Object Design: Design Patterns

Software Engineering I
Lecture 13

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

Miscellaneous

- **Mid Term Exams**
  - Will be given out next Tuesday
- **Exercises for Software Engineering I**
  - Miniproject mandatory
  - Duration: January 11 - February 8
  - Exercises and Miniproject are mandatory for Final Exam
  - Room: 00.08.038
  - Time: 16:30-18:00 or 17:00-18:30 (Based on your vote)
- **New instructor**
  - **Maximillian Kögel or short Max (koegel@in.tum.de)**
  - Max will run the exercise sessions and the miniproject
  - First meeting: This Thursday!
- **Final Exam**
  - **Date: February 17, 10:00-12:30 (note the Change!)**

Outline of the Lecture

- Good Models
- Design 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
- Design Patterns 2
  - Proxy
  - Command
  - Observer
  - Strategy
  - Abstract Factory
  - Builder

Detailed Schedule for the Miniproject

- **January 11:**
  - Review of Design Patterns
  - Introduction: The Game Asteroids
  - Team Assignments for first Exercise (about 5 per team)
  - Exercise 1 given out
  - Exercise 1 due on January 17
- **January 18:**
  - Sample solution for Exercise 1 on Lecture Portal
  - Introduction: The Game Management System ARENA
  - Explanation of Problem Statement
  - Exercise 2 given out
  - Exercise 2 due on January 24
- **January 25:**
  - Miniproject Clinique ("First aid")
  - Questions and answers about analysis and system design
- **February 1:**
  - Miniproject Clinique
  - Questions and answers about object and implementation
- **February 8:**
  - Presentation in front of the class
  - Team-based presentation
  - Testing of soft skills
    - Control of topic
    - Control of audience
    - Control of time

What are Good Models?

- Good models do not tax the mind
  - A good model requires minimal mental effort to understand
- Good models reduce complexity
  - Turn complex tasks into easy ones (by good choice of representation)
  - By the use of symmetries
- Good models use abstractions
  - Taxonomies
- Good models have organizational structure:
  - Memory limitations are overcome with an appropriate representation ("natural model")

Is this a good Model?

```
public interface SeatImplementation {
  public int GetPosition();
  public void SetPosition(int newPosition);
}
public class Stubcode implements SeatImplem    ation {
  public int GetPo
    // stub code f
  }
  ...
}
public class AimSe                             tion {
  public int GetPo
    // actual cal                              m
  }
  ….
}
public class SARTSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the SART seat simulator
 }
  ...
}
```

It depends!

A Game: Get-15

- Start with the nine numbers 1,2,3,4, 5, 6, 7, 8 and 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 take it.
- The first person to have any three numbers that total 15 wins the game.
- Example:

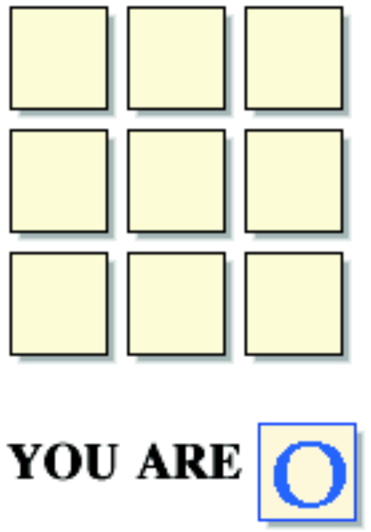
You:	**1**	**5**	**3**	**8**	
Opponent:	**6**	**9**	**7**	**2**	**Opponent Wins!**

Characteristics of Get-15

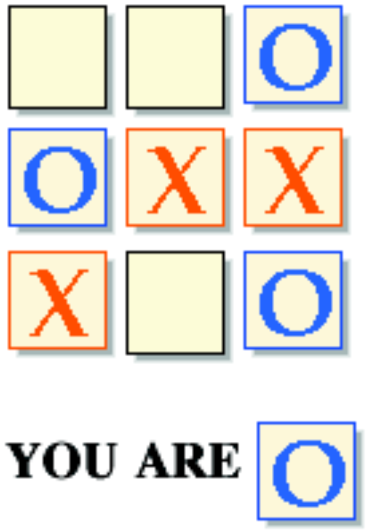
- Hard to play,
- The game is especially hard, if you are not allowed to write anything done.

- 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

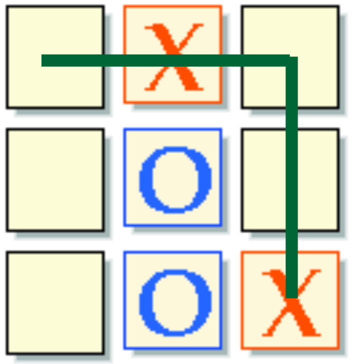
Another Game: Tic-Tac-Toe

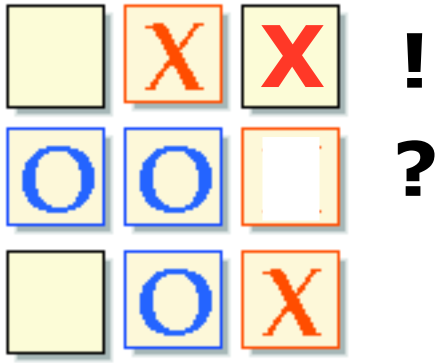


A Draw Sitation

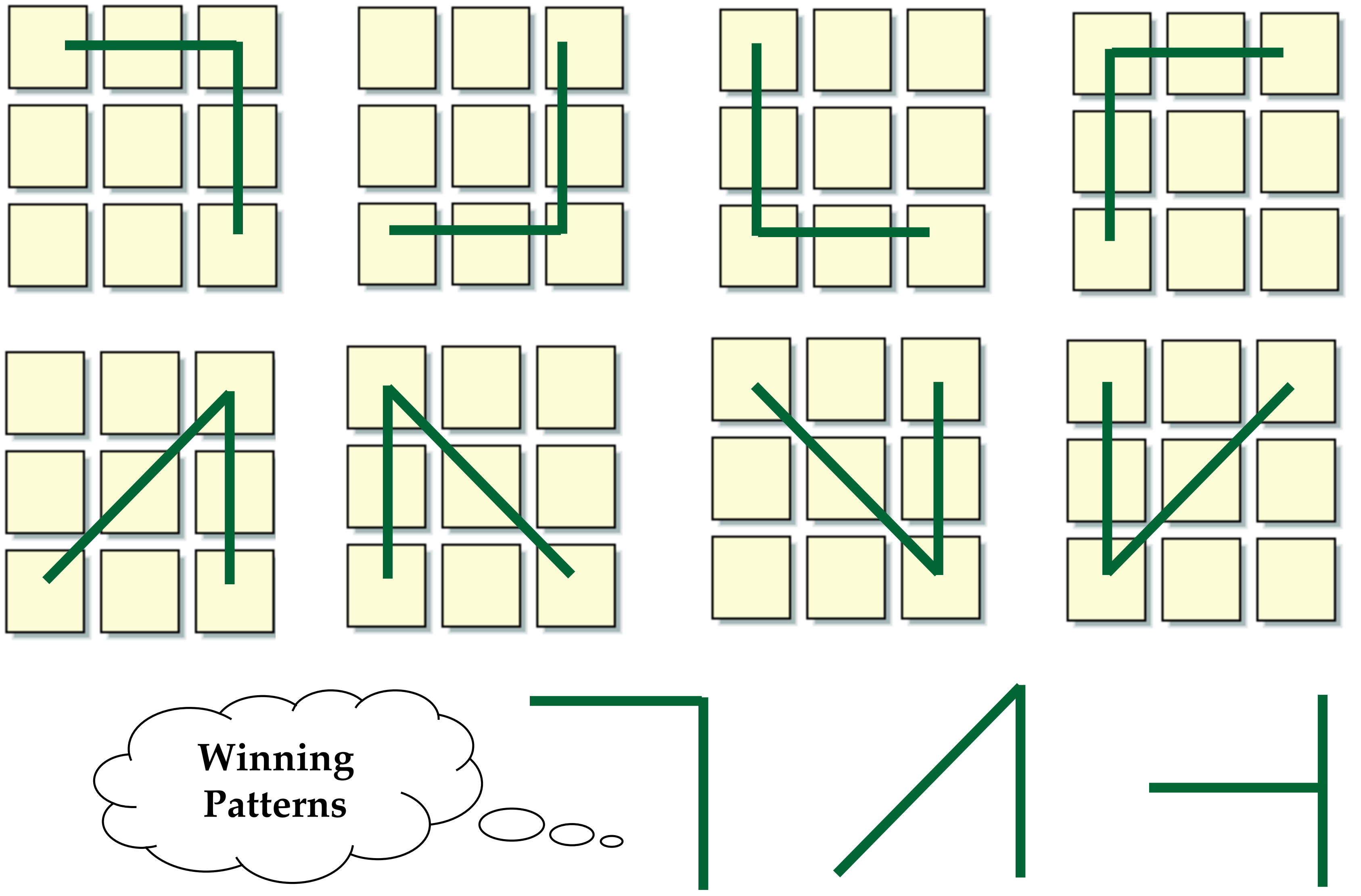


Strategy for determining a winning move



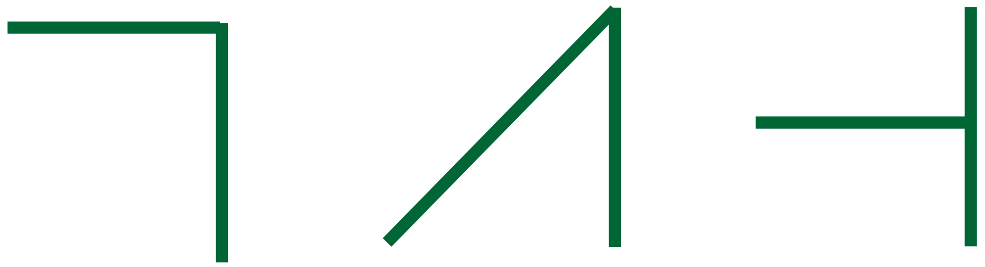


Winning Situations for Tic-Tac-Toe



Tic-Tac-Toe is "Easy"

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



- **Symmetries**:
  - The player needs to remember only these three patterns to deal with 8 different game siuations
  - The player needs to memorize only 3 opening moves and their responses.

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

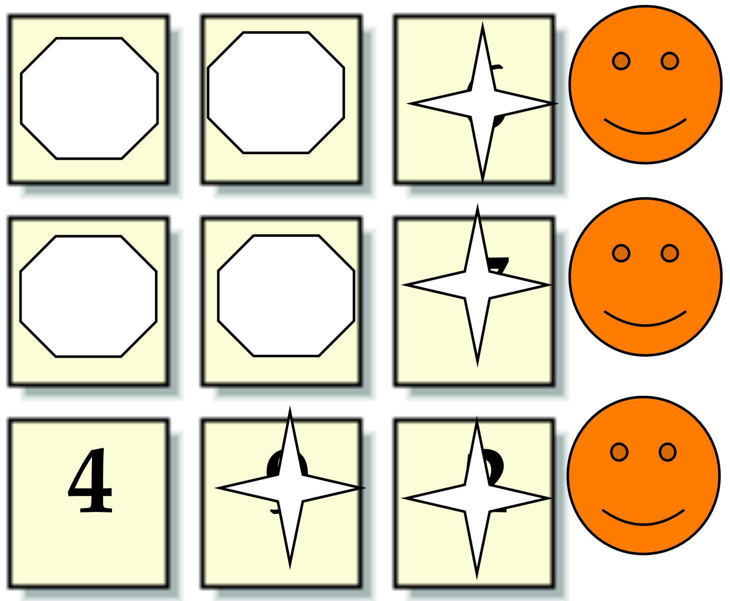
- Any three numbers that solve the 15 problem also solve tic-tac-toe.
- Any tic-tac-toe solution is also a solution the 15 problem
- To see the relationship between the two games, we simply arrange the 9 digits into the following pattern

8	1	6
3	5	7
4	9	2

⯃ You: 1 5 3 8

✦ Opponent: 6 9 7 2

8	1	6
3	5	7
4	9	2



- During object modeling we do many transformations and changes to the object model.
- It is important to make sure the object design model stays simple!

- Design patterns are used to keep system models simple (and reusable)

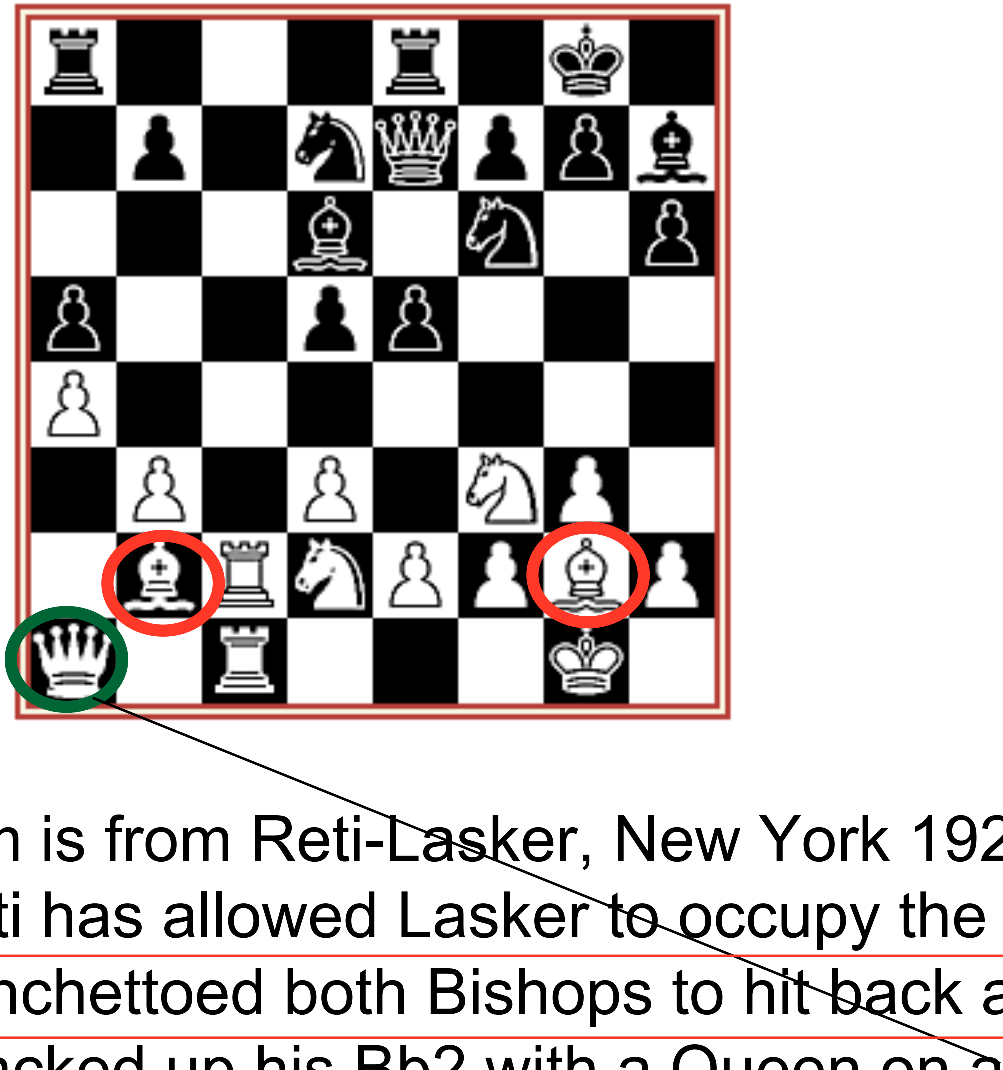
Is this a Good Model?

1.Nf3 d5 2.c4 c6 3.b3 Bf5 4.g3 Nf6 5.Bg2 Nbd7 6.Bb2 e6 7.O-O Bd6 8.d3 O-O 9.Nbd2 e5 10.cxd5 cxd5 11.Rc1 Qe7 12.Rc2 a5 13.a4 h6 14.Qa1 Rfe8 15.Rfc1

This is a fianchetto!

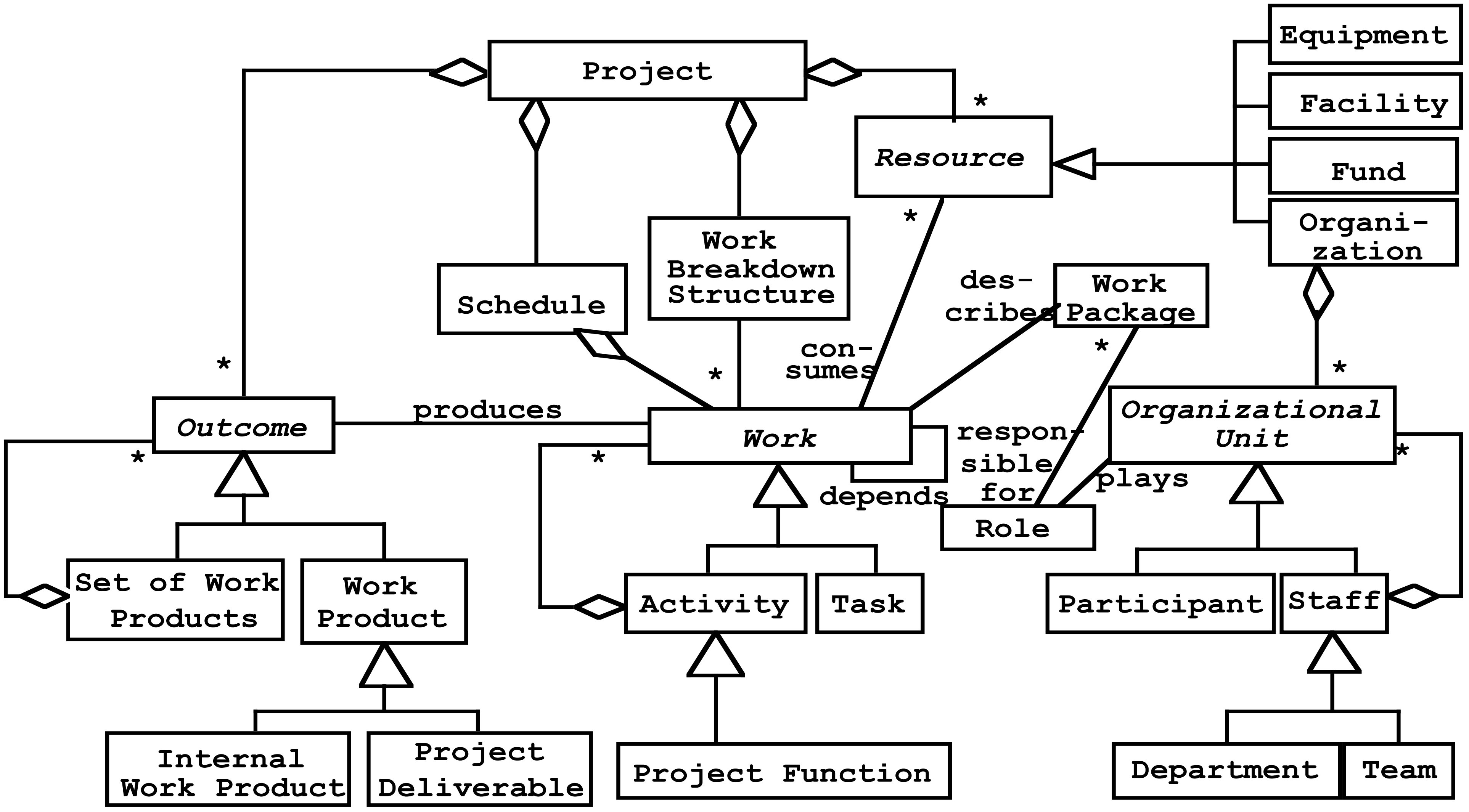
The fianchetto is one of the basic building-blocks of chess thinking.

Fianchetto (Reti-Lasker)



The diagram is from Reti-Lasker, New York 1924. We can see that Reti has allowed Lasker to occupy the center but Rtei has fianchettoed both Bishops to hit back at this, and has even backed up his Bb2 with a Queen on a1!

Is this a Good Model?



Design Patterns reduce the Complexity of Models

- To communicate a complex model we use navigation and reduction of complexity
- 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 further, we
  - Apply the use of inheritance
    - If the model is still too complex, we show the subclasses on a separate slide
  - Then identify patterns in the model

Finding Objects

- The hardest problems in object-oriented system development are:
  - Identifying objects
  - Decomposing the system into objects
- Requirements Analysis focuses on application domain:
  - Object identification
- System Design addresses both, application and implementation domain:
  - Subsystem Identification
- Object Design focuses on implementation domain:
  - Additional solution objects

Techniques for Finding Objects

- Requirements Analysis
  - Start with Use Cases. Identify participating objects
  - Textual analysis of flow of events (find nouns, verbs, ...)
  - Extract application domain objects by interviewing client (application domain knowledge)
  - Find objects by using general knowledge
- System Design
  - Subsystem decomposition
    - Layers and partitions
- Object Design
  - Find additional objects to reduce the object design gap
    - Applying solution domain knowledge

Another Source for Finding Objects: Design Patterns

- A design pattern describes
  - A problem which occurs over and over again in an environment
  - The core of the solution in such a way that one can reuse the this solution without having to solve the problem again.

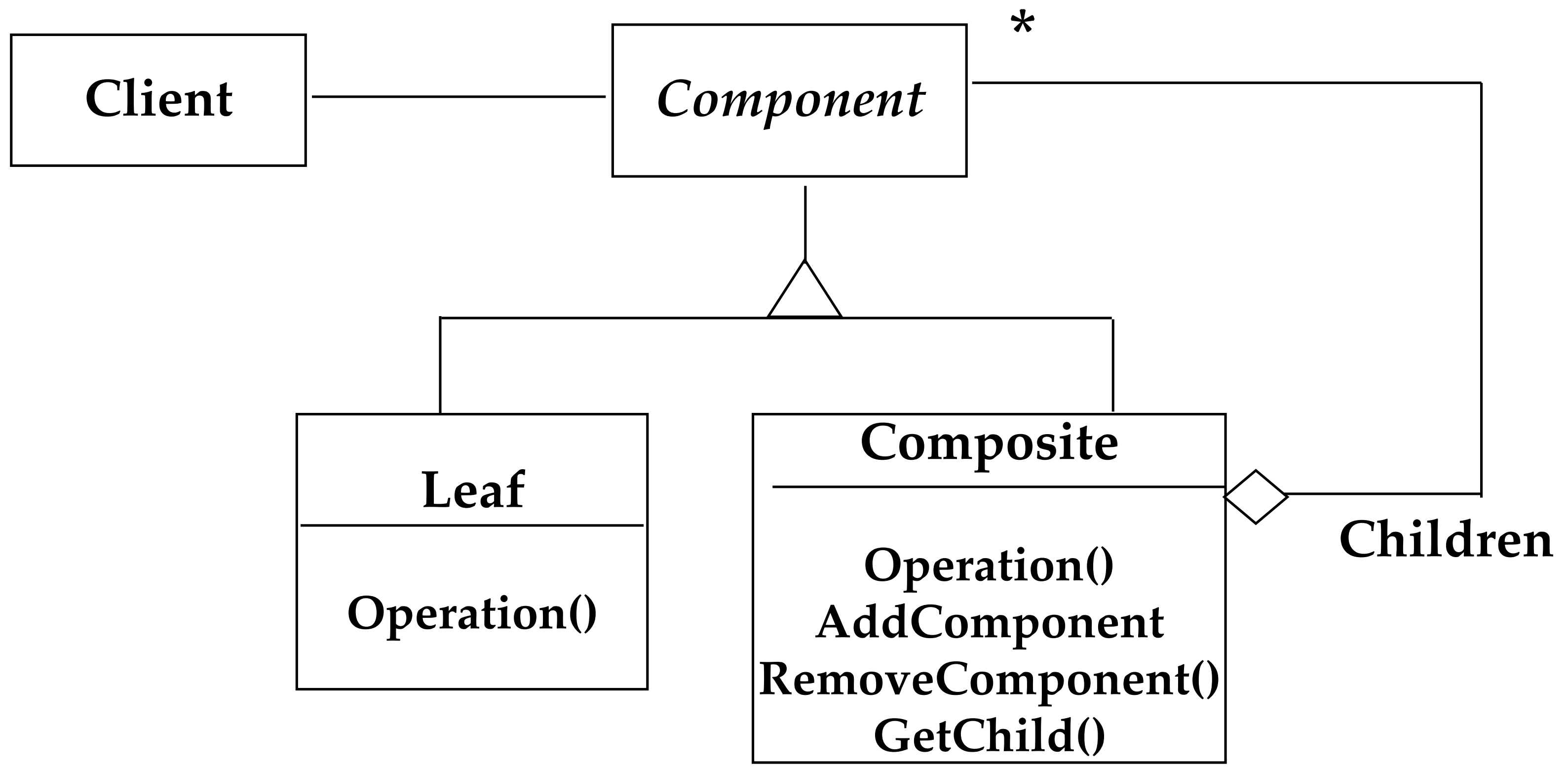
What is common between these definitions?

- Definition Software System
  - A software system consists of subsystems which are either other subsystems or collection of classes

- Definition Software Lifecycle:
  - The software lifecycle consists of a set of development activities which are either other actitivies or collection of tasks

Introducing the Composite Pattern

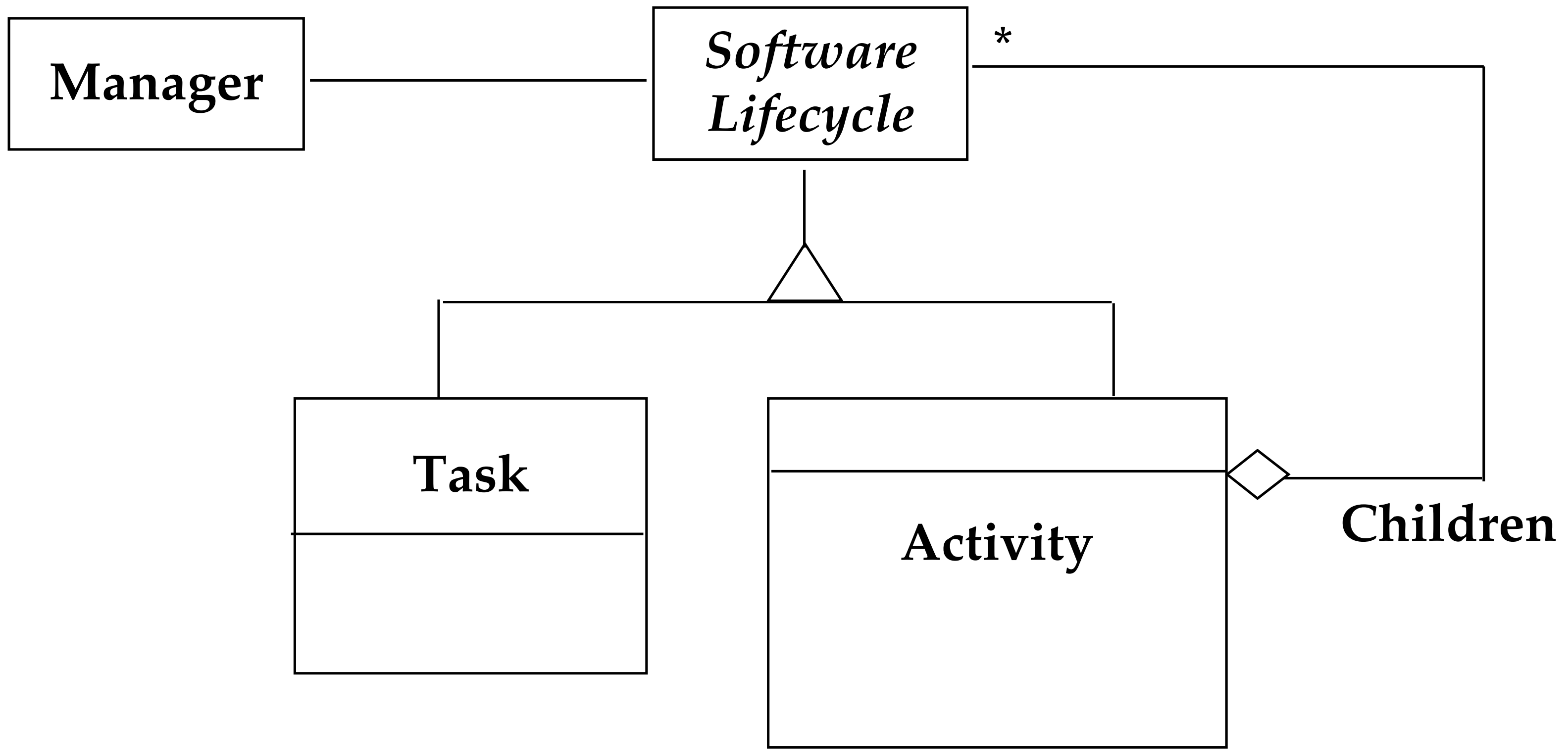
- Models tree structures that represent part-whole hierarchies with arbitrary depth and width.
- The Composite Pattern lets client treat individual objects and compositions of these objects uniformly



Commonality of Software System and Software Lifecycle

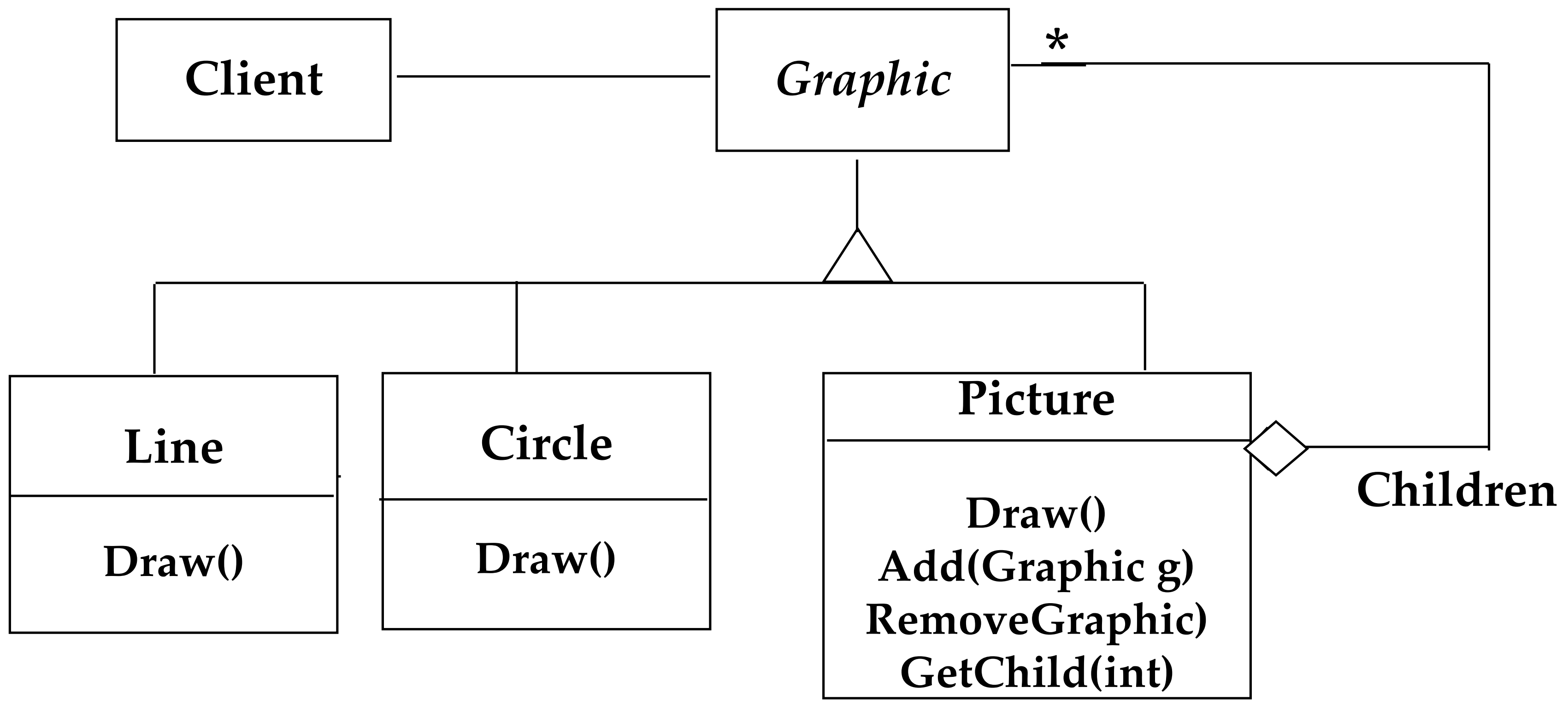
- Software System:
  - Composite: Subsystem
    - A software system consists of subsystems which consists of subsystems , which consists of subsystems, which...
  - Leaf node: Class
- Software Lifecycle:
  - Composite: Activity
    - The software lifecycle consists of activities which consist of activities, which consist of activities, which....
  - Leaf node: Task

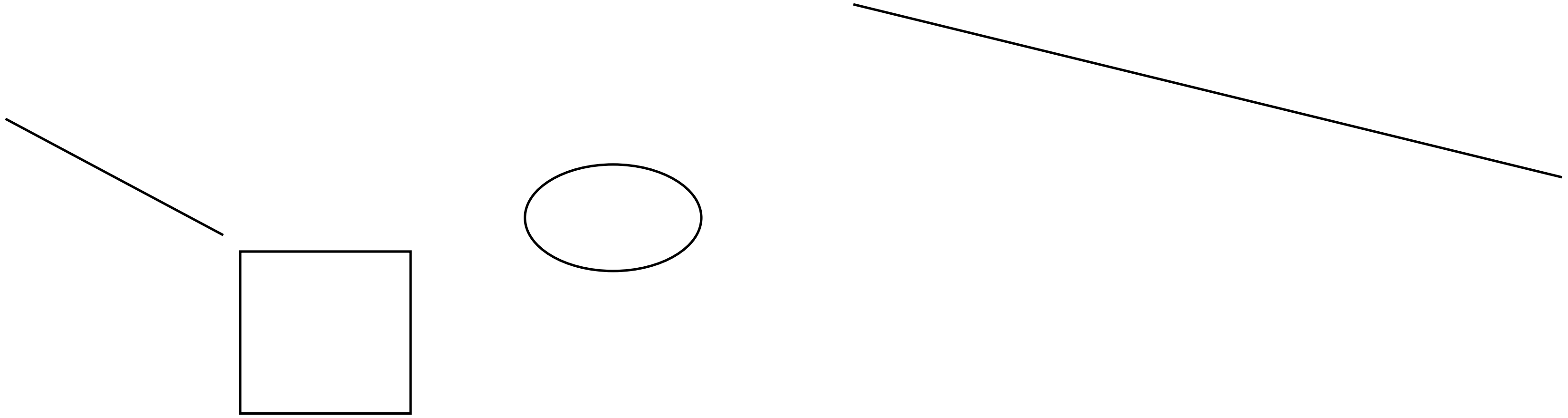
Modeling the Software Lifecycle with a Composite Pattern



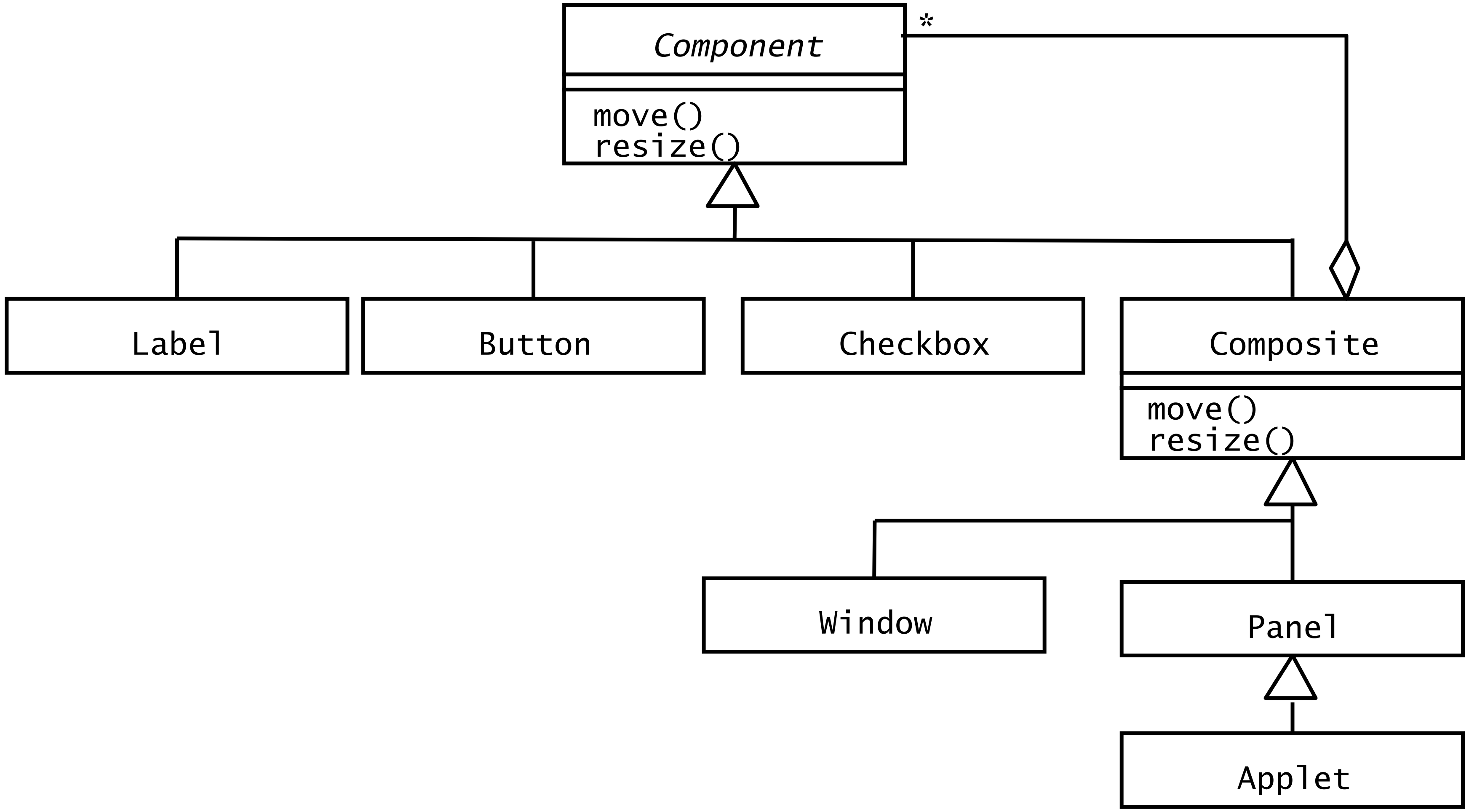
Graphic Applications also use Composite Patterns

- Java's *Graphic* Class represents both primitives (Line, Circle) and their containers (Picture)





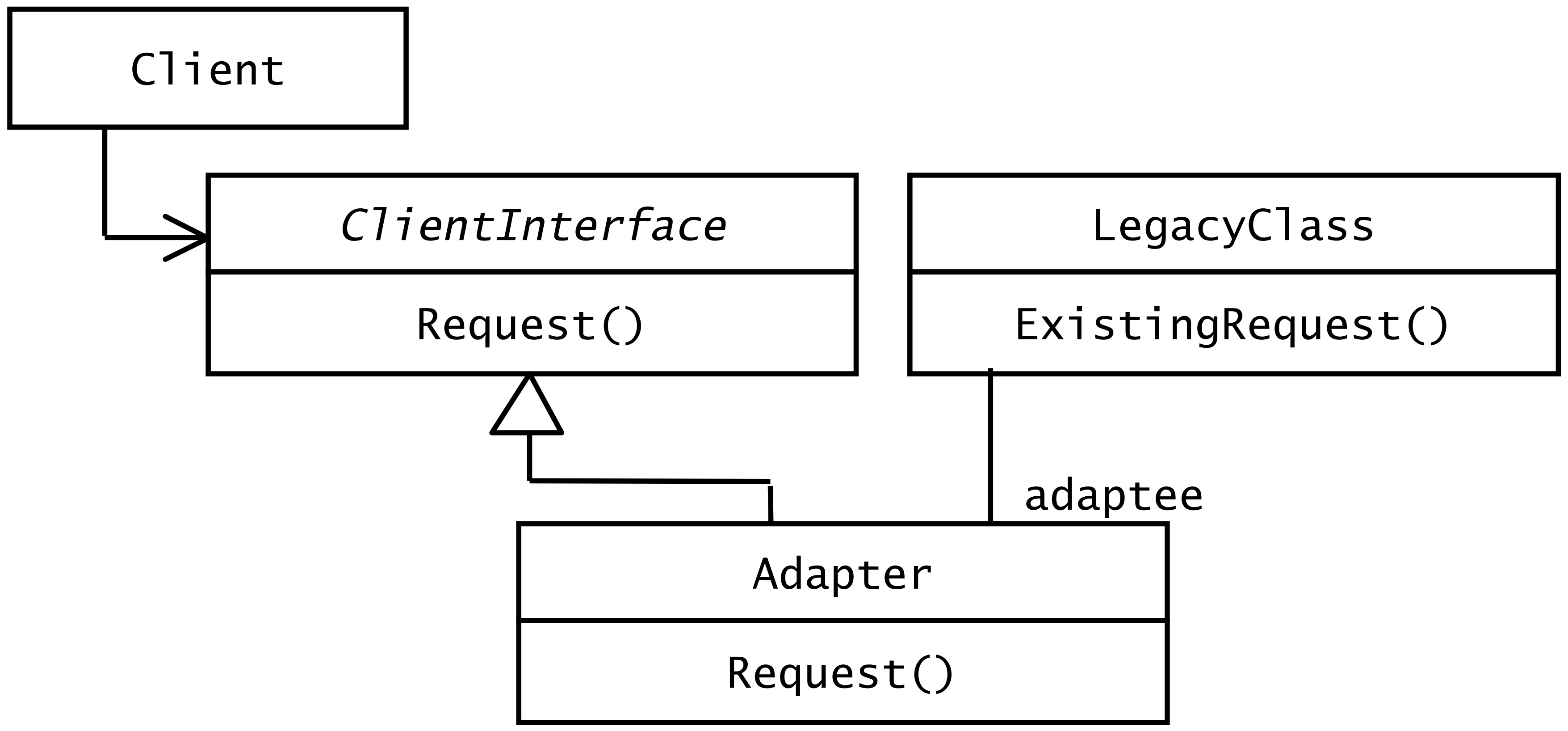
A Java Applet can also be modeled with the Composite Pattern



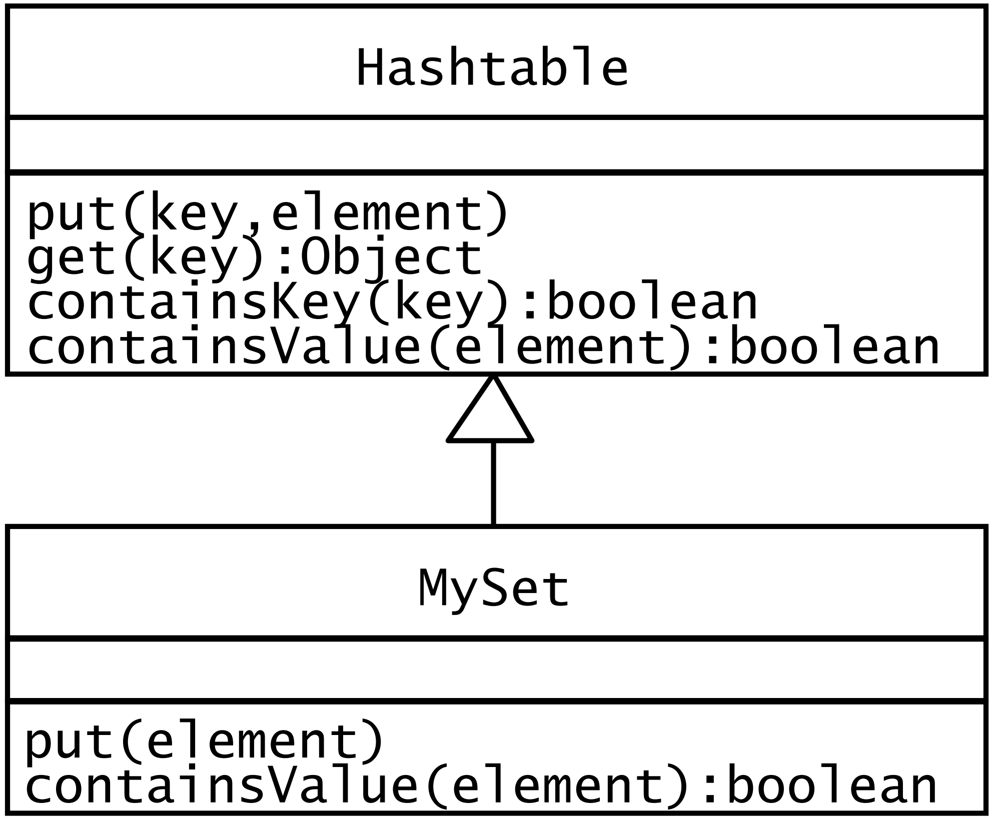
Adapter Pattern

- Adapter Pattern: Converts the interface of a class into another interface clients expect.
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Also known as a wrapper
- Two adapter patterns:
  - Class adapter:
    - Uses multiple inheritance to adapt one interface to another
  - Object adapter:
    - Uses single inheritance and delegation

Adapter pattern

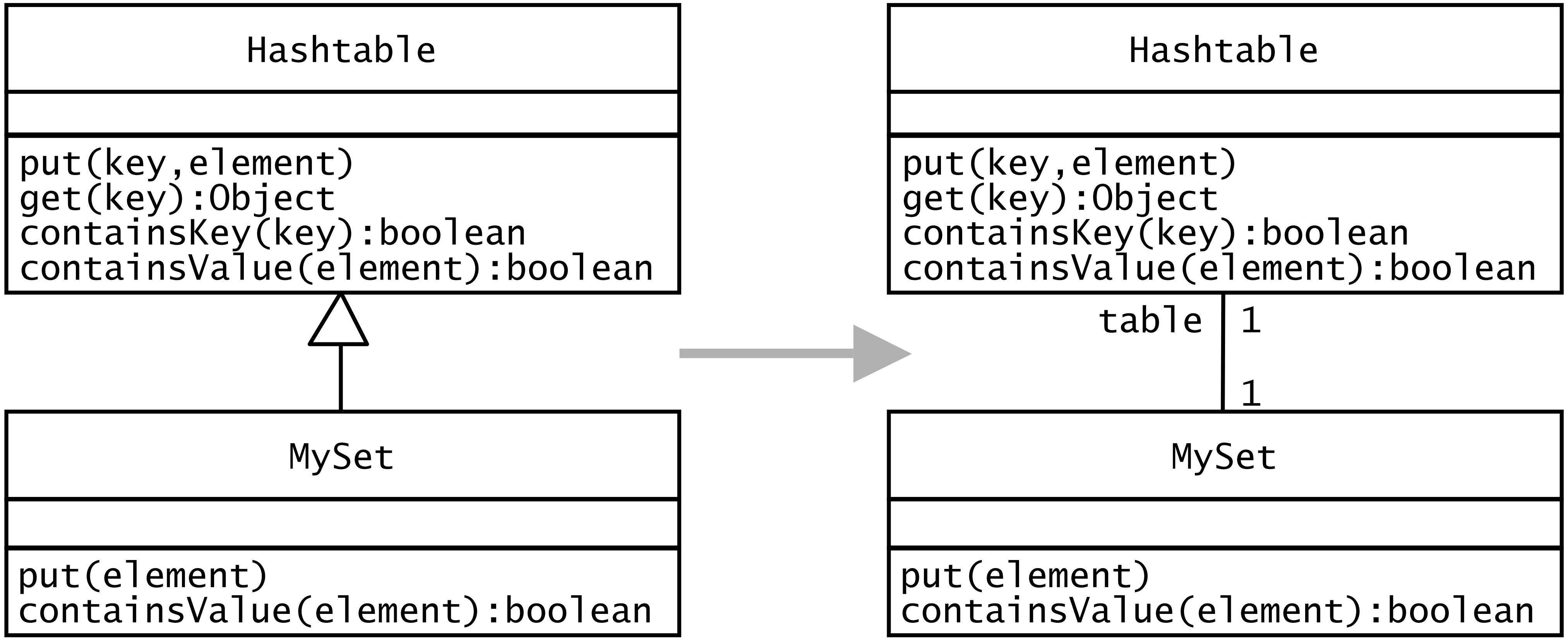


Example: Realizing a Set with a Hashtable



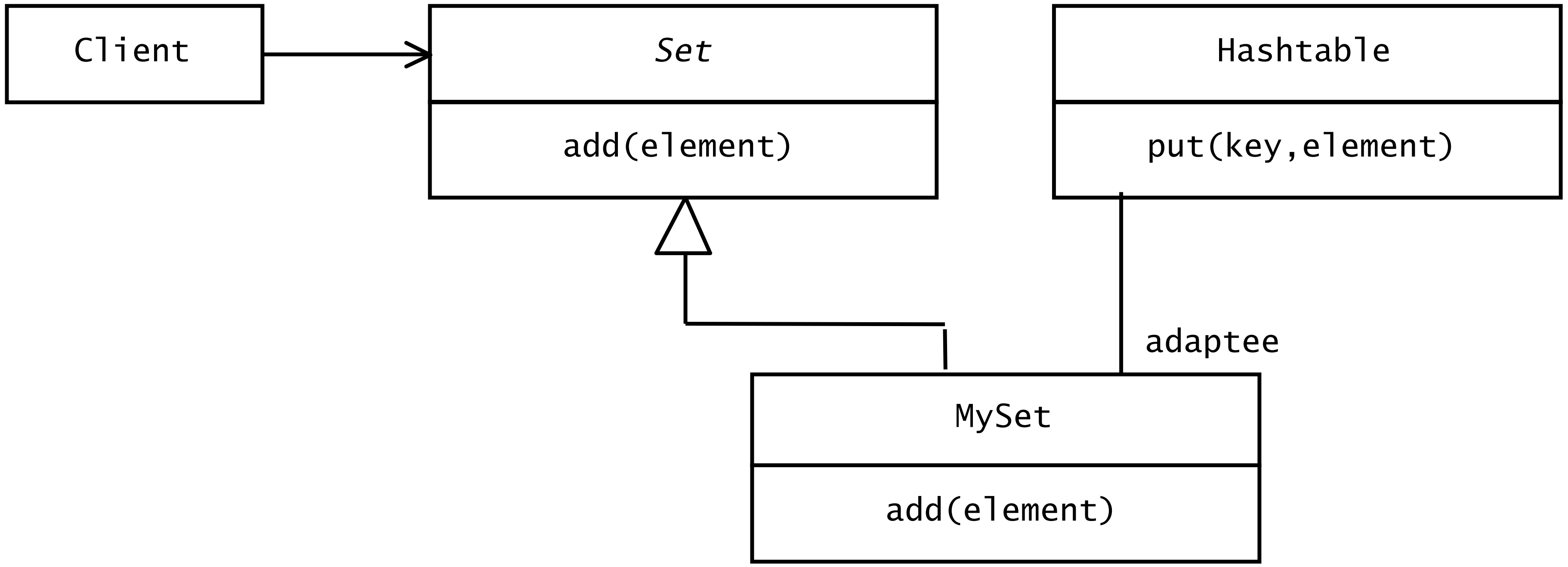
Implementation Inheritance Solution

Example: Realizing a Set with a Hashtable



Solution with Delegation

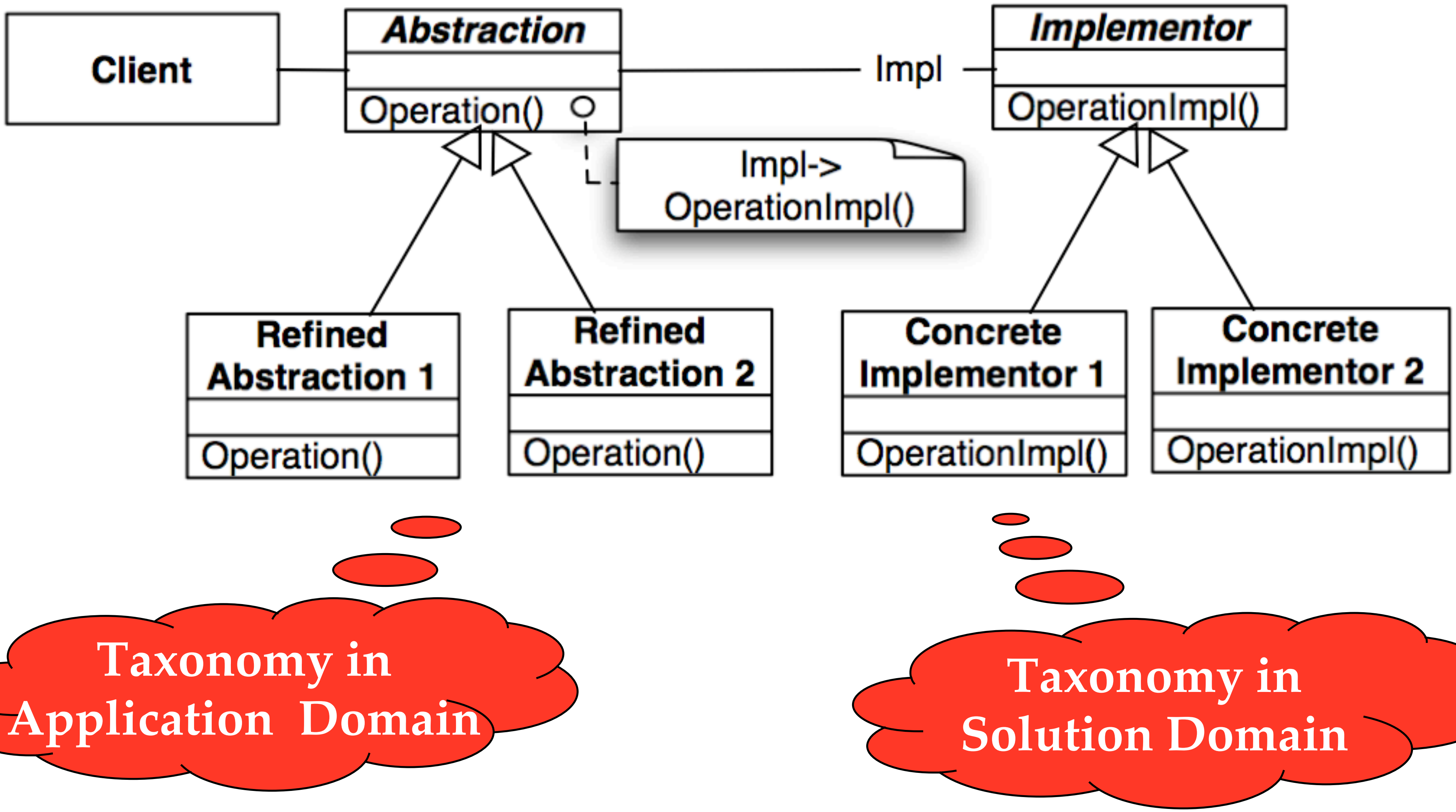
Best Solution: Use of an Adapter



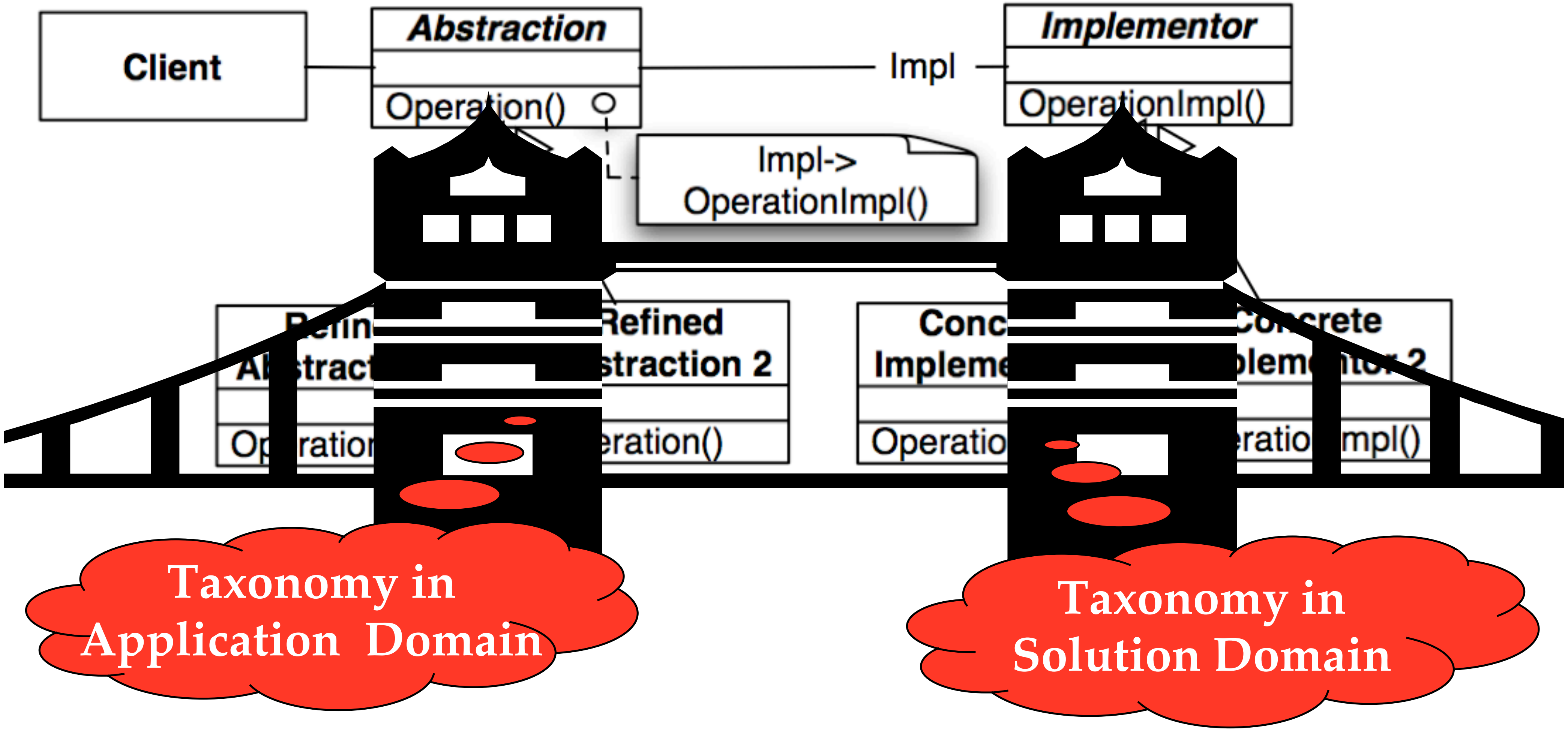
Bridge Pattern

- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”
- Also know as a Handle/Body pattern
- This allows different implementations of an interface to be decided upon dynamically, that means, at the runtime of the system.

Bridge Pattern



Why the Name Bridge Pattern?



Example

Requirements:

- An Asteroid game with two different levels, easy and hard, the hard level treats collisions between asteroids and the space ship differently
- Player should be able to choose game level

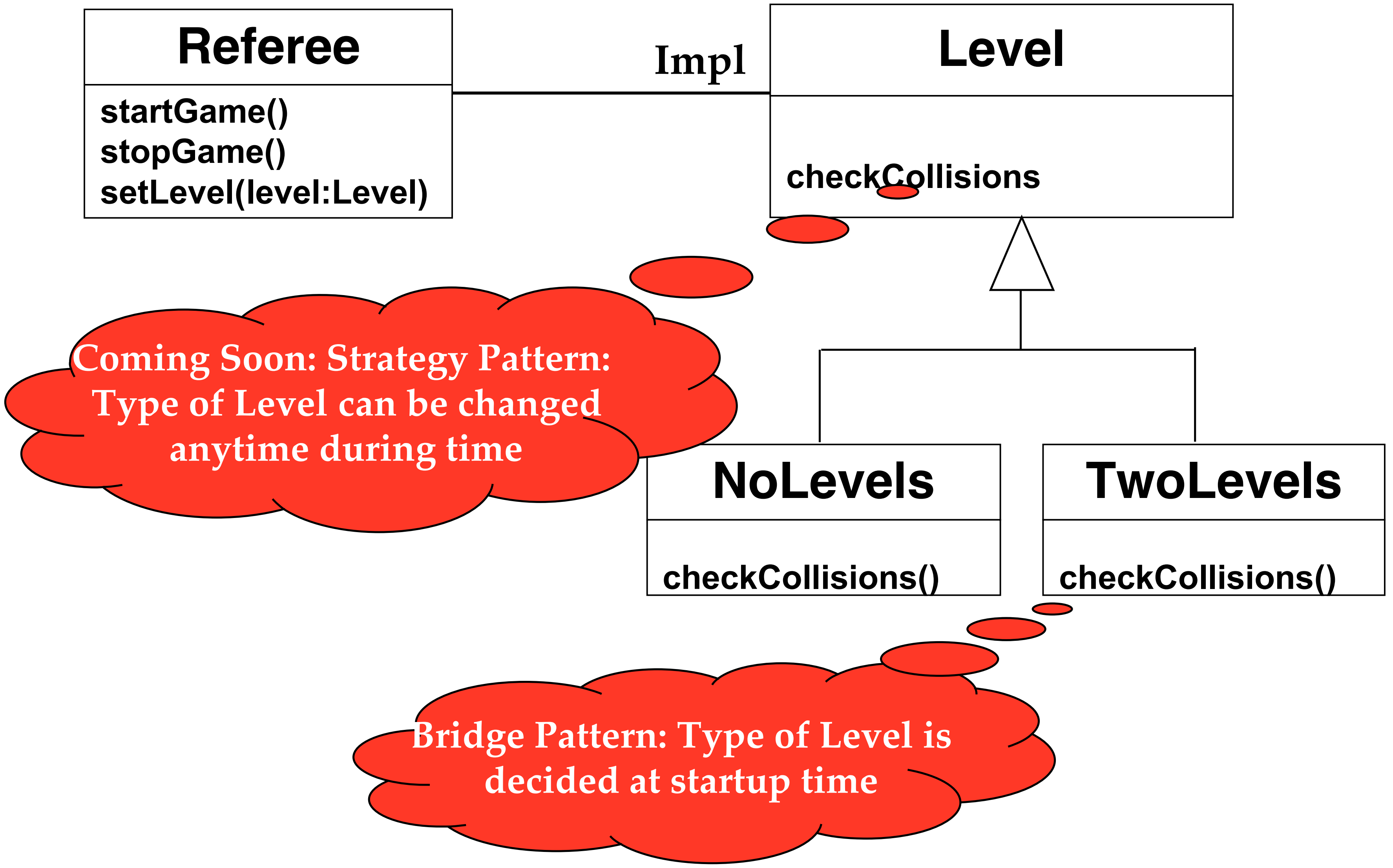
Problem:

- The old system had only one level
- Old customer base needs to be supported

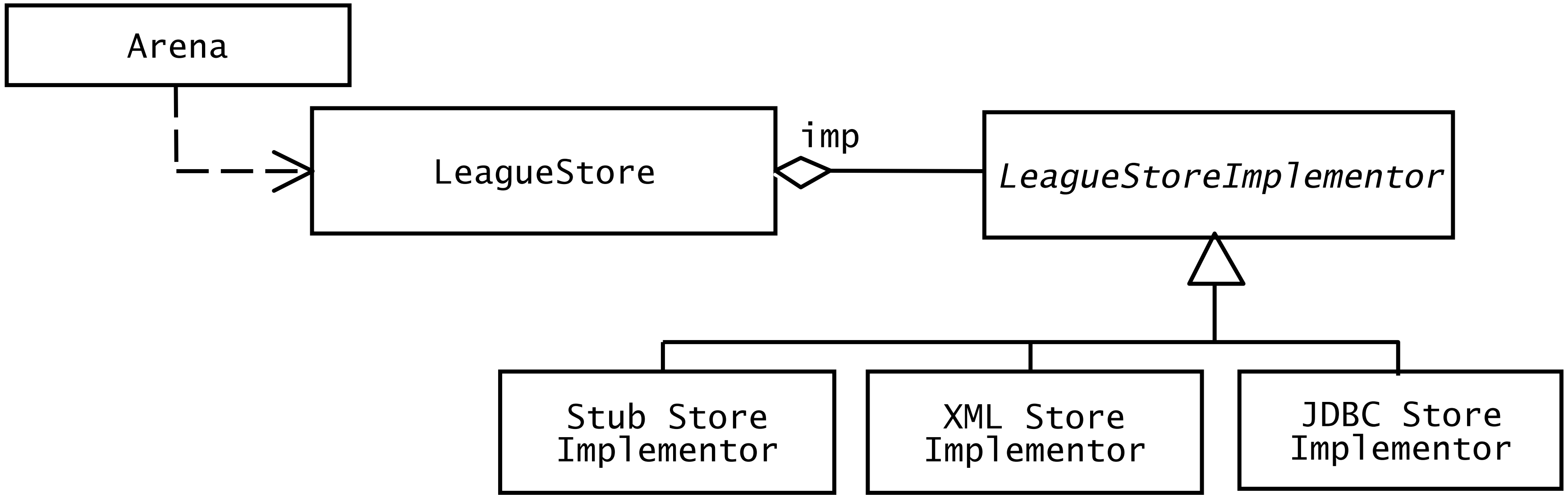
Solution:

- Release a system that supports the old version without game levels and the new version with two game levels.
- Allow the switching between old and new code at system startup time

System Model with Bridge Pattern


Referee
startGame()
stopGame()
setLevel(level:Level)
Impl
Level
checkCollisions
NoLevels
checkCollisions()
TwoLevels
checkCollisions()
Coming Soon: Strategy Pattern: Type of Level can be changed anytime during time
Bridge Pattern: Type of Level is decided at startup time

Using the Bridge Pattern to support multiple Database vendors

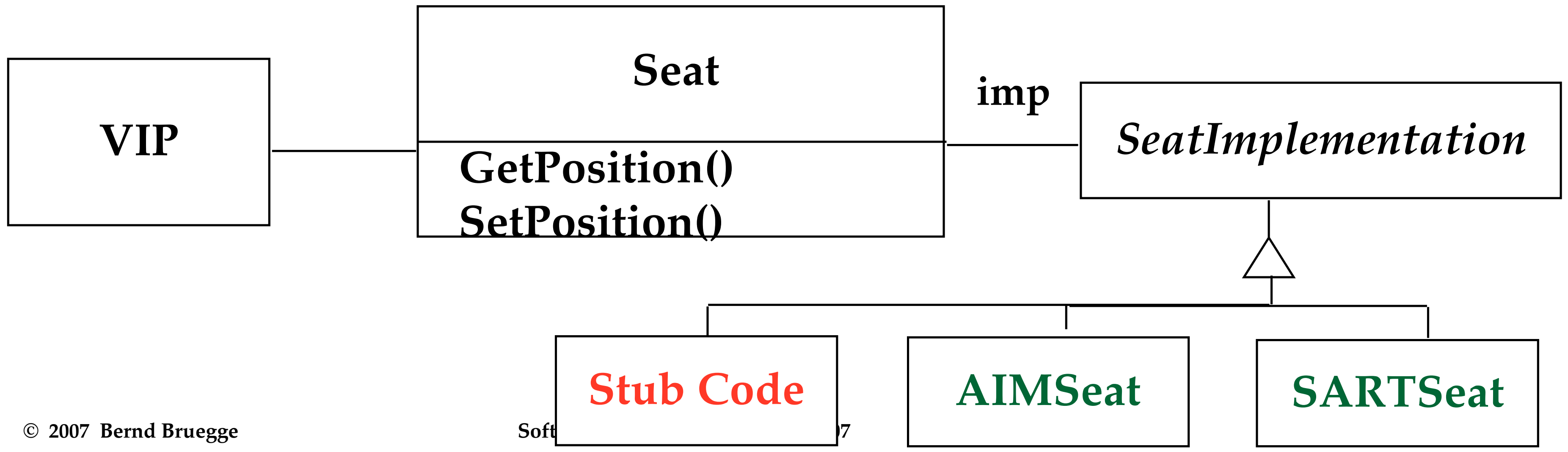


Using the Bridge Pattern during Testing

- 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 Stub Code

Two seat simulators are also available: AIM and SART

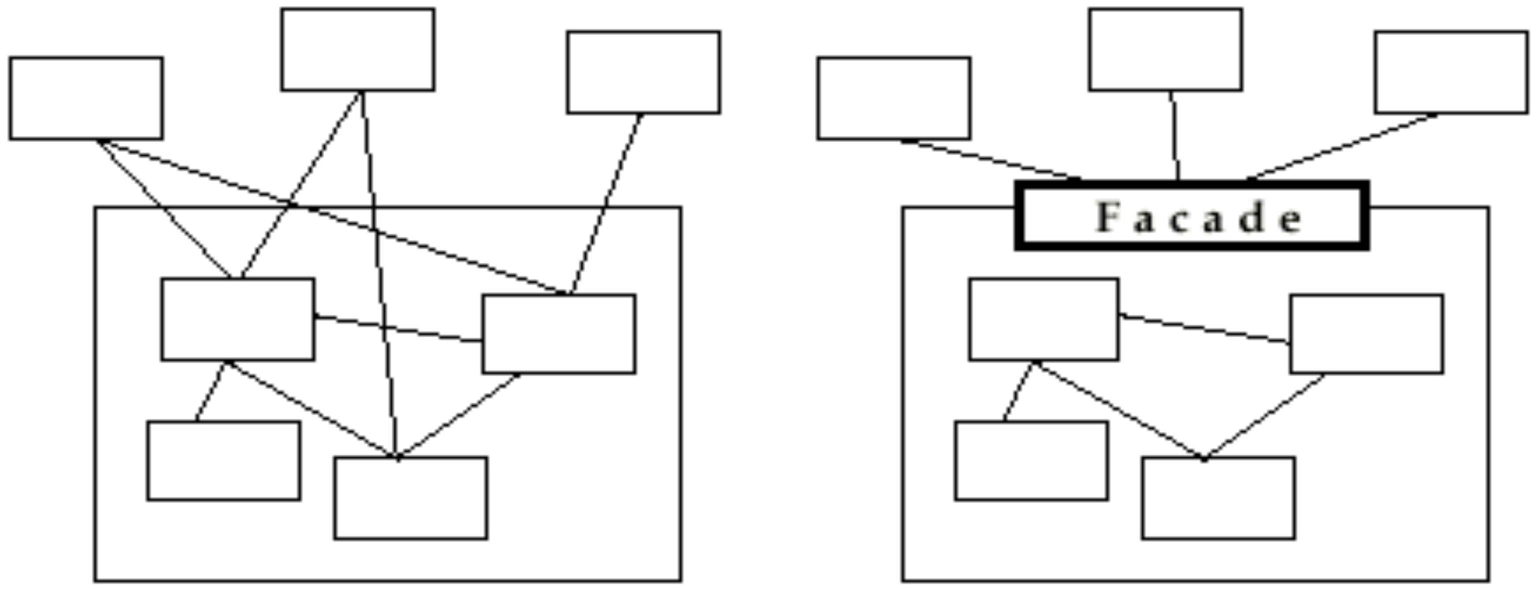


Adapter vs Bridge

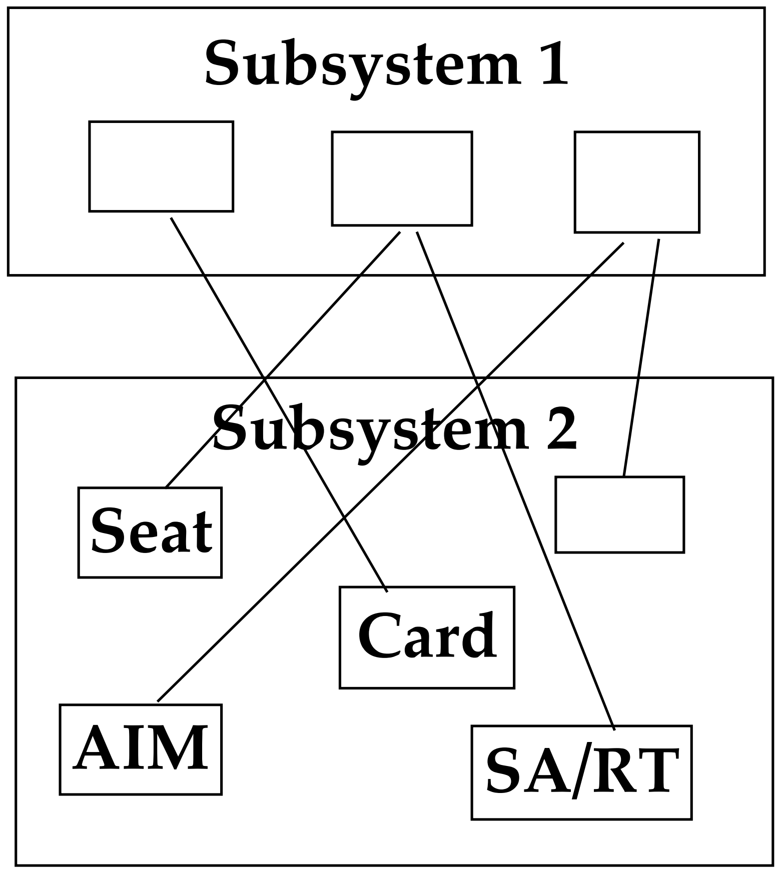
- Similarities:
  - Both design patterns are used to hide the details of the underlying implementation.
- Difference:
  - An adapter makes unrelated components work together
  - A bridge is used up-front in the design to let abstractions and implementations vary independently.

Facade 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 (abstracts out the details)
- Facades allow us to provide a closed architecture

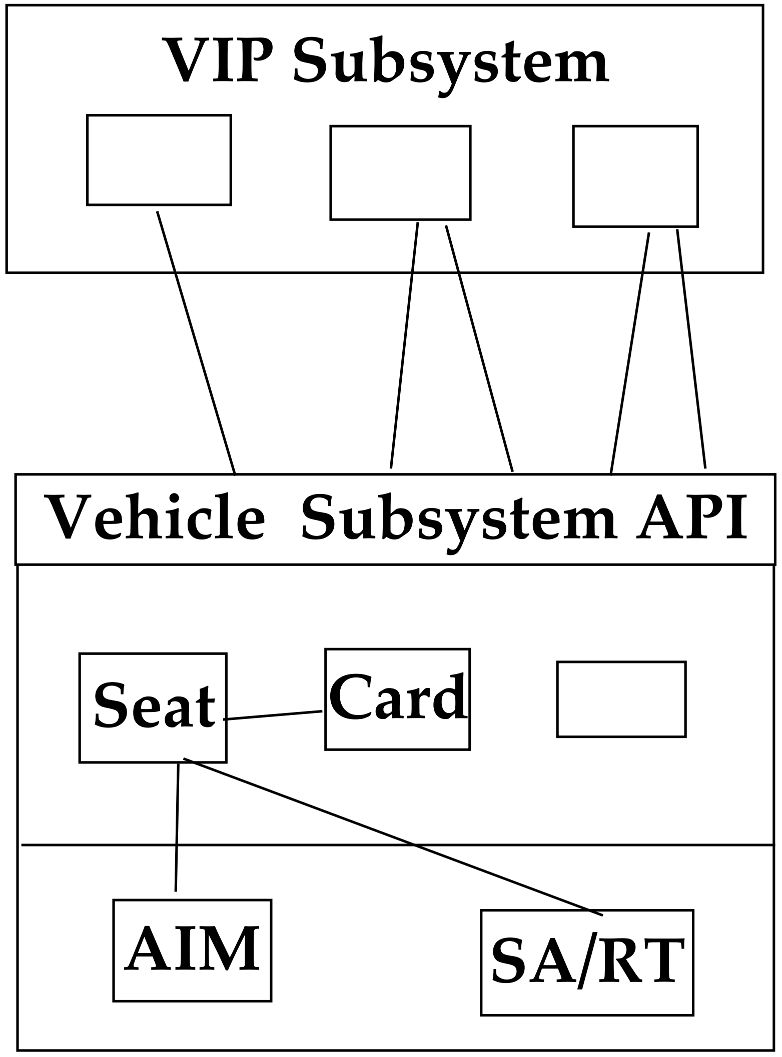


Design Example



Realizing an Opaque Architecture with a Facade

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers
- If a façade is used the subsystem can be used in an early integration test
  - We need to write only a stub



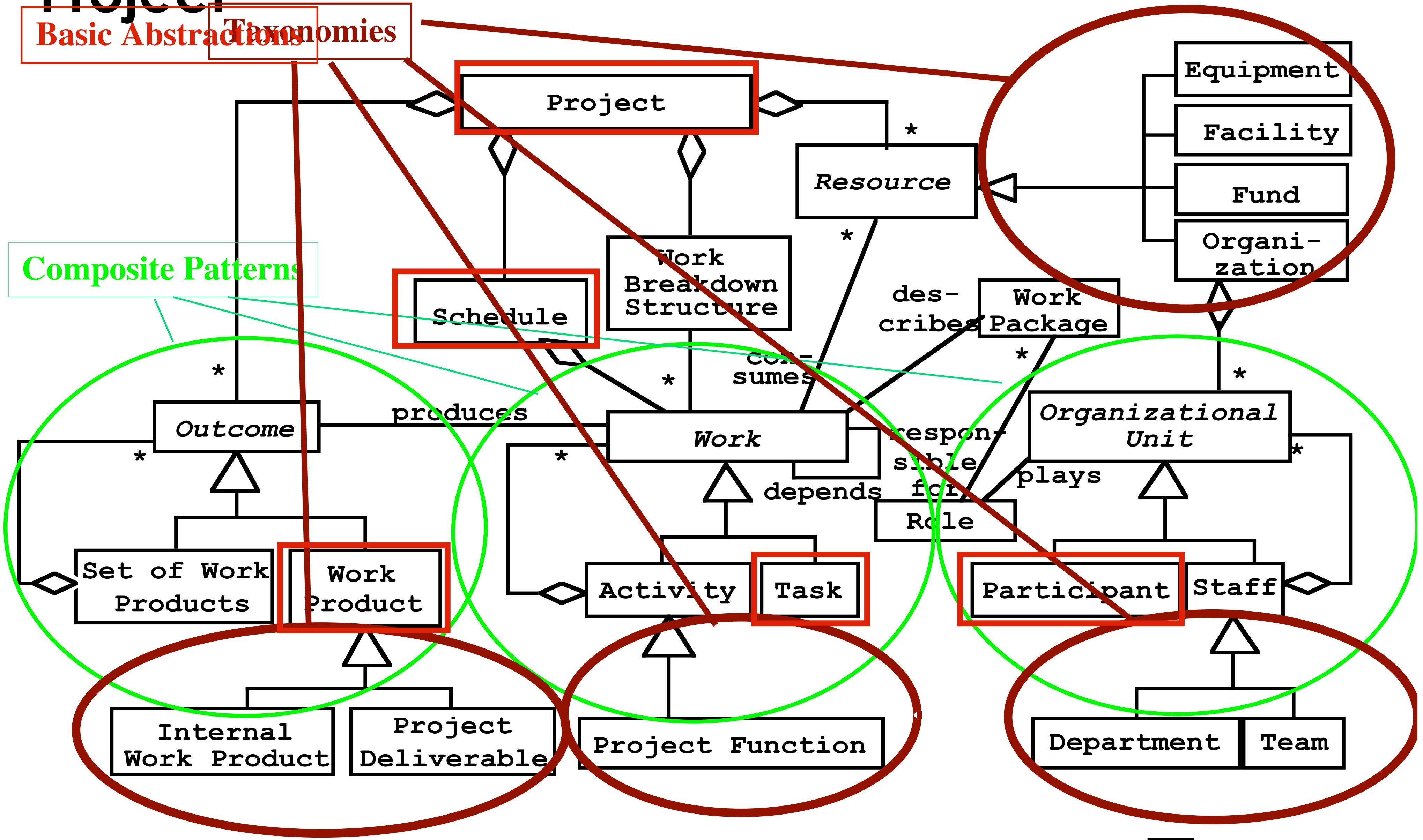
Using Design Patterns for Subsystem Design

- Realization of the Interface Object: Facade
  - Provides the interface to the subsystem
- Interface to stable application domain objects or existing systems: Adapter
  - Provides the interface to the existing system (legacy system)
  - The existing system is not necessarily object-oriented!
- Interface to changing application domain objects or more than one implementation: Bridge
  - Provides the interface to the implementations
  - Decision for a specific implementation is made at startup time

Design Patterns encourage Good Designs

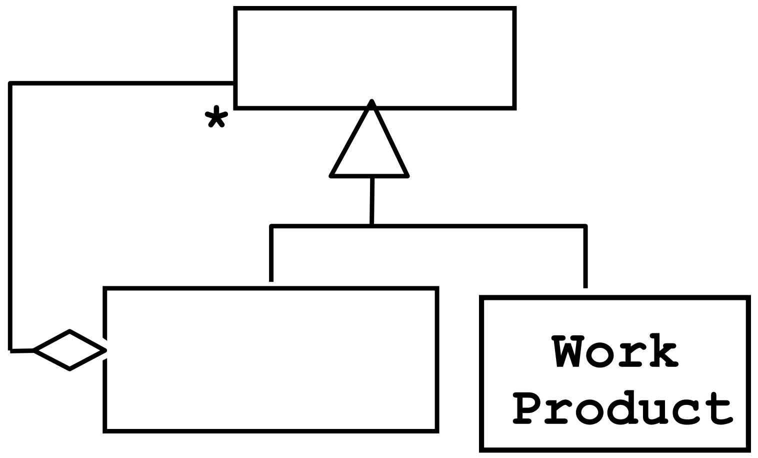
- A facade pattern should be used
  - by all subsystems in a software system. The façade defines the services of the subsystem
- The Adapter Pattern should be used
  - to interface to existing components
- The Bridge Pattern should be used
  - to interface to a set of implementations

Example: A More Complex Model of a Software Project



Exercise

- Redraw the complete model for Project from your memory using the following knowledge
  - The key abstractions are task, schedule, and participant
  - Work Product, Task and Participant are modeled with composite patterns, for example



  - There are taxonomies for each of the key abstractions
- You have 5 minutes!

Additional References

- Design:
  - E. Gamma et.al., Design Patterns, 1994.
- Analysis:
  - M. Fowler, Analysis Patterns: Reusable Object Models, 1997
- System design:
  - F. Buschmann et. al., Pattern-Oriented Software Architecture: A System of Patterns, 1996
- Middleware:
  - T. J. Mowbray & R. C. Malveau, CORBA Design Patterns, 1997
- Process modeling:
  - S. W. Ambler, Process Patterns: Building Large-Scale Systems Using Object Technology, 1998.
- Configuration management:
  - W. J. Brown et. Al., AntiPatterns and Patterns in Software Configuration Management, 1999.

First Prize: Bottle of Champagne (or similar value)

- Price criteria
  - Identify all the patterns
  - Recover all the names
  - Find all abstractions

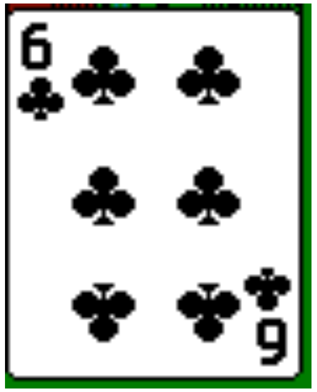
  - First prize:
    - Recover the maximum number of abstractions, design patterns and names.

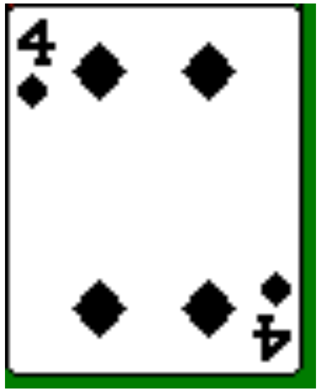
Summary

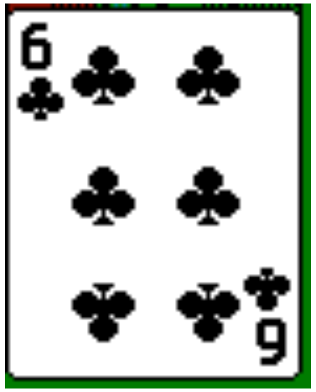
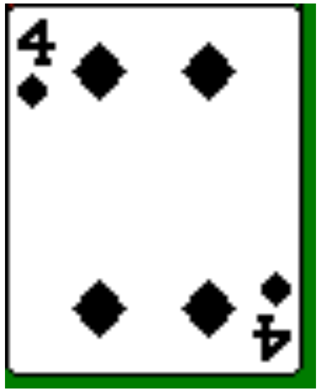
- Design patterns are partial solutions to common problems such as
  - separating an interface from alternate implementations
  - wrapping around a set of legacy classes
  - protecting a caller from changes associated with specific platforms
- A design pattern is composed of a small number of classes
  - uses delegation and inheritance
  - provides a robust and modifiable solution.
- These classes can be adapted and refined for the specific system under construction.
  - Customization of the system
  - Reuse of existing solutions

Patterns are not the cure for everything

What is wrong in the following pictures?





Summary (2)

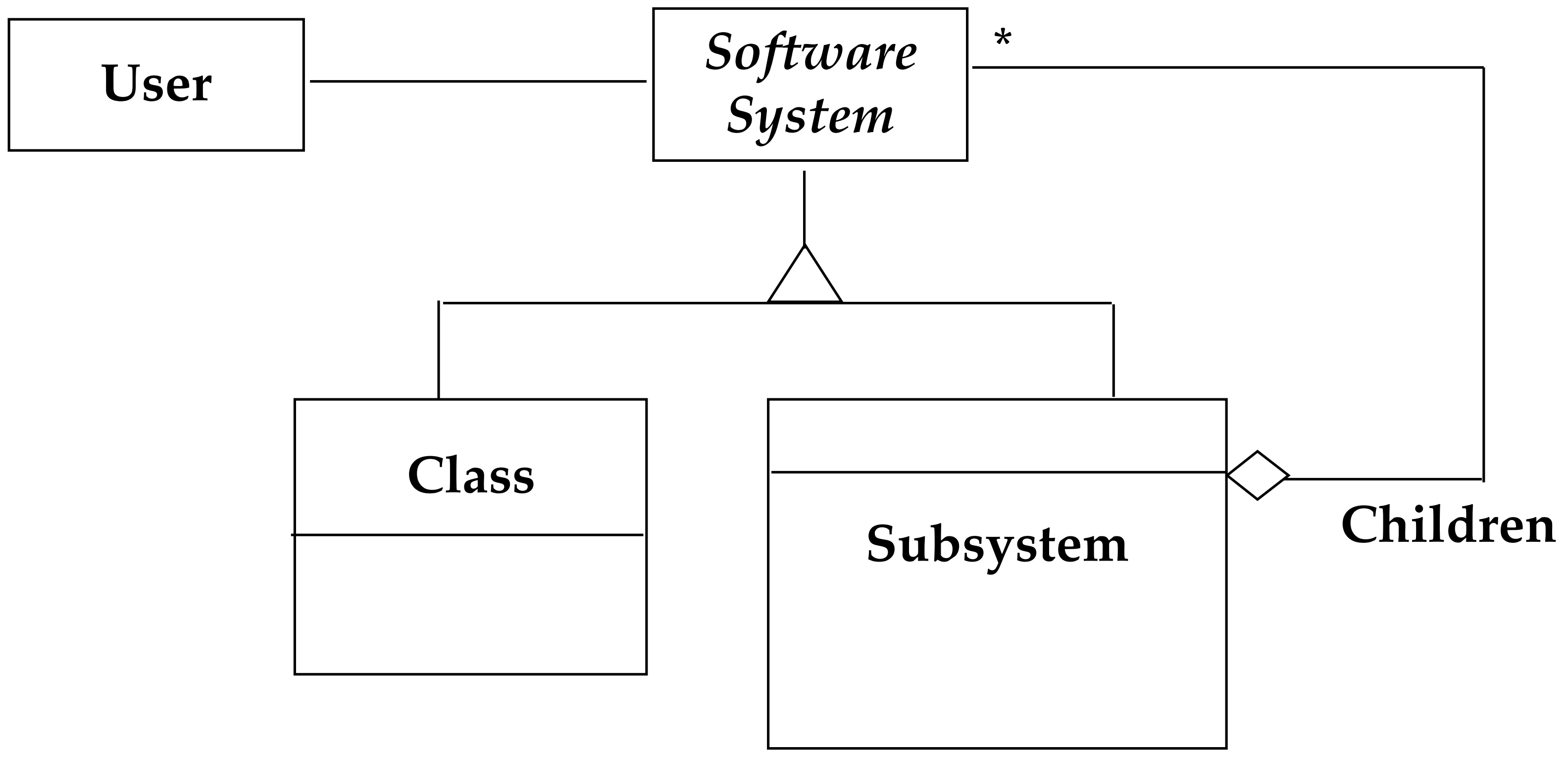
- Composite Pattern:
  - Models trees with dynamic width and dynamic depth
- Adapter Pattern:
  - Interface to reality
- Bridge Pattern:
  - Interface to reality and preparation for the future
- Facade Pattern:
  - Interface to a subsystem, hiding the internals.

[redacted]

Seat Implementation

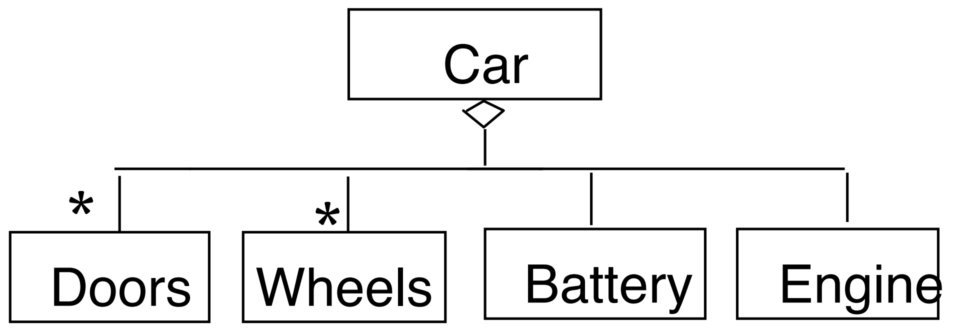
```
public interface SeatImplementation {
  public int GetPosition();
  public void SetPosition(int newPosition);
}
public class Stubcode implements SeatImplementation {
  public int GetPosition() {
    // stub code for GetPosition
  }
  ...
}
public class AimSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the AIM simulation system
  }
  ….
}
public class SARTSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the SART seat simulator
 }
  ...
}
```

Solution: Modeling a Software System with a Composite Pattern

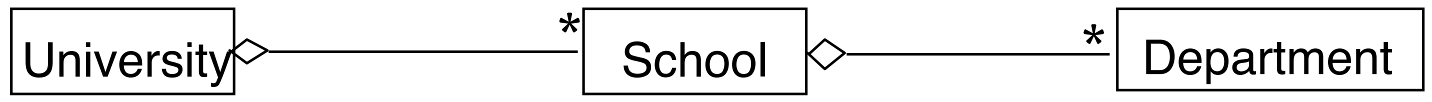


The Composite Patterns models dynamic aggregates

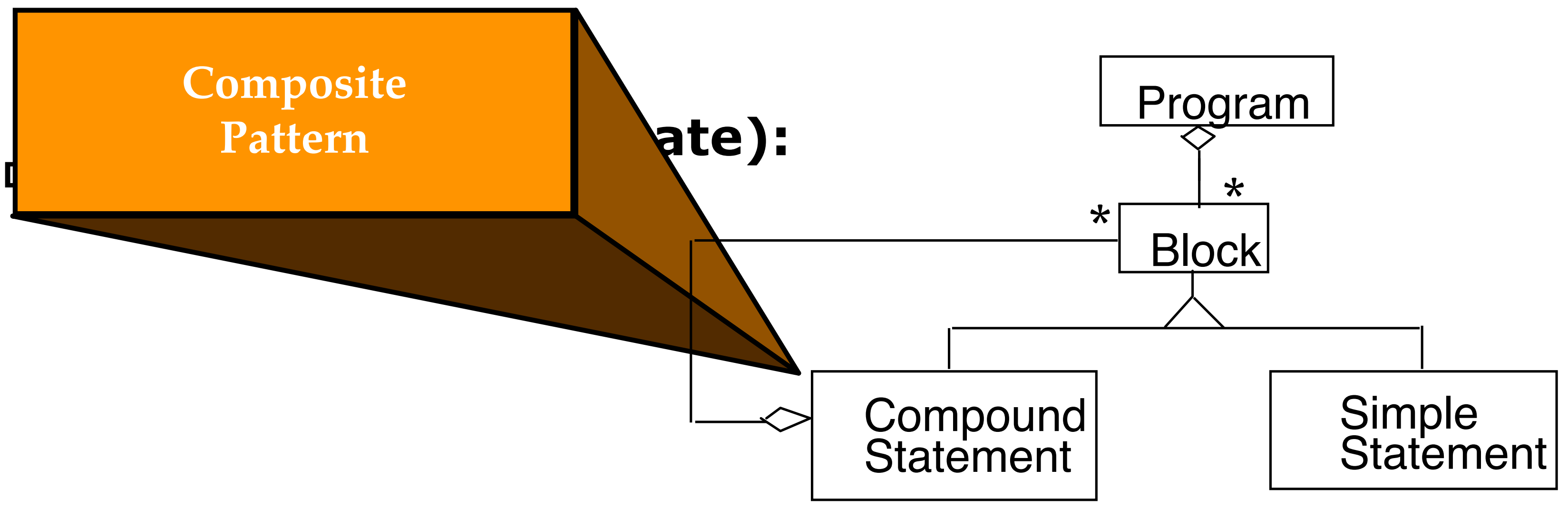
Fixed Structure:



Organization Chart (variable aggregate):



ate):



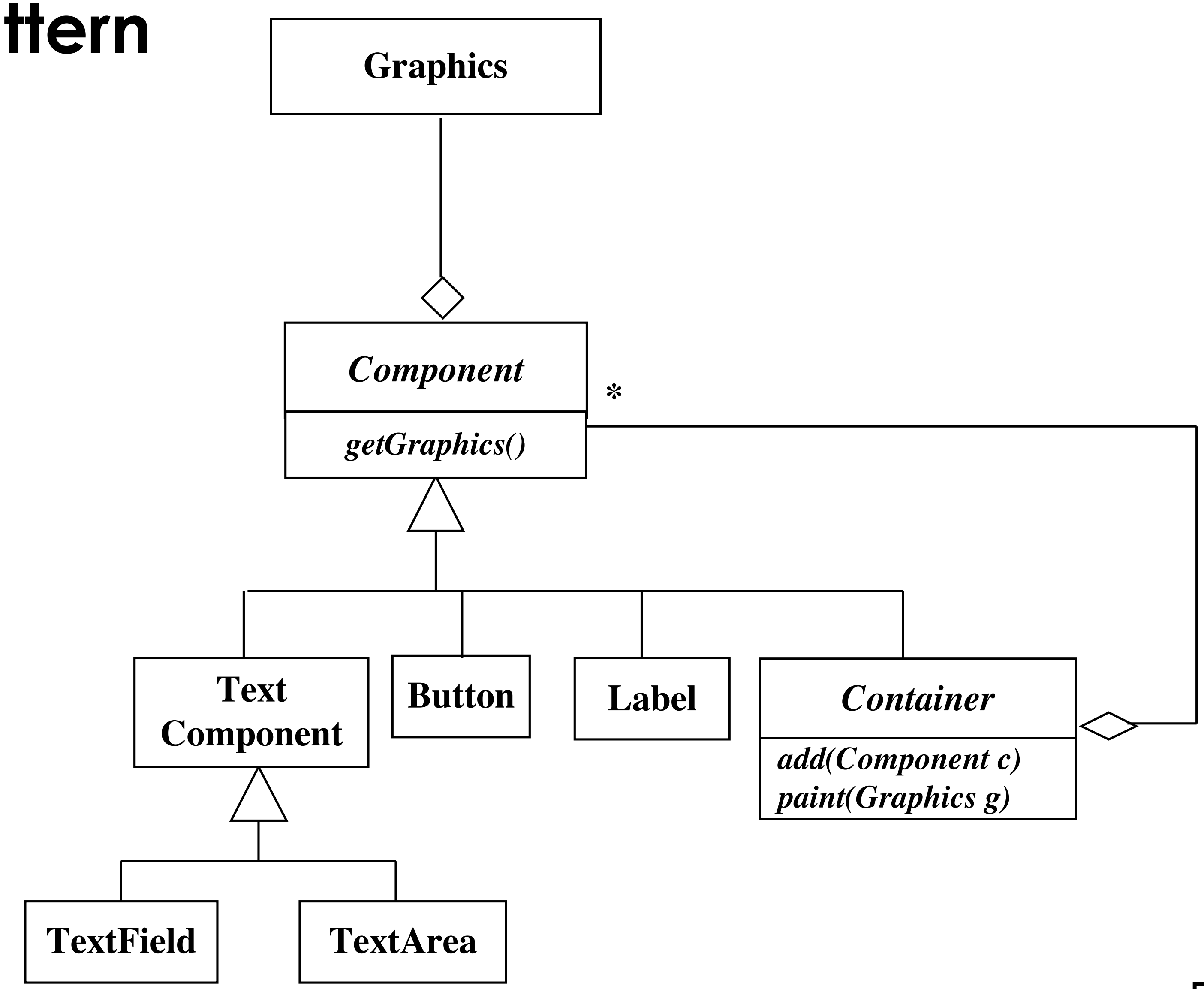
Ideal Subsystem Structure

- The ideal structure of a subsystem consists of
  - An interface object
  - A set of application domain objects (entity objects) modeling real entities or existing systems
    - Some of the application domain objects are interfaces to existing systems
  - One or more control objects

Additional Design Heuristics

- Never use implementation inheritance, always use interface inheritance
- A subclass should never hide operations implemented in a superclass
- If you are tempted to use implementation inheritance, use delegation instead

Java‘s AWT library can be modeled with the component pattern



Notation used in the Design Patterns Book

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995
- Based on OMT Notation (a precursor to UML)
- Notational differences between the notation used by Gamma et al. and UML. In Gamma et al:
  - Attributes come after the Operations
  - Associations are called acquaintances
  - Multiplicities are shown as solid circles ●
  - Dashed line : Instantiation Assocation (Class can instantiate objects of associated class) (In UML it denotes a dependency)
  - UML Note is called Dogear box (connected by dashed line to class operation): Pseudo-code implementation of operation

Paradigms

Paradigms are like rules

They structure the environment and make them understandable

Information that does not fit into the paradigm is invisible.

Patterns are a special case of paradigms