Design Patterns

Bernd Brügge
Technische Universität München
Lehrstuhl für Angewandte Softwaretechnik

18 January 2002

Odds and Ends

- Additional Literature references
- Mid-Term exam
- Software testers wanted

Additional References

- This lecture:
  - **E. Gamma, R. Helm, R. Johnson, J. Vlissides**
    Design Patterns
    Addison-Wesley, Reading, MA, 1994.
    ISBN 0-201-63361-2
- Previous Lecture (topic data management, mapping class diagrams into relational databases):
  - **M. Blaha & W. Premerlani**
    Object-Oriented Modeling and Design for Database Applications
    Prentice Hall, Upper Saddle River, NJ
    ISBN 0-13-123829-9

Mid-term Exam results

- Maximum number of points: 120
- Grading Scale
  - **1,0 >= 115**
  - **1,3 >= 110**
  - **1,7 >= 100**
  - **2,0 >= 95**
  - **2,3 >= 90**
  - **2,7 >= 85**
  - **3,0 >= 80**
  - **3,3 >= 75**
  - **3,7 >= 70**
  - **4,0 >= 60**
  - **4,3 < 60**

- **Performance**

1 Student	**1,3**
6 Students	**1,7**
4 Students	**2,3**
1 Student	**3,0**
2 Students	**4,0**
4 Students	**Fail**

- Your graded exam can be picked at Allen Dutoit‘s office (H-1 1207)
  - **Right after class (from 12:00-13:00pm)**
- Sample solutions available under
  - **http://tramp.globalse.org/doc/presentations/midterm_solutions.pdf**
- Allen is available for questions about the exam
  - **His office hours Tuesday 13:00 to 14:00**

Accdenture looks for Software Tester

- For the test phase of the project described by Frank Mang in his talk yesterday, Accenture looks for students as system testers
- Opportunity to get some insight into the activity of IT consulting.
- Flexible work schedule
- Good payment
- Contact: sabine.freser-specht@accenture.com
- Mobile phone: 0175/ 57-68805

Outline of the next two Lectures

- 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
  - **Proxy**
  - **Observer**
  - **Abstract Factory**
  - **Builder**

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**
  - **Try to identify layers and partitions**
- Object Design
  - **Find additional objects by applying implementation domain knowledge**

Another Source for Finding Objects : Design Patterns

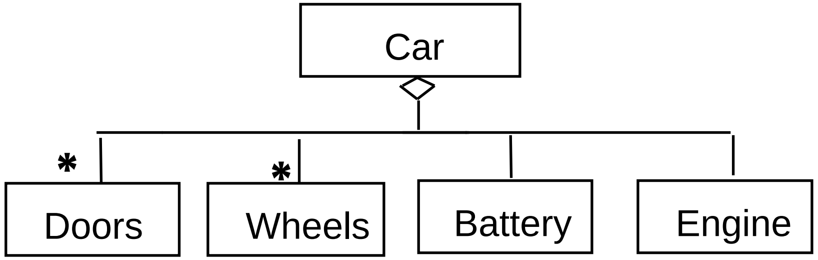
- Observation [Gamma et al 95]:
  - **Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's.**
- There is a need for reusable and flexible designs
- Design knowledge complements application domain knowledge and implementation domain knowledge.
- What are Design Patterns?
  - **A design 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 the this solution a million times over, without ever doing it the same twice**

Design Patterns Notation

- 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 (* ●**
  - **Inheritance shown as triangle**
  - **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**

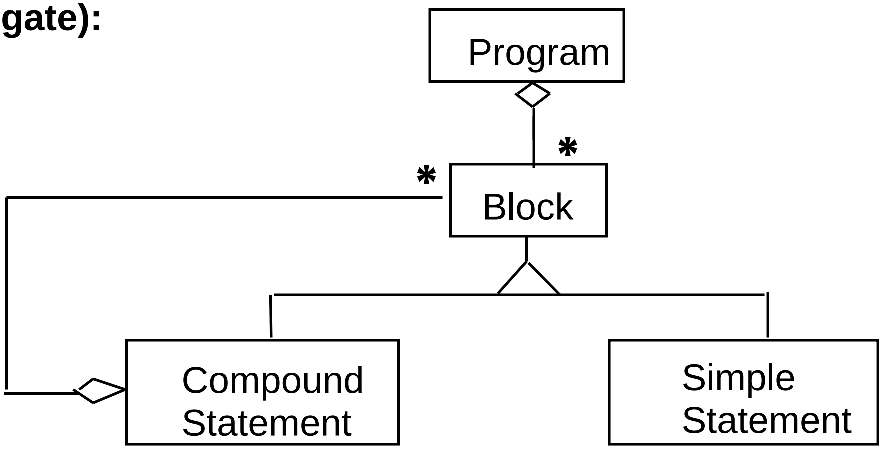
Review: Modeling Typical Aggregations

Fixed Structure:



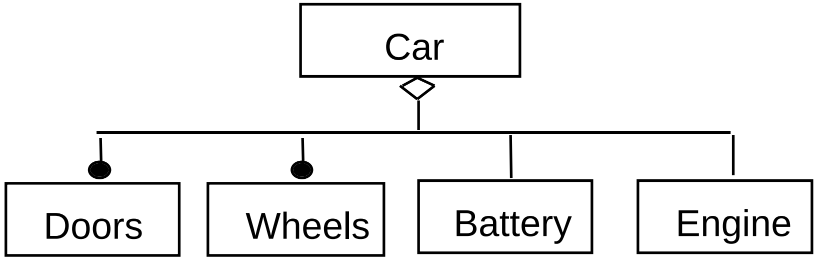
Organization Chart (variable aggregate):

Dynamic tree (recursive aggregate):



Review: Modeling Typical Aggregations (in OMT Notation)

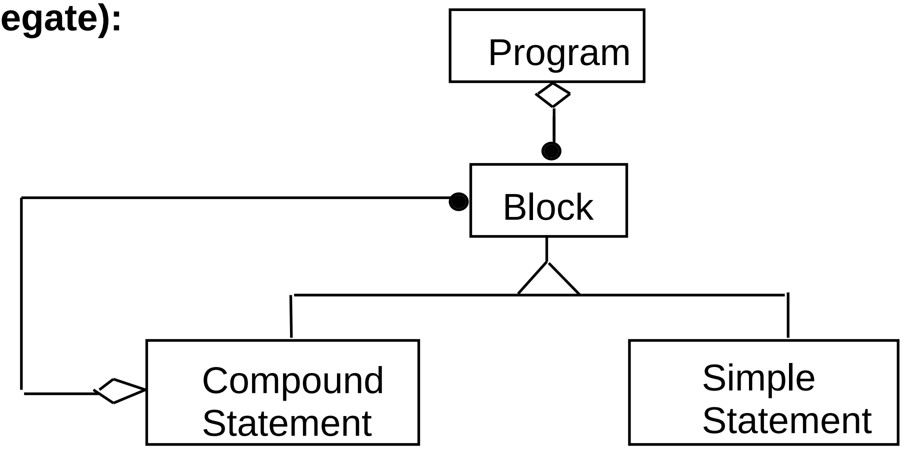
Fixed Structure:


Car
Doors
Wheels
Battery
Engine

Organization Chart (variable aggregate):

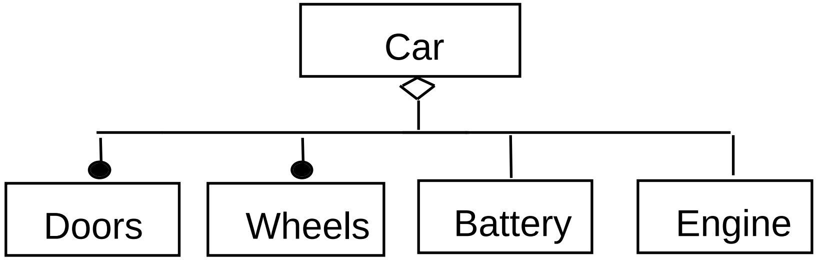

University
School
Department

Dynamic tree (recursive aggregate):

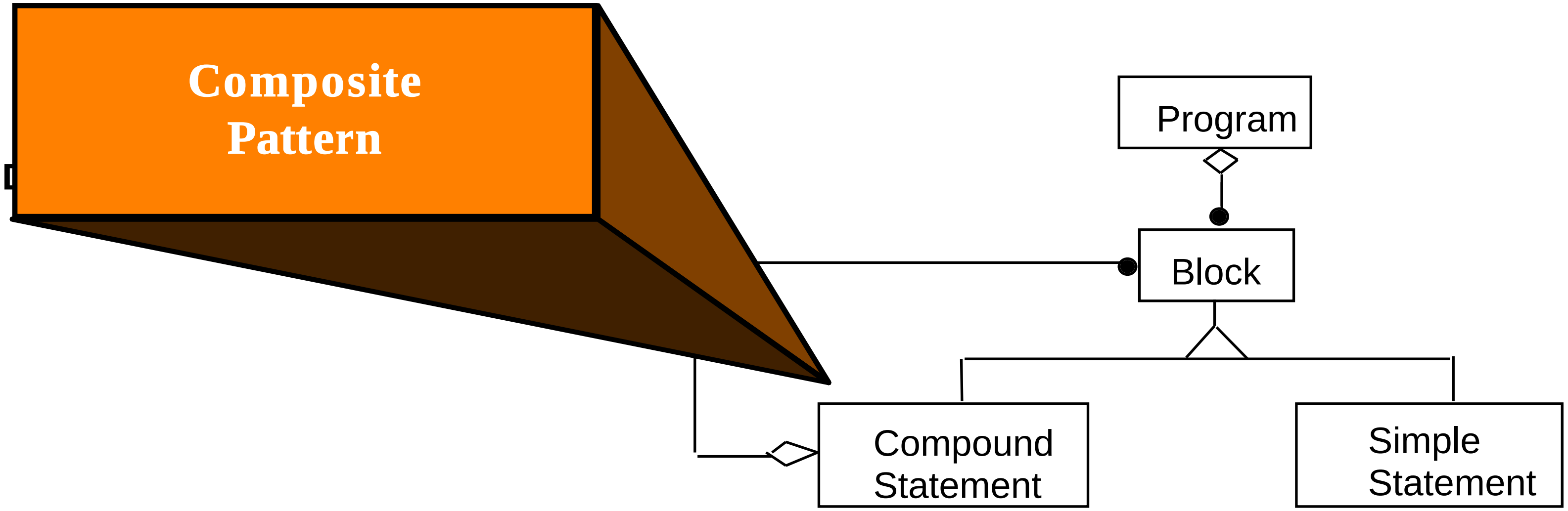

Program
Block
Compound Statement
Simple Statement

Review: Modeling Typical Aggregations

Fixed Structure:

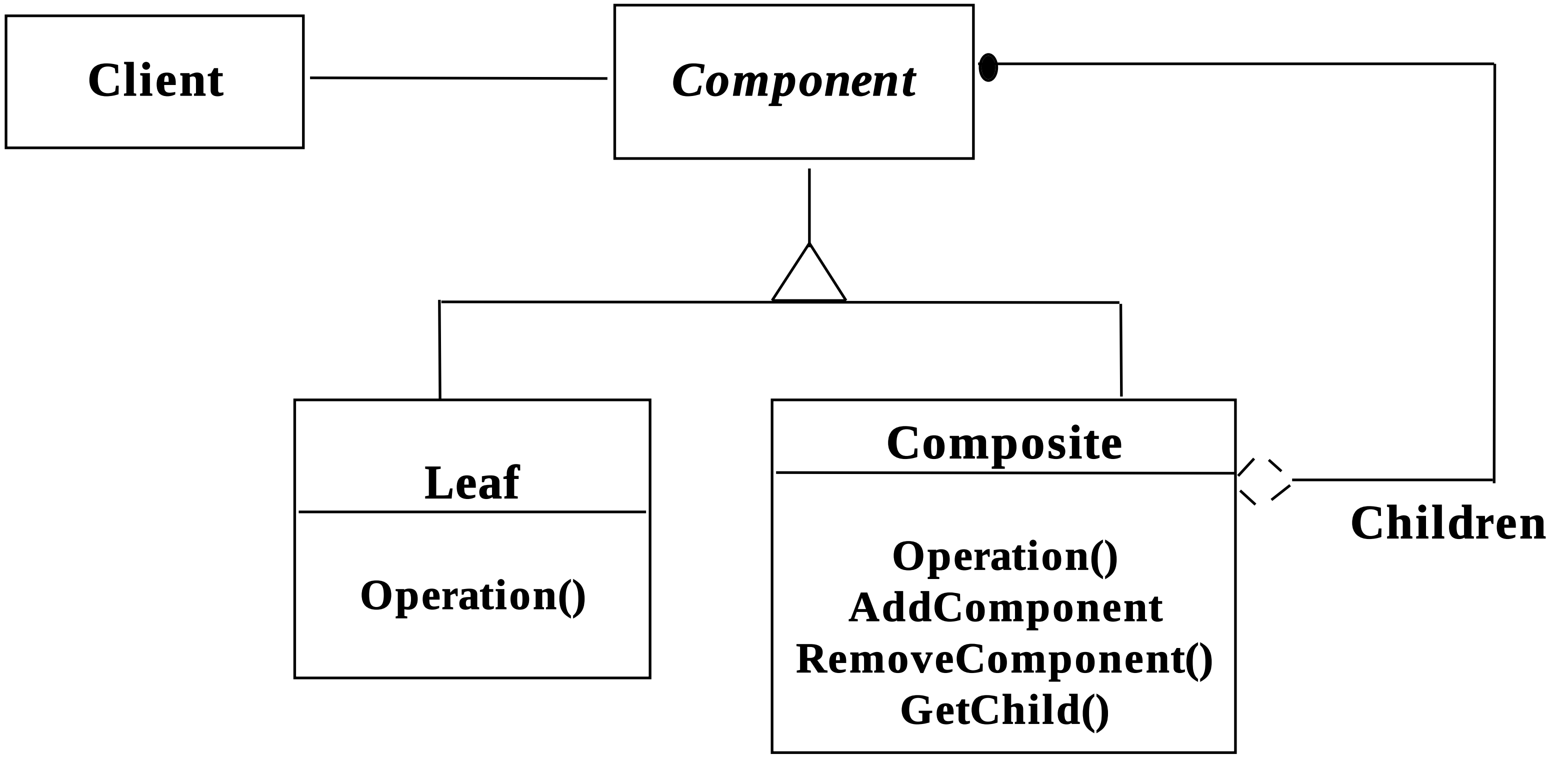


Organization Chart (variable aggregate):



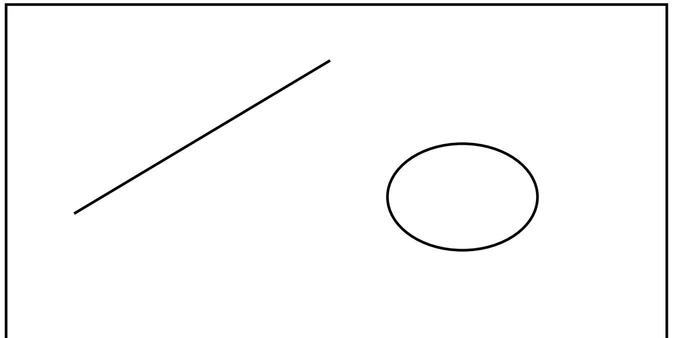
Composite Pattern

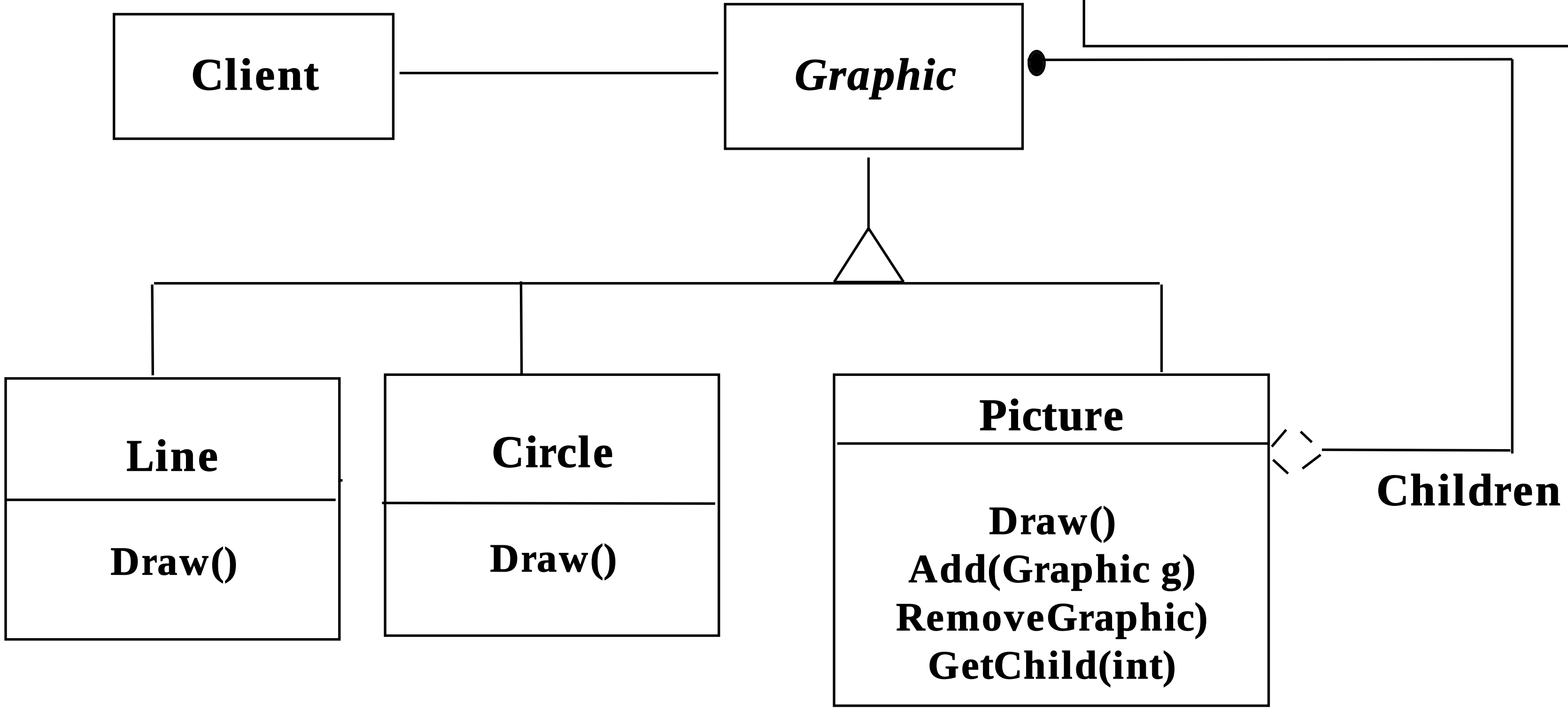
- Models tree structures that represent part-whole hierarchies with arbitrary depth and width.
- The Composite Pattern lets client treat individual objects and compositions of these objects uniformly



Many Graphic Applications use Composite Patterns

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





Client

Graphic

Line

Draw()

Circle

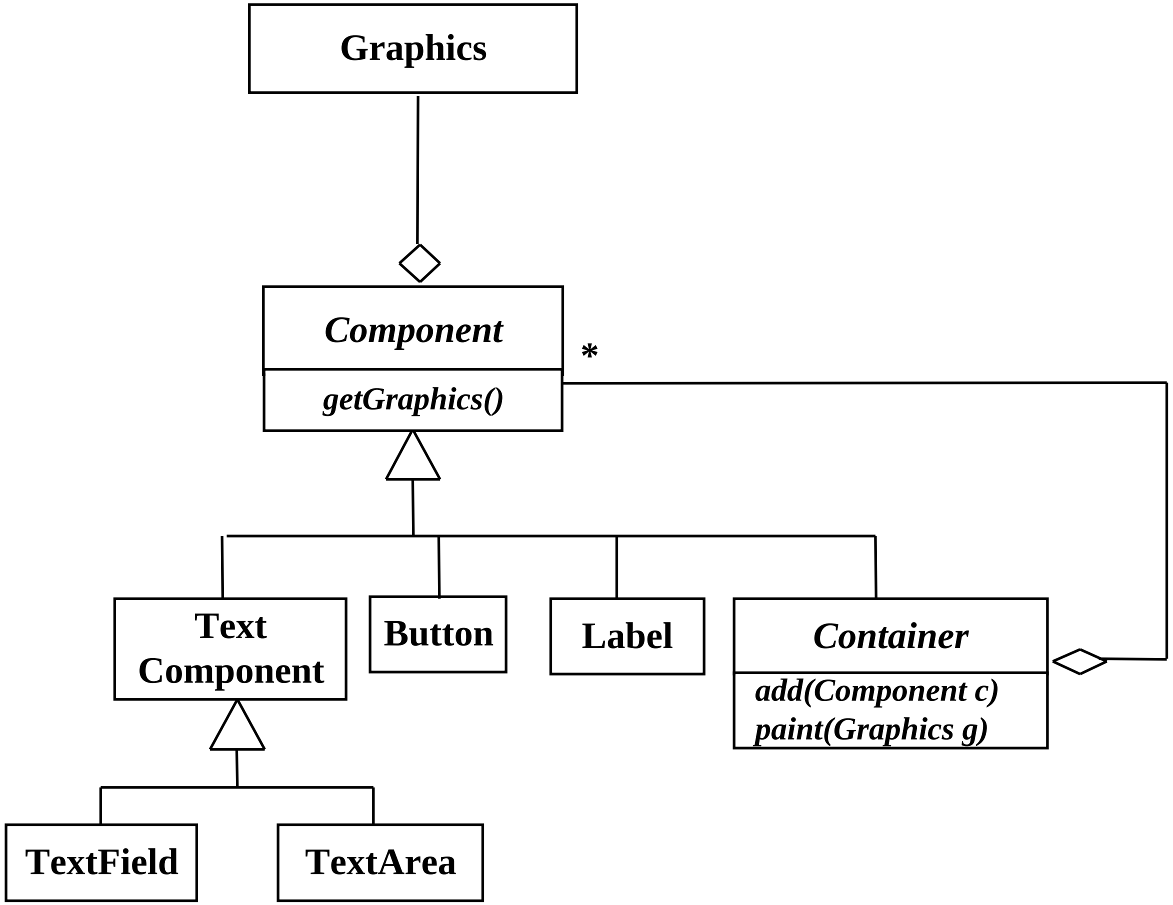
Draw()

Picture

Draw()
Add(Graphic g)
RemoveGraphic)
GetChild(int)

Children

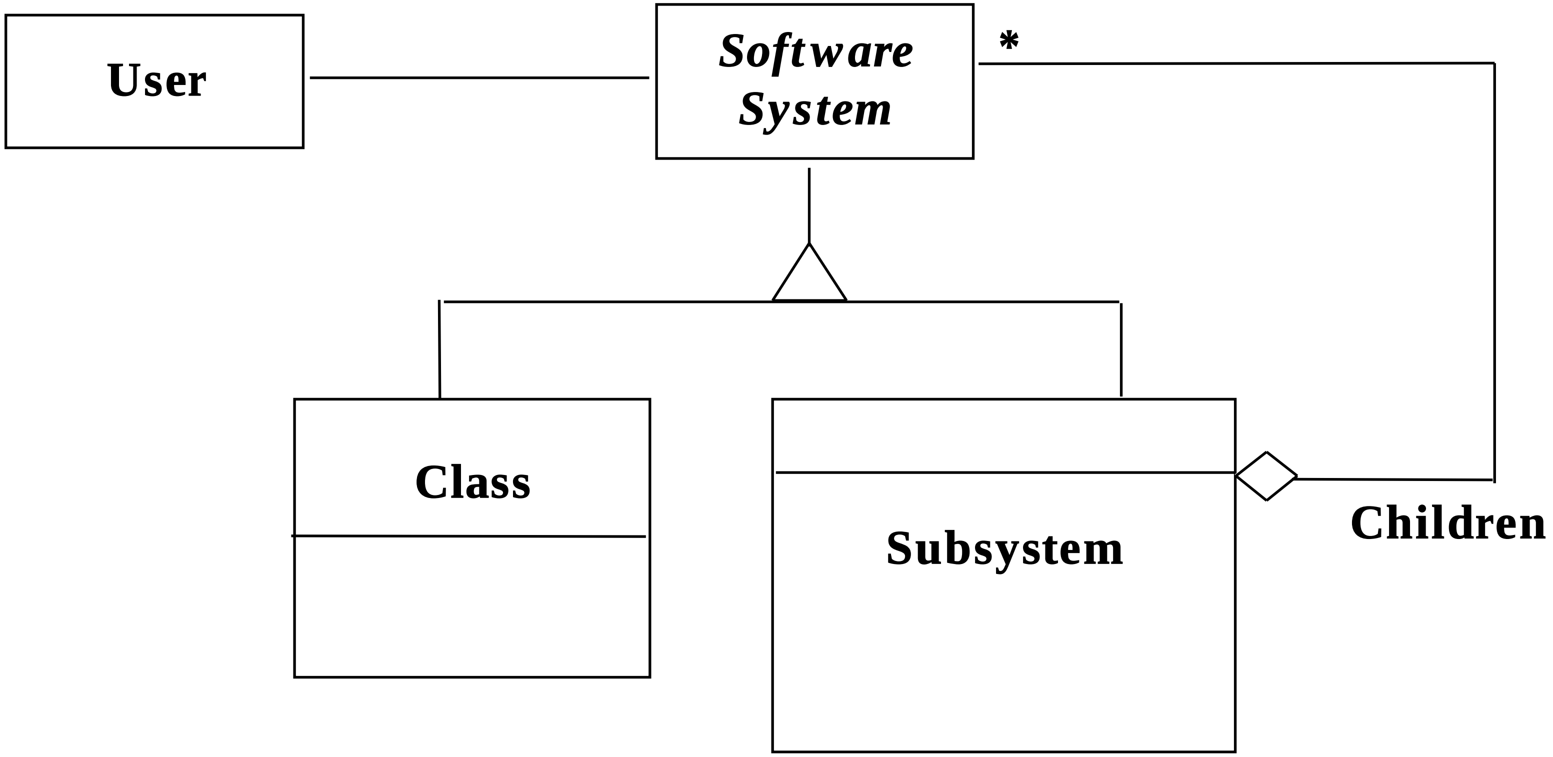
Java‘s AWT library can be modeled with the component pattern



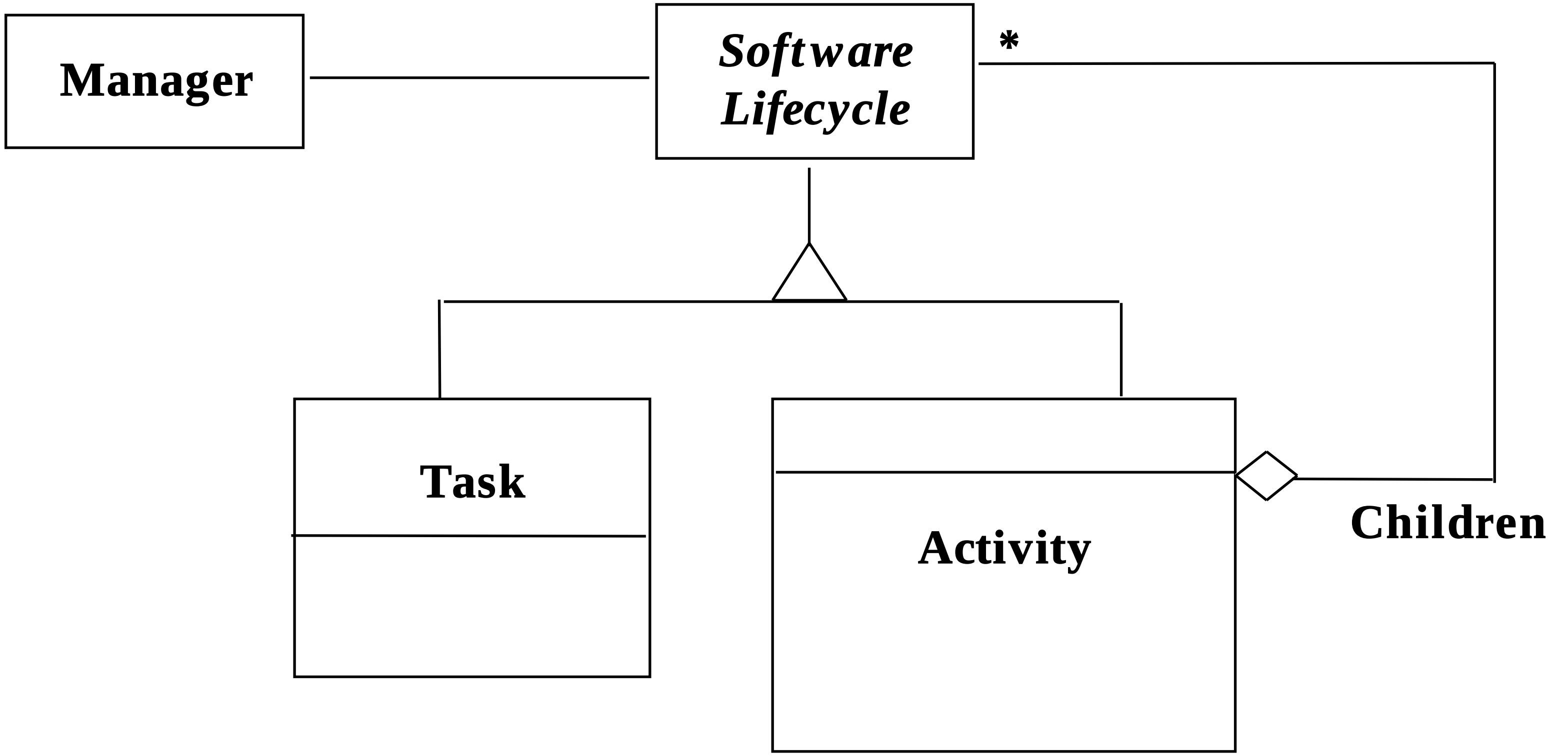
We can also model aspects of Software Development with a Composite Pattern

- Software System:
  - **Definition: A software system consists of subsystems which are either other subsystems or collection of classes**
  - **Composite: Subsystem (A software system consists of subsystems which consists of subsystems , which consists of subsystems, which...)**
  - **Leaf node: Class**
- Software Lifecycle:
  - **Definition: The software lifecycle consists of a set of development activities which are either other actitivies or collection of tasks**
  - **Composite: Activity (The software lifecycle consists of activities which consist of activities, which consist of activities, which....)**
  - **Leaf node: Task**

Modeling a Software System with a Composite Pattern



Modeling the Software Lifecycle with a Composite Pattern

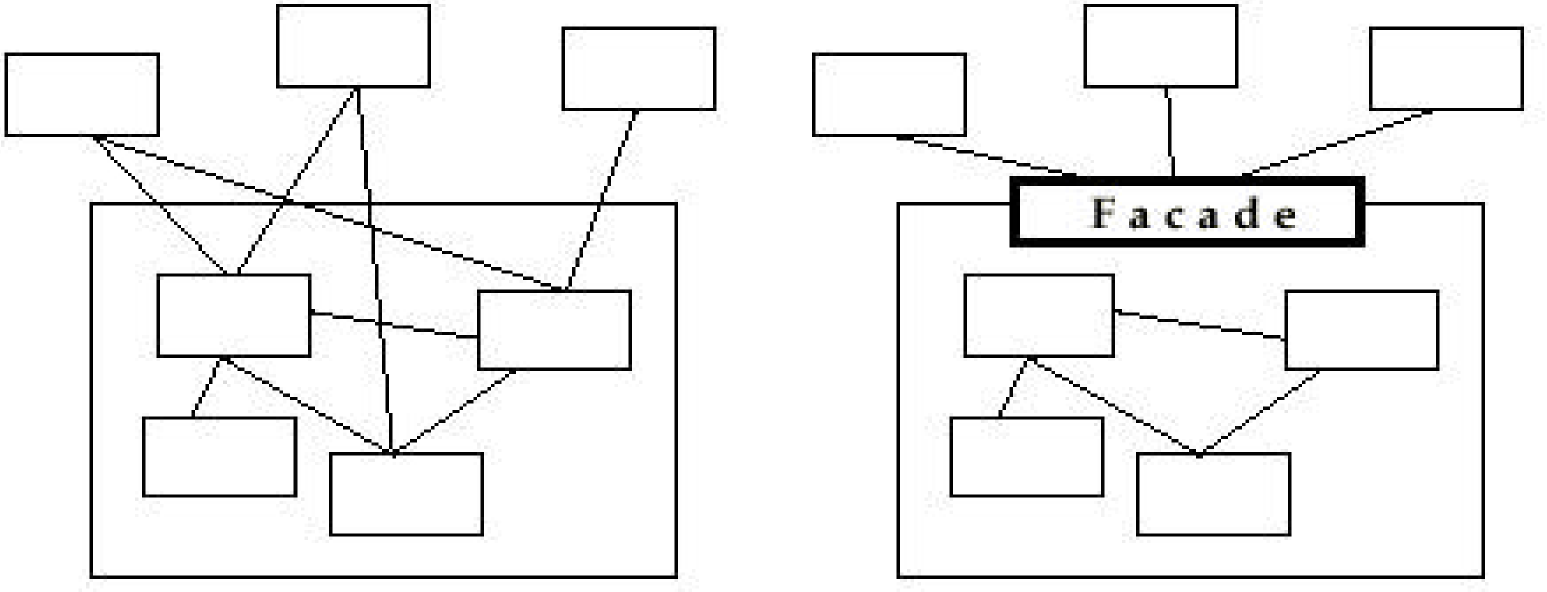


More patterns: Façade, Adapter, Bridge

- 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**
- We can use design patterns to realize these subsystems
- Realization of the Interface Object: ***Facade***
  - **Provides the interface to the subsystem**
- Interface to existing systems: ***Adapter or Bridge***
  - **Provides the interface to existing system (legacy system)**
  - **The existing system is not necessarily object-oriented!**

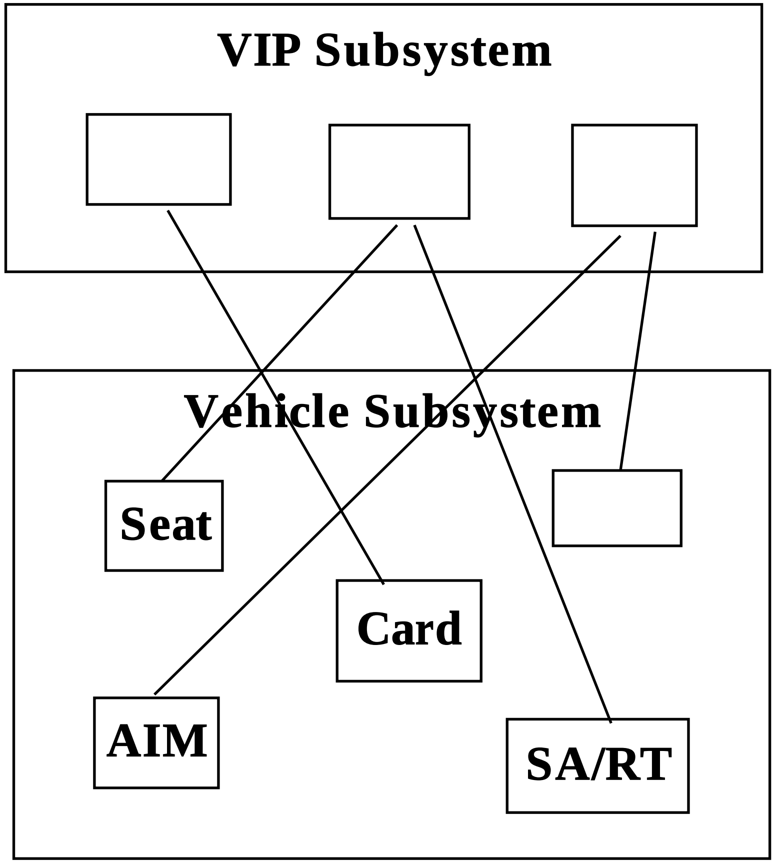
Facade Pattern

- 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 (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture



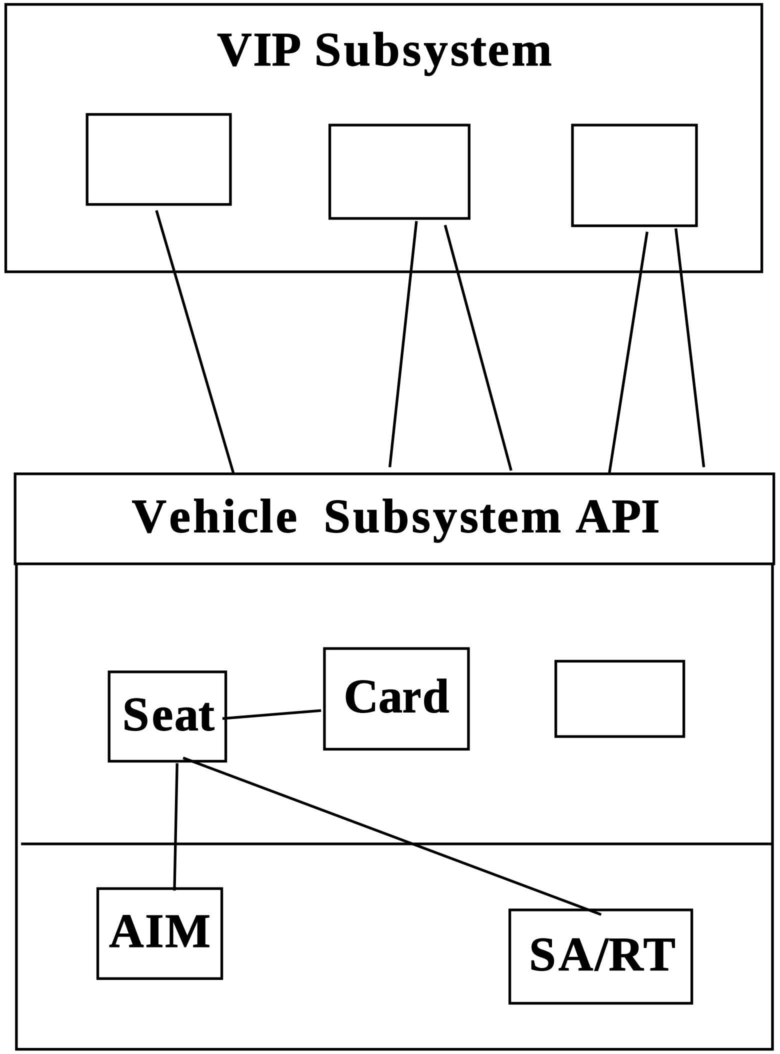
Open vs Closed Architecture

- Open architecture:
  - **Any client can see into the vehicle subsystem and call on any component or class operation at will.**
- Why is this good?
  - **Efficiency**
- Why is this bad?
  - **Can't expect the caller to understand how the subsystem works or the complex relationships within the subsystem.**
  - **We can be assured that the subsystem will be misused, leading to non-portable code**



Realizing a Closed 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 driver**



Review of goals and some terms

- Before we go to the next design pattern let's review the goal and some terms

Reuse

- Main goal:
  - **Reuse knowledge from previous experience to current problem**
  - **Reuse functionality already available**
- **Composition (also called Black Box Reuse)**
  - **New functionality is obtained by *aggregation***
  - **The new object with more functionality is an aggregation of existing components**
- **Inheritance (also called White-box Reuse)**
  - **New functionality is obtained by *inheritance*.**
- Three ways to get new functionality:
  - **Implementation inheritance**
  - **Interface inheritance**
  - **Delegation**

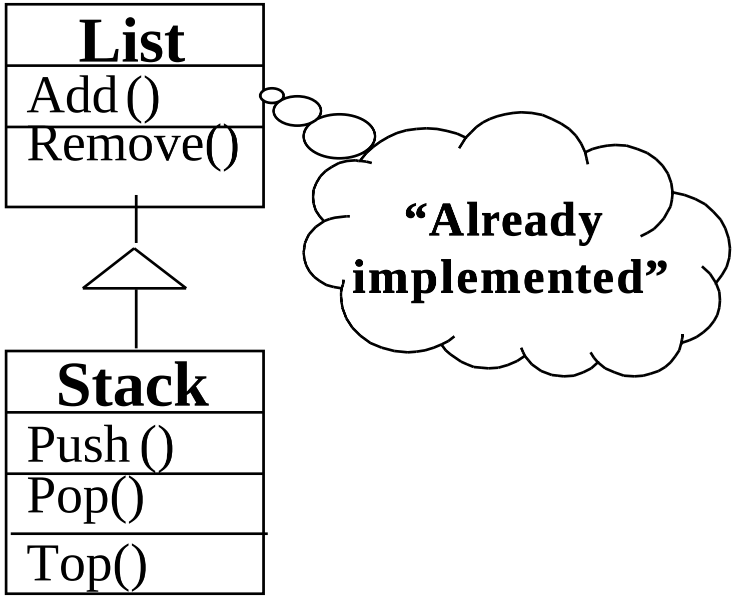
Implementation Inheritance vs Interface Inheritance

- Implementation inheritance
  - **Also called class inheritance**
  - **Goal: Extend an applications' functionality by reusing functionality in parent class**
  - **Inherit from an existing class with some or all operations *already implemented***

- Interface inheritance
  - **Also called subtyping**
  - **Inherit from an abstract class with all operations specified, but not yet implemented**

Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation.
  - Example: I have a **List** class, I need a **Stack** class. How about subclassing the **Stack** class from the **List** class and providing three methods, **Push()** and **Pop(), Top()**?



List
Add()
Remove()

Stack
Push()
Pop()
Top()

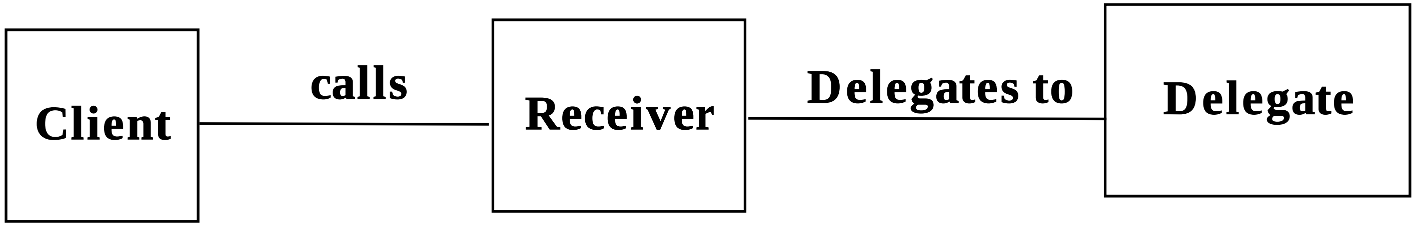
“Already implemented”

- Problem with implementation inheritance:

  Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?

Delegation 1/18/02

- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
  - **A receiving object delegates operations to its delegate.**
  - **The developer can make sure that the receiving object does not allow the client to misuse the delegate object**

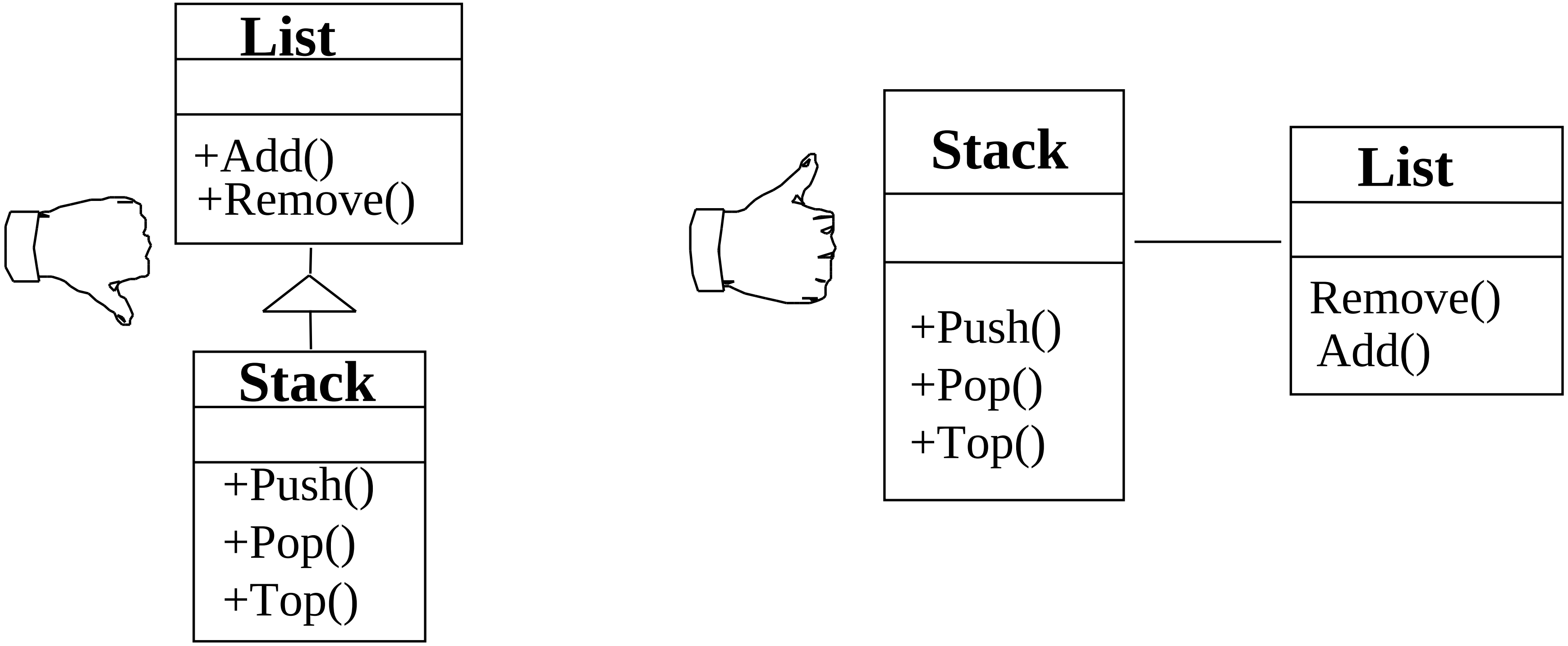


Delegation or Inheritance?

- Delegation
  - **Pro:**
    - **Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)**
  - **Con:**
    - **Inefficiency: Objects are encapsulated.**
- Inheritance
  - **Pro:**
    - **Straightforward to use**
    - **Supported by many programming languages**
    - **Easy to implement new functionality**
  - **Con:**
    - **Inheritance exposes a subclass to the details of its parent class**
    - **Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)**

Delegation instead of Inheritance

- Delegation: Catching an operation and sending it to another object.
- Which solution is better?

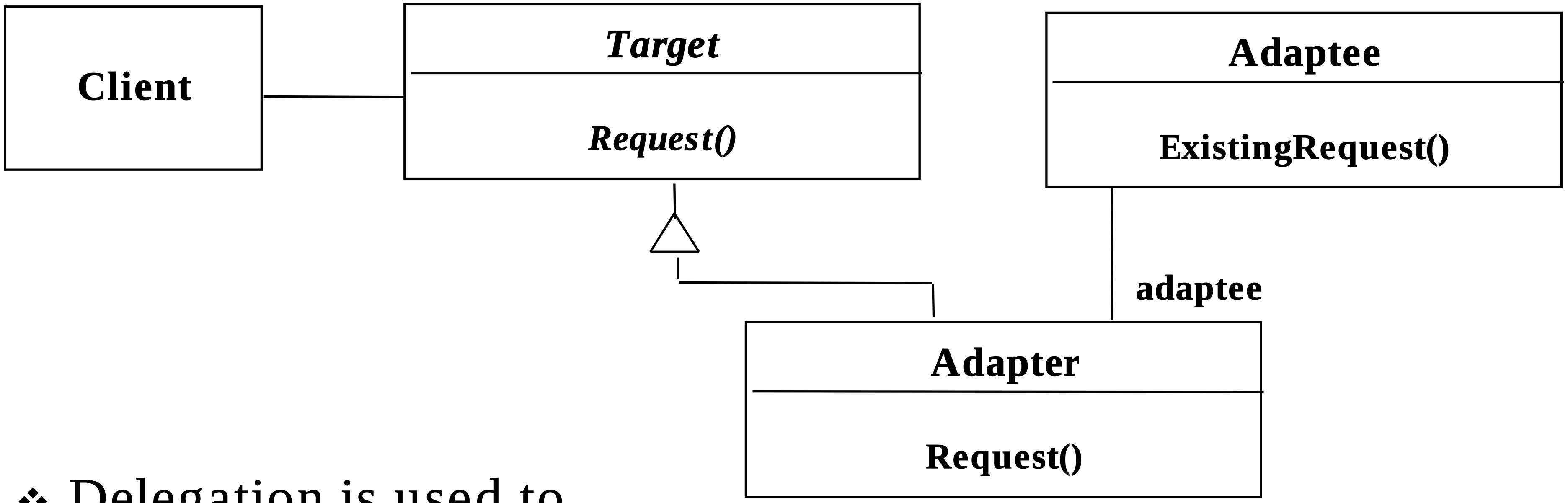


Many design patterns use a combination of inheritance and delegation

Adapter Pattern

- “Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces
- 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**
- Object adapters are much more frequent. We will only cover object adapters (and call them therefore simply adapters)

Adapter pattern



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

Bridge Pattern

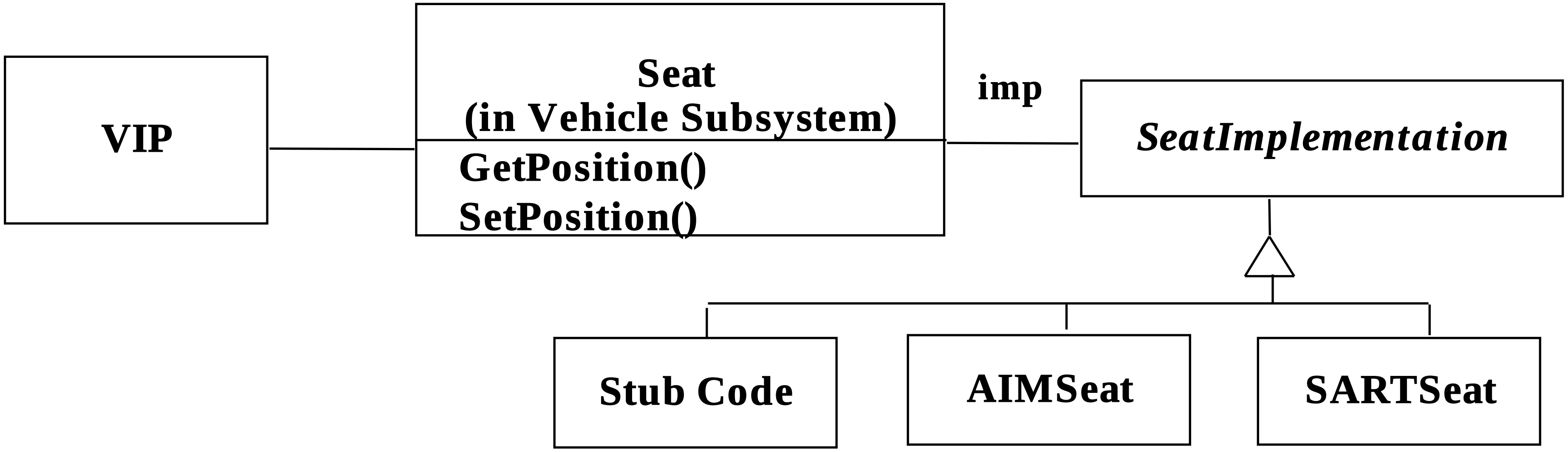
- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”. (From [Gamma et al 1995])

- Also know as a Handle/Body pattern.

- Allows different implementations of an interface to be decided upon dynamically.

Using a Bridge

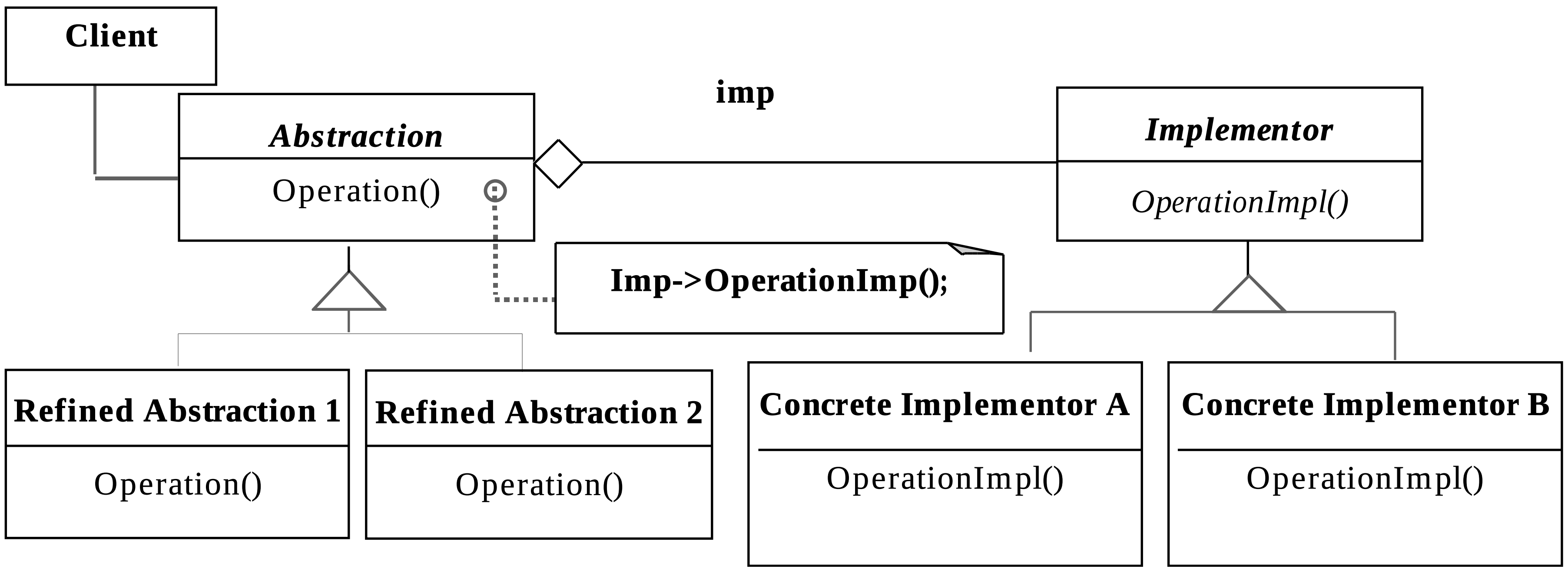
- The bridge pattern is used to provide multiple implementations under the same interface.
- Examples: Interface to a component that is incomplete, not yet known or unavailable during testing
- JAMES Project (WS 97-98): if seat data is required to be read, but the seat is not yet implemented, not yet known or only available by a simulation, provide a bridge:



JAMES Bridge Example Implementation in Java

```
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
 }
  ...
}
```

Bridge Pattern(151)



Client

imp

Abstraction

Operation()

Implementor

OperationImpl()

Imp->OperationImp();

Refined Abstraction 1

Operation()

Refined Abstraction 2

Operation()

Concrete Implementor A

OperationImpl()

Concrete Implementor B

OperationImpl()

Adapter vs Bridge

- Similarities:
  - **Both 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.**

Design Patterns encourage good Design Practice

- A facade pattern should be used by all subsystems in a software system. The façade defines all the services of the subsystem.
  - **The facade will delegate requests to the appropriate components within the subsystem.**
- Adapters should be used to interface to any existing proprietary components.
  - **For example, a smart card software system should provide an adapter for a particular smart card reader and other hardware that it controls and queries.**
- Bridges should be used to interface to a set of objects where the full set is not completely known at analysis or design time.
  - **Bridges should be used when the subsystem must be extended later (extensibility).**

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

Summary

- Composite Pattern:
  - **Models trees with dynamic width and dynamic depth**
- Facade Pattern:
  - **Interface to a Subsystem**
  - **Closed vs Open Architecture**
- Adapter Pattern:
  - **Interface to Reality**
- Bridge Pattern:
  - **Interface Reality and Future**
- Read Design Patterns Book
  - **Learn how to use it as a reference book**

Other Design Patterns

- Creational Patterns
    - **Abstract Factory Pattern (“Device Independence”)**
- Structural Patterns
    - **Proxy (“Location Transparency”)**
- Behavioral Patterns
    - **Command (“Request Encapsulation”, “unlimited undos”)**
    - **Observer (“Publish and Subscribe”)**
    - **Strategy (“Policy vs Mechanism”, “Encapsulate family of algorithms”)**