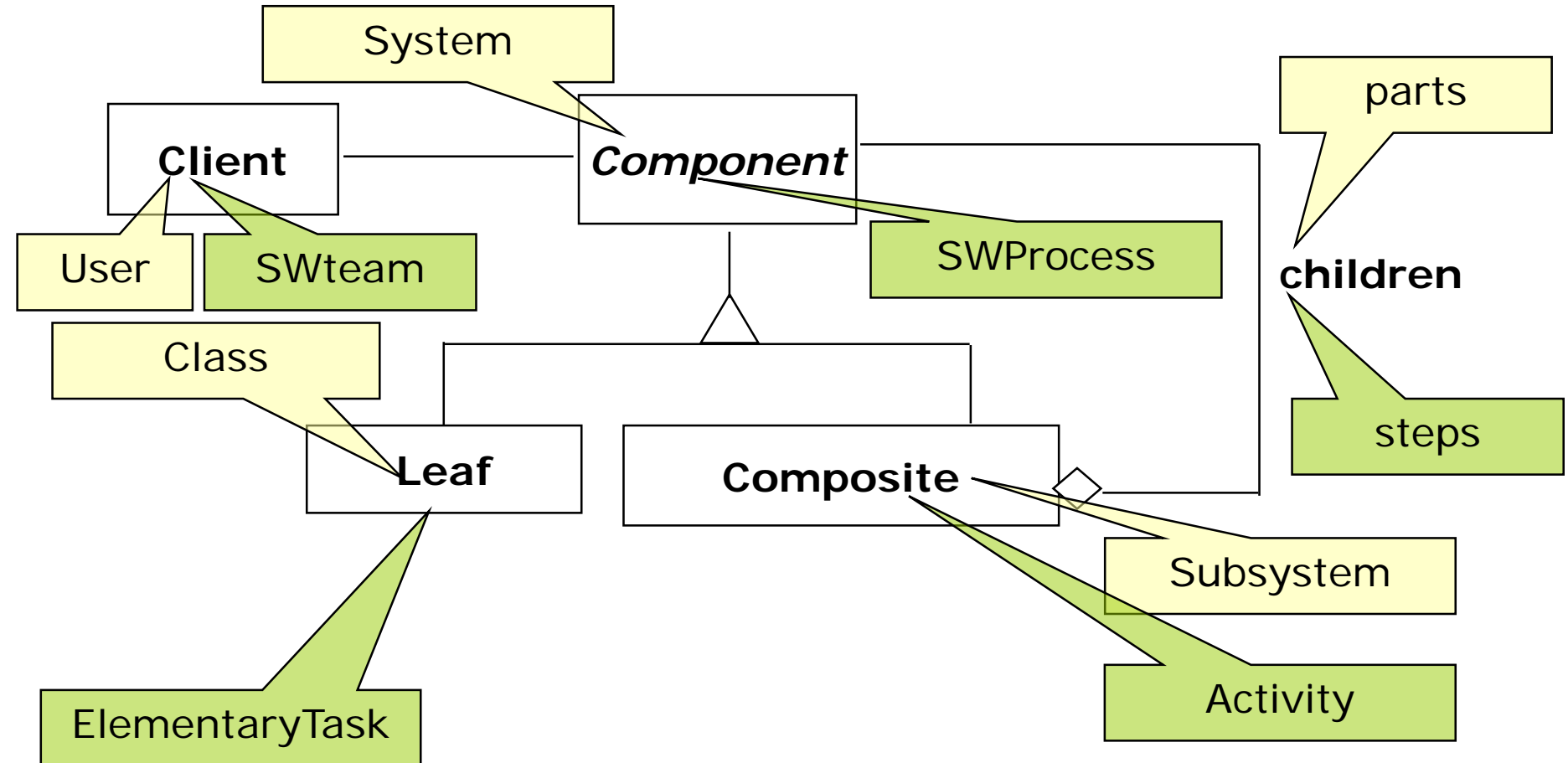


- Problem: Represent part/whole hierarchies so that
 1. they can have arbitrary depth and width
 2. can be created and modified dynamically
 3. composite parts can be handled just like elementary parts
- Solution idea:
 - Have a common superclass Component
 - Have two kinds of subclasses, one for elementary parts, one for composite parts
 - The composite part classes are containers holding Component objects
 - This realizes (1) and (2)
 - Operations common to all parts are defined in the Component class
 - This realizes (3)
- <http://c2.com/cgi/wiki?CompositePattern>

Two applications of the Composite Pattern

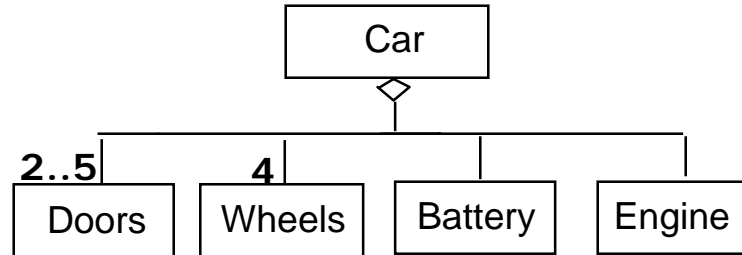
A software system consists of parts which are either themselves systems (called subsystems) or indiv. classes

A SW dev. process consists of steps which are either smaller processes (called activities) or elementary tasks

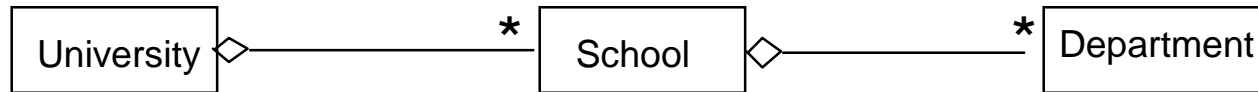


The Composite Patterns models dynamic aggregates

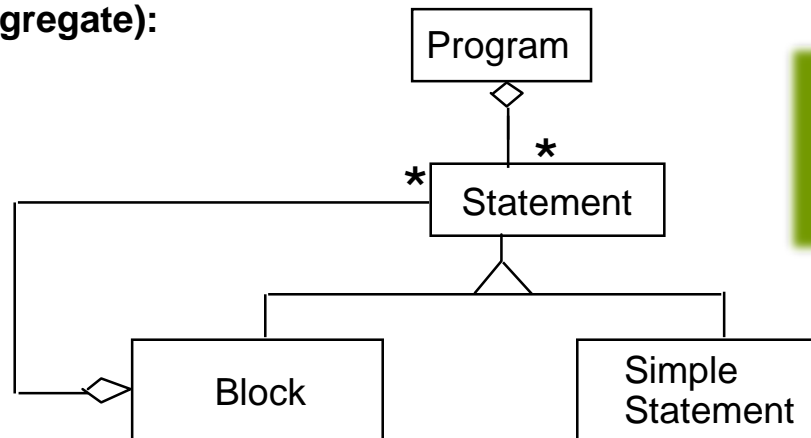
Fixed Structure:



Organization Chart (variable aggregate):



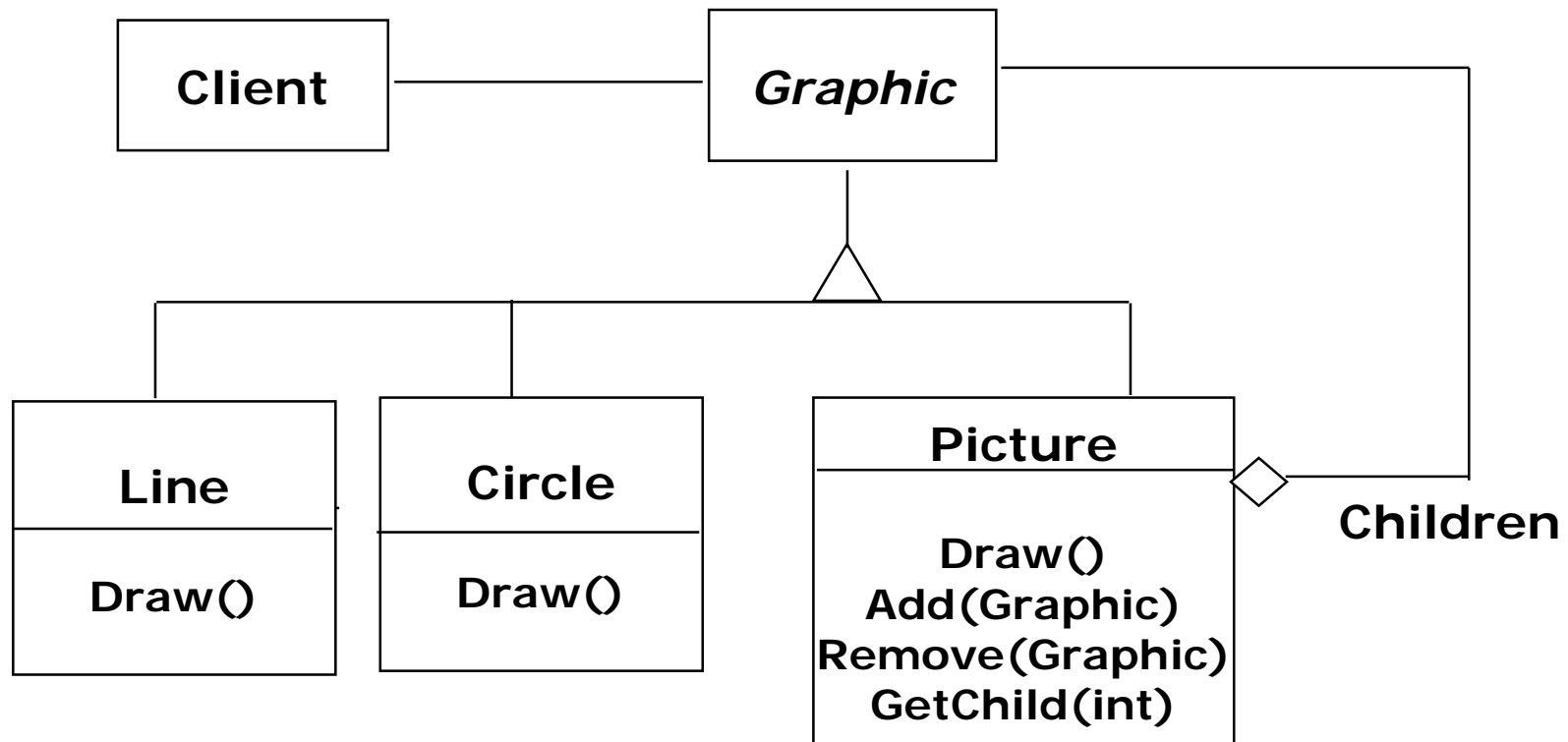
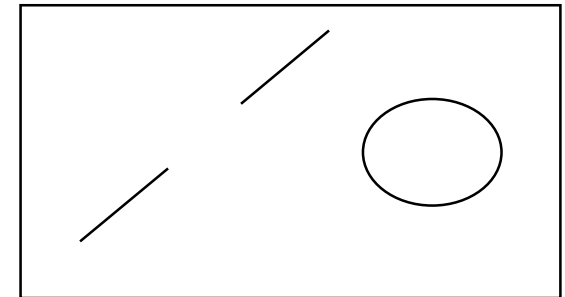
Dynamic tree (recursive aggregate):



Composite Pattern

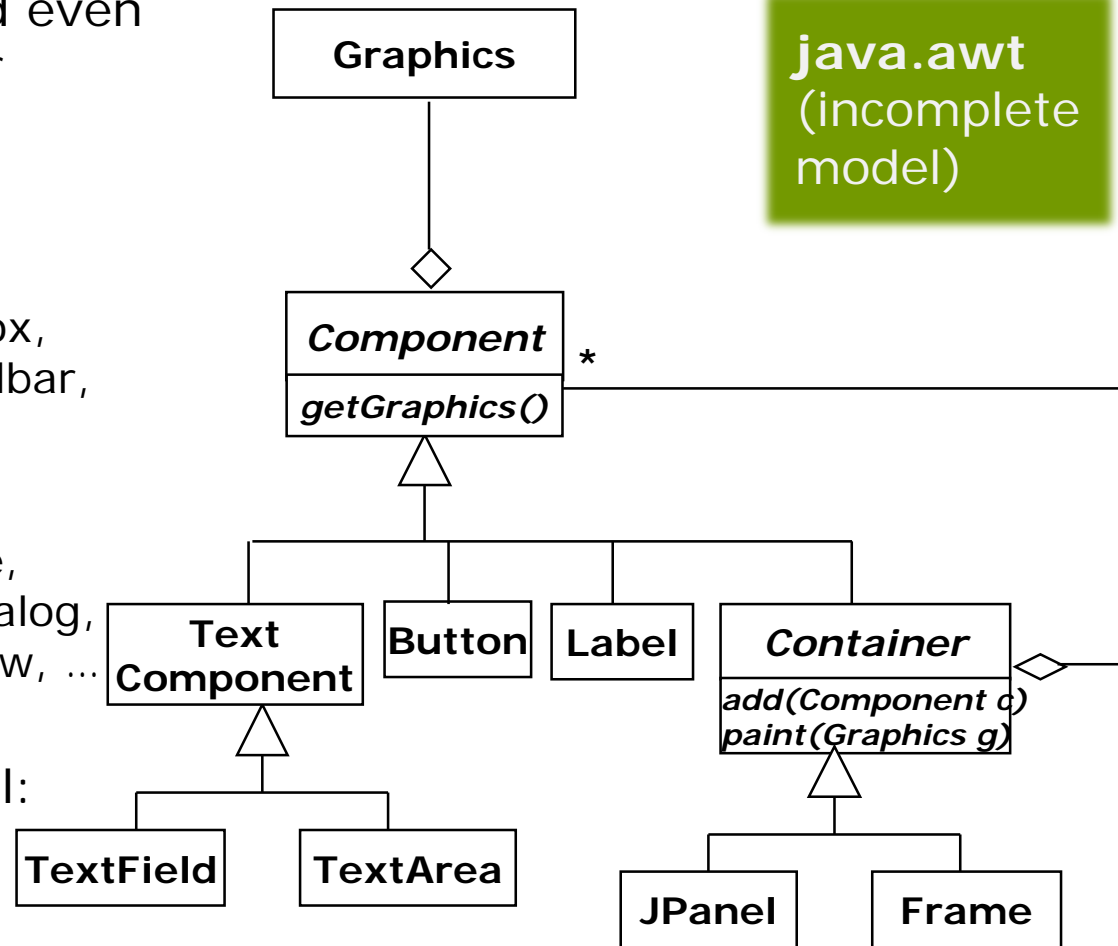
Graphic applications also use Composite Patterns

The *Graphic* Class
represents both primitives
(Line, Circle) and their
containers (Picture)



More variants: many primitives and many containers

- Some Composite structures have many primitives and even several kinds of container
- E.g. the basic Java GUI framework **java.awt**
- Primitives:
 - Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, TextArea, TextField
- Containers
 - Container, Dialog, Frame, CellRendererPane, FileDialog, Panel, ScrollPane, Window, ...
- This is important about design patterns in general:
Basic idea is fixed,
details vary!



java.awt
(incomplete
model)

Design Patterns

reduce the complexity of models

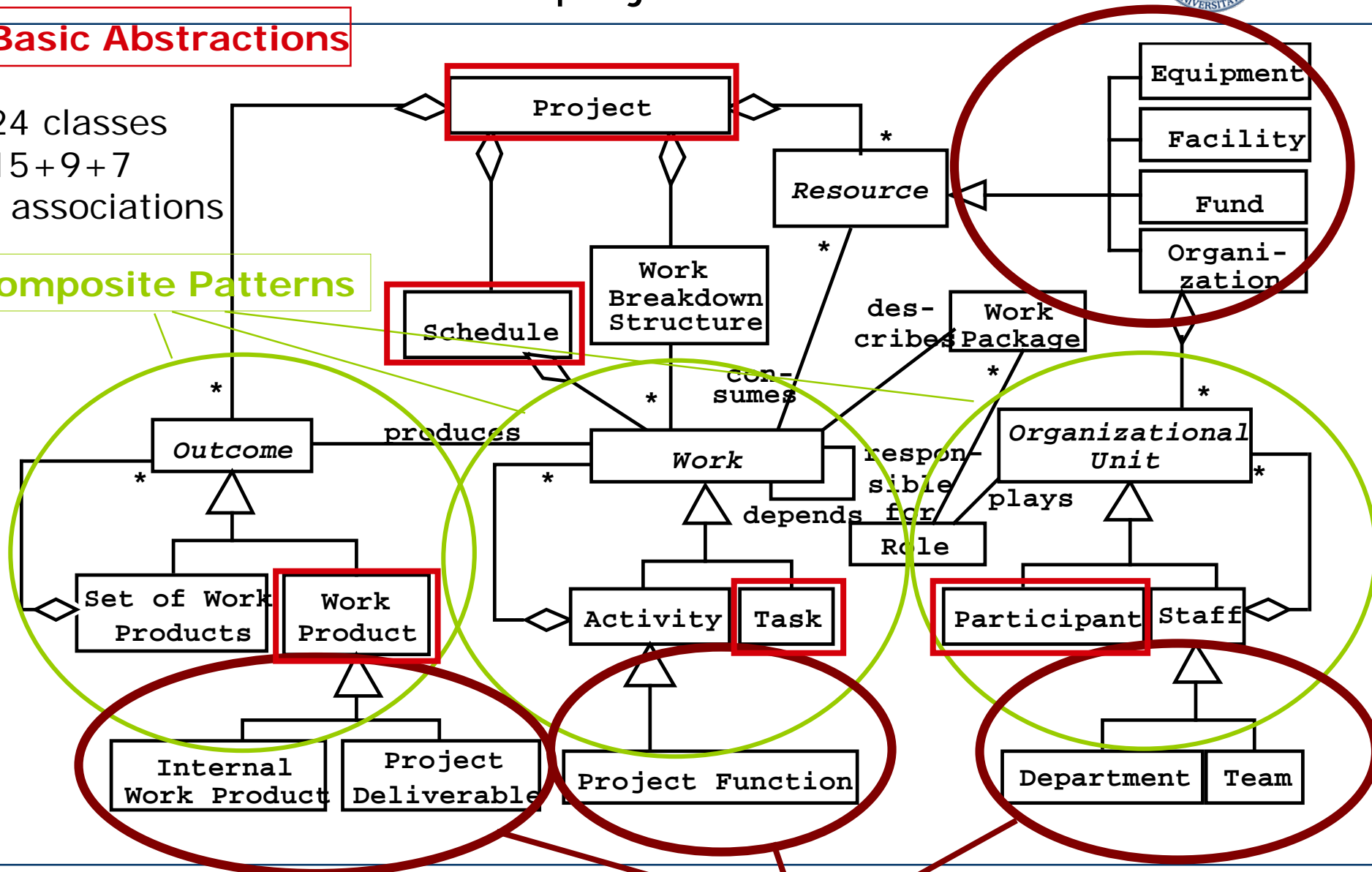
- To communicate a complex model we use navigation and reduction of complexity
 - We do not simply use a picture from a CASE tool and dump it in front of somebody
 - The key is to navigate through the model so the user can follow it
- We start with a very simple model and then decorate it incrementally
 - Start with key abstractions (use animation)
 - Then decorate the model with the additional classes
- To reduce the complexity of the model even further, we
 - Apply the use of inheritance (for taxonomies, and for design patterns)
 - If the model is still too complex, we show subclasses only separately
 - Then identify (or introduce) patterns in the model
 - We make sure to use the name of the patterns

Example: a model of a software project

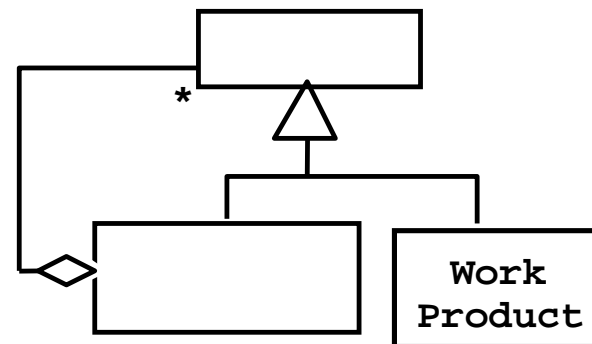
Basic Abstractions

24 classes
15+9+7
associations

Composite Patterns



- There are 55 basic elements (classes, associations) in the model
 - plus association names and multiplicities
- Your short-term memory can hold about 5 to 9 elements
- Redraw the complete model for Project from your memory using the following knowledge
 - Key abstractions: **Project**, **WorkProduct**, **Task**, **Schedule**, **Participant**
 - WorkProduct, Task and Participant are modeled with composite patterns, such as

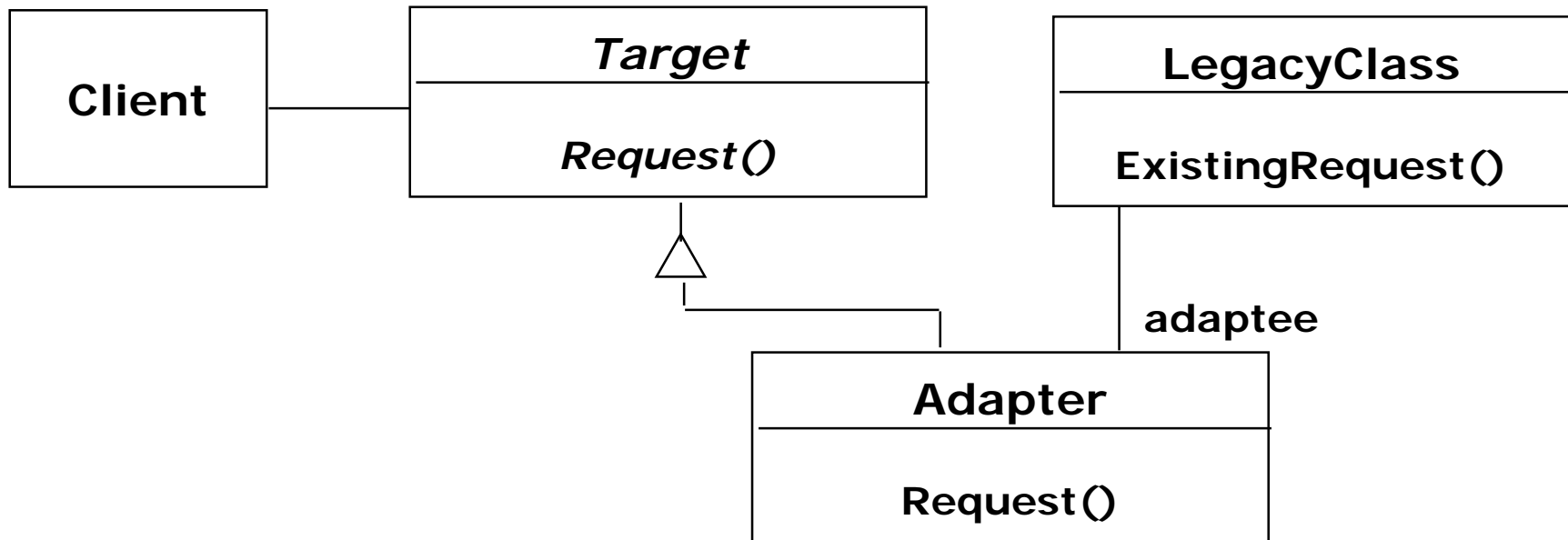


- You have 5 minutes!

Adapter pattern

Also known as *Wrapper* pattern

- Problem: We need to provide a service that conforms to a given target interface T.
We have an existing (legacy) implementation of that service, but it has a different interface S.
- Solution idea: Introduce an adapter class A that implements T based on S
 - Then use an A object plus an S object in place of a T object
- Used in Interface engineering and reengineering
- Two adapter patterns:
 - Class adapter: Uses multiple inheritance
 - Object adapter: Uses single inheritance and delegation
- Object adapters are much more frequent
 - We will only cover object adapters

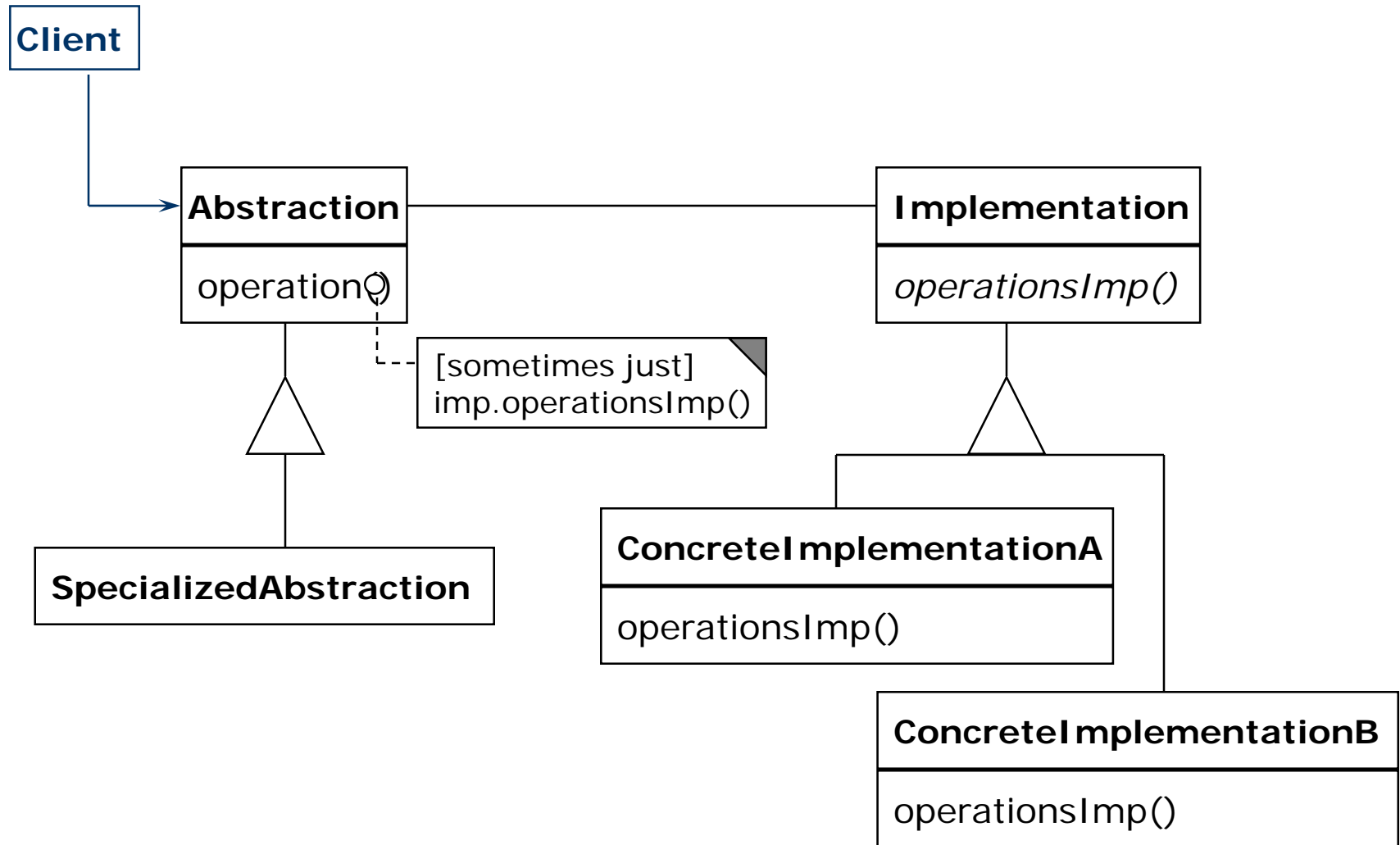


- Target and Adaptee (usually called legacy system) pre-exist the Adapter
 - Target may be realized as an interface in Java
- Interface inheritance is used to specify the interface of the Adapter class
- Delegation is used to bind an Adapter and a legacy class (Adaptee)

Also known as *Handle/Body* pattern

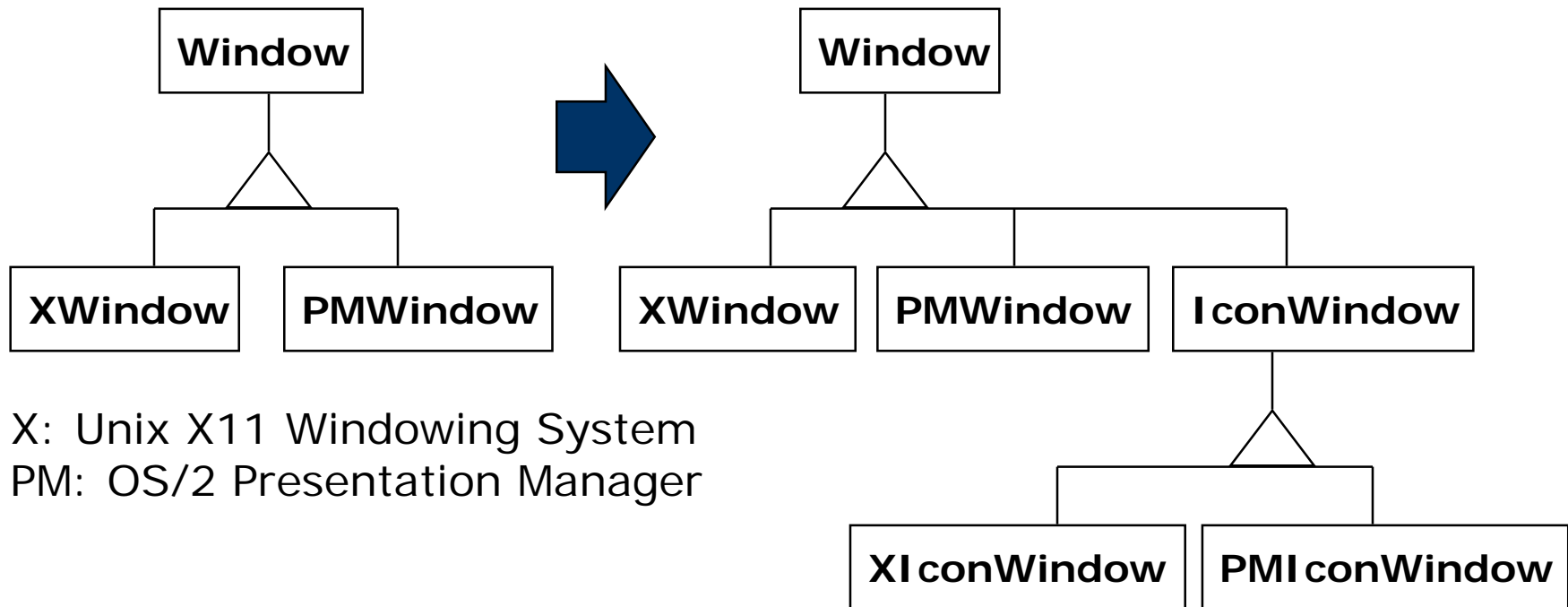
- Problem: We need a complex domain abstraction (that may even evolve over time) that is realized on a technical basis that also evolves (or may vary or be exchanged completely)
 - Put differently: We want to decouple an abstraction from its implementation so that the two can vary independently
- Allows different implementations of an interface to be decided upon dynamically

Solution structure of Bridge pattern

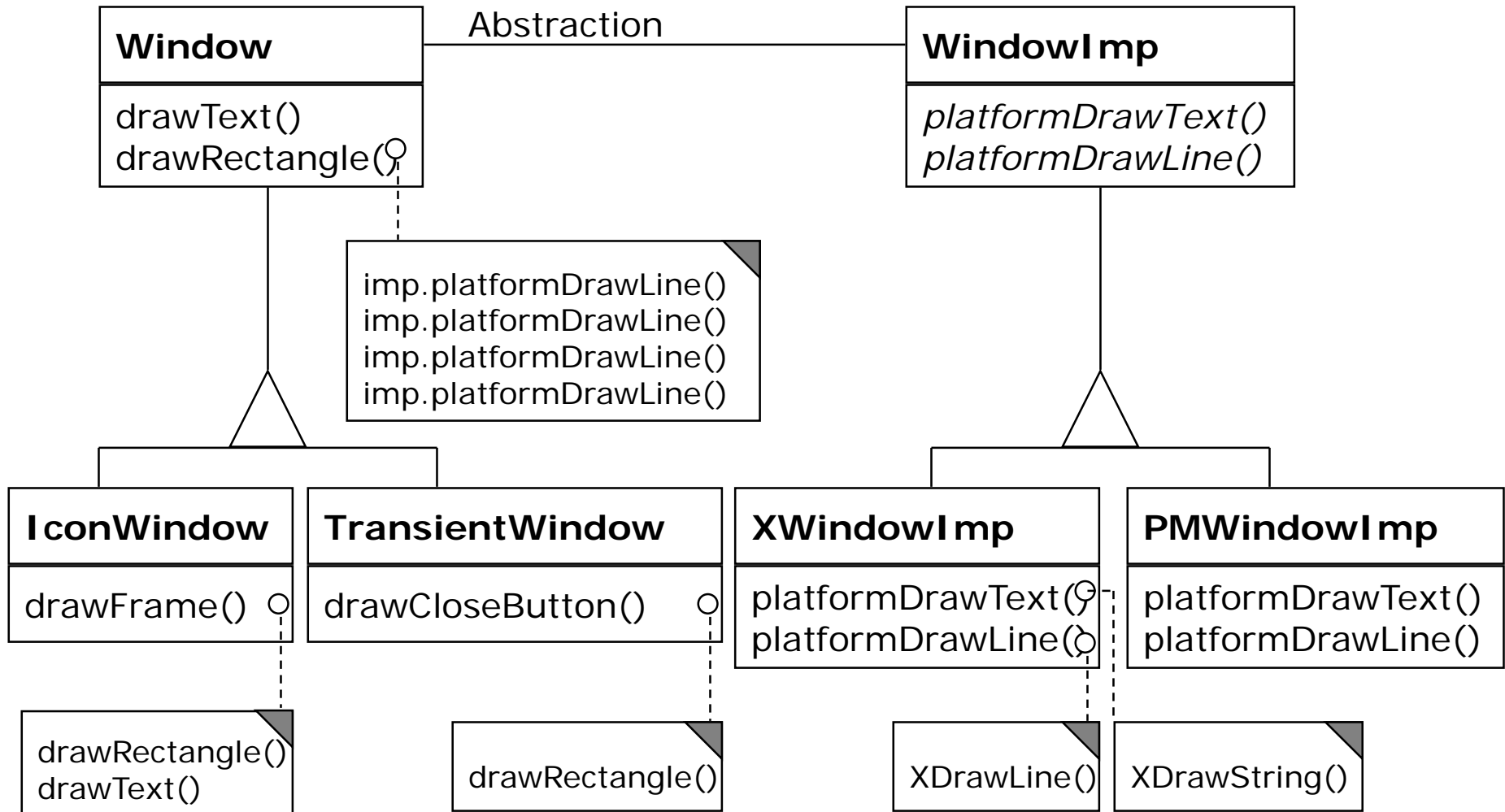


Bridge motivation

- GUI libraries often need two inheritance hierarchies:
 - multiple classes for the GUI domain abstractions (design space)
 - multiple implementations for each (solution space)
 - (one per platform: Mac, Windows, X11, OS/2, etc.)
- Combining these into one leads to giant hierarchies:



Bridge example

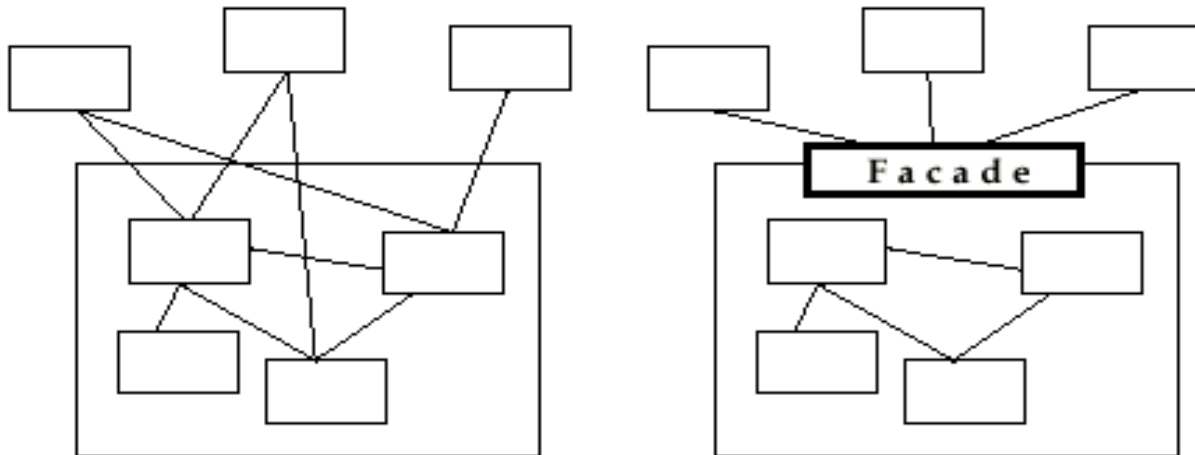


(Simplified. Actual GUI libraries are more complex than this)

- Similarities:
 - Both are used to hide the details of the underlying implementation
- Difference:
 - The adapter pattern is geared towards making unrelated components work together
 - Applied to systems after they're designed (reengineering, interface engineering)
 - A bridge, on the other hand, 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 "object zoo", even if these are not known at analysis or system design time

Façade pattern

- Provides a unified interface to a set of objects in a subsystem
- A facade defines a higher-level interface that makes the subsystem easier to use
 - i.e. it abstracts away many details
- Facades allow us to provide a closed architecture
 - When a module consists of multiple classes, the Façade represents the module's interface



Design patterns encourage reusable designs

- A facade pattern should be used for each subsystem in a software system; it defines the visible services
 - The facade will delegate requests to the appropriate components within the subsystem
 - Most of the time the façade does not need to be changed when the component is changed
- Adapters interface to existing components
 - For example, a smart card software system should interface to different smart card readers via different adapters
- Bridges should be used to interface to a set of objects
 - where the full set is not known at analysis or design time
 - when the subsystem must be extended later after the system has been deployed and client programs are in the field (dynamic extension)

1. Avoid implementation inheritance, always prefer interface inheritance
 - Because implementation inheritance often results in cascading changes when you modify the superclass
 - When you are tempted to use implementation inheritance, consider delegation instead
2. Apply "design by contract" throughout each inheritance hierarchy
 - Each subclass operation must require at most the preconditions of the superclass and must provide at least the postconditions of the superclass
 - Because only then code using the superclass will always also work correctly with each subclass
 - Make sure not to violate this rule when redefining superclass methods
 - A subclass must never hide operations implemented in a superclass

- Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides: "*Design Patterns: Elements of Reusable Software*", 1994.
 - The classic "Gang of Four" (GoF) book. Collection of basic design patterns found when constructing GUI frameworks, but useful in many situations
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: "*Pattern-Oriented Software Architecture: A System of Patterns*", 1996
 - The other classic (sometimes called "Gang of Five" book). Discusses architecture patterns, design patterns, idioms, and pattern systems
- <http://c2.com> "*The Portland Pattern Repository*"
 - The world's first wiki, created for discussing design patterns (and very many other things).
 - Interesting!

- Design patterns are solution ideas for common problems such as
 - separating an interface from (a number of alternate) existing implementations
 - wrapping around a (set of) legacy class(es)
 - protecting a caller from platform-specific changes
- A (oo-)design pattern describes how to compose a few classes
 - use delegation and inheritance
 - provide a robust and modifiable solution
- The idea underlying the pattern should be adapted/refined for the specific system under construction
 - Customization of the design and purpose
 - Reuse of existing solutions
 - Combination with other patterns

Thank you!