Object-Oriented Software Engineering

Using UML, Patterns, and Java

Chapter 8, Object Design Introduction to Design Patterns

- 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 helps keep system models simple.

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
  - Extract objects from Use Case scenarios (dynamic model)
- 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

- What are Design Patterns?
  - The recurring aspects of designs are called *design patterns* [Gamma et al 1995].
  - A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context.
  - It 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 twice. Many of them have been systematically documented for all software developers to use.

Studying patterns is an effective way to learn from the experience of others

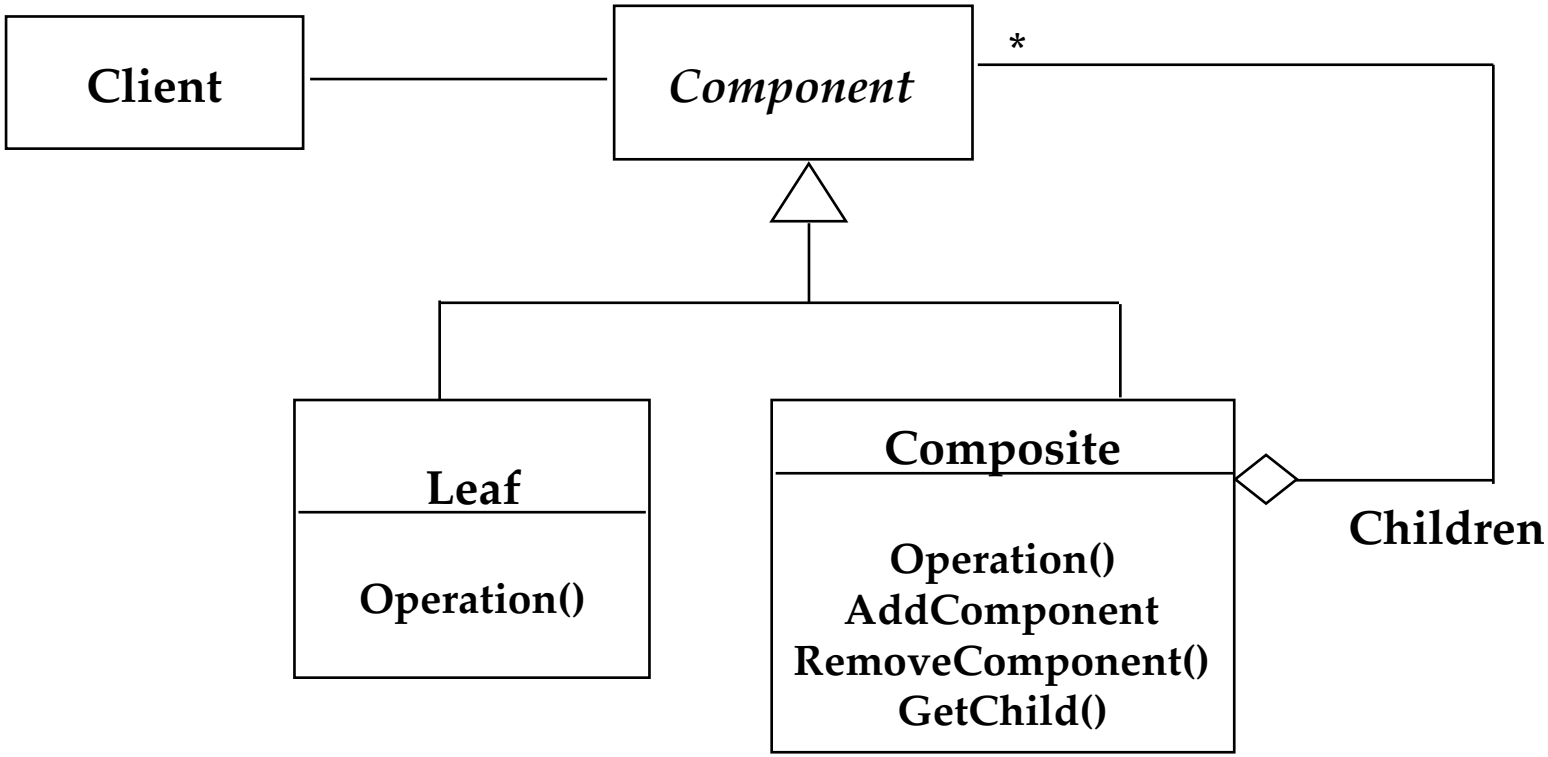
What is common between these definitions?

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

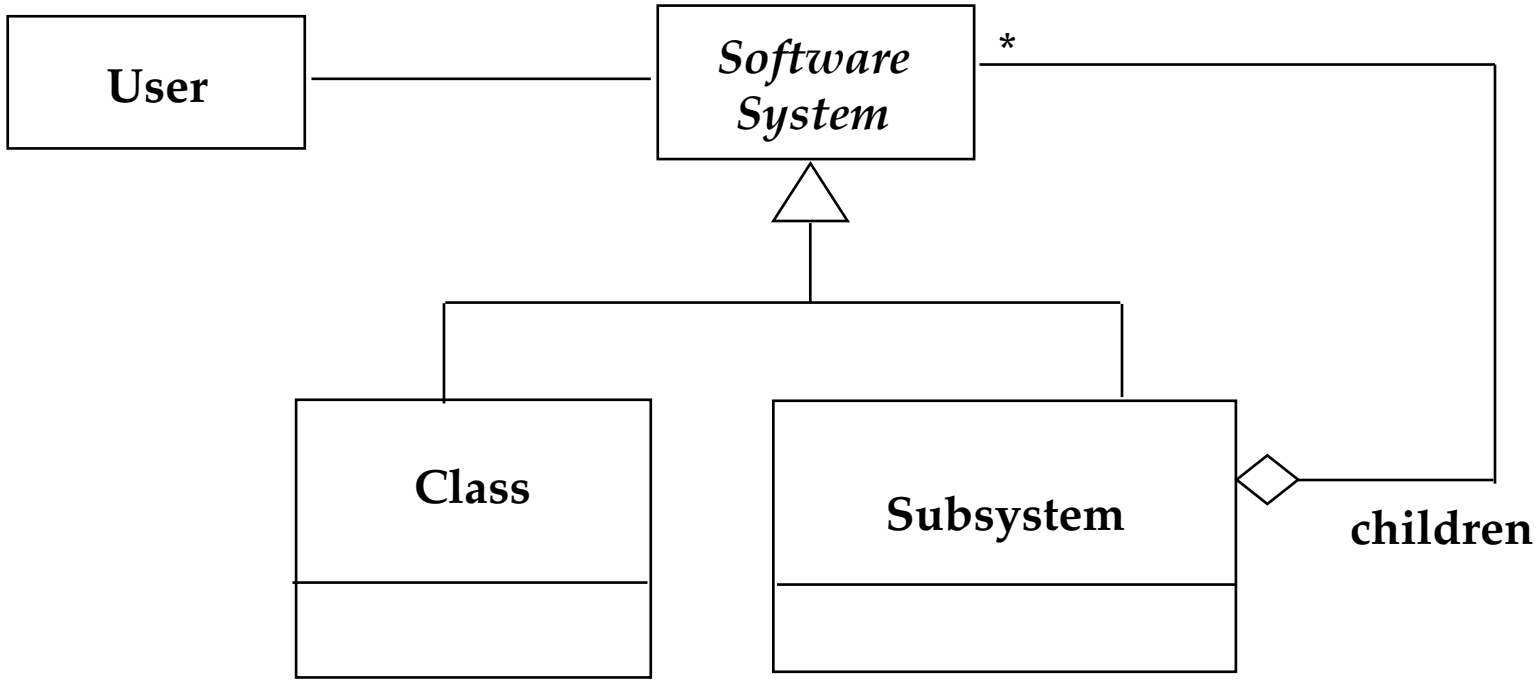
- Definition Software Lifecycle:
  - The software lifecycle consists of a set of development activities which are either other actitivies or collection of tasks

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

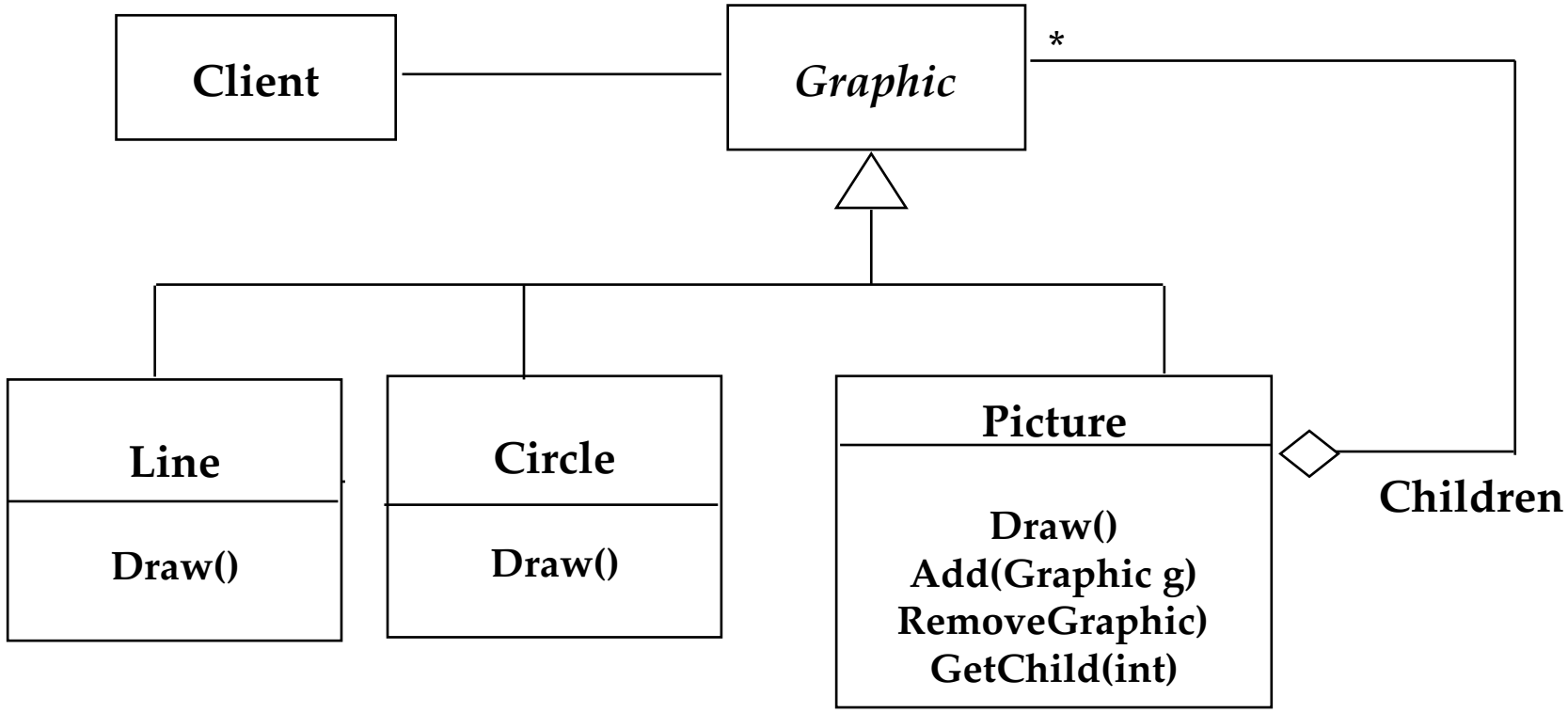


Modeling a Software System with a Composite Pattern



Graphic Applications also use Composite Patterns

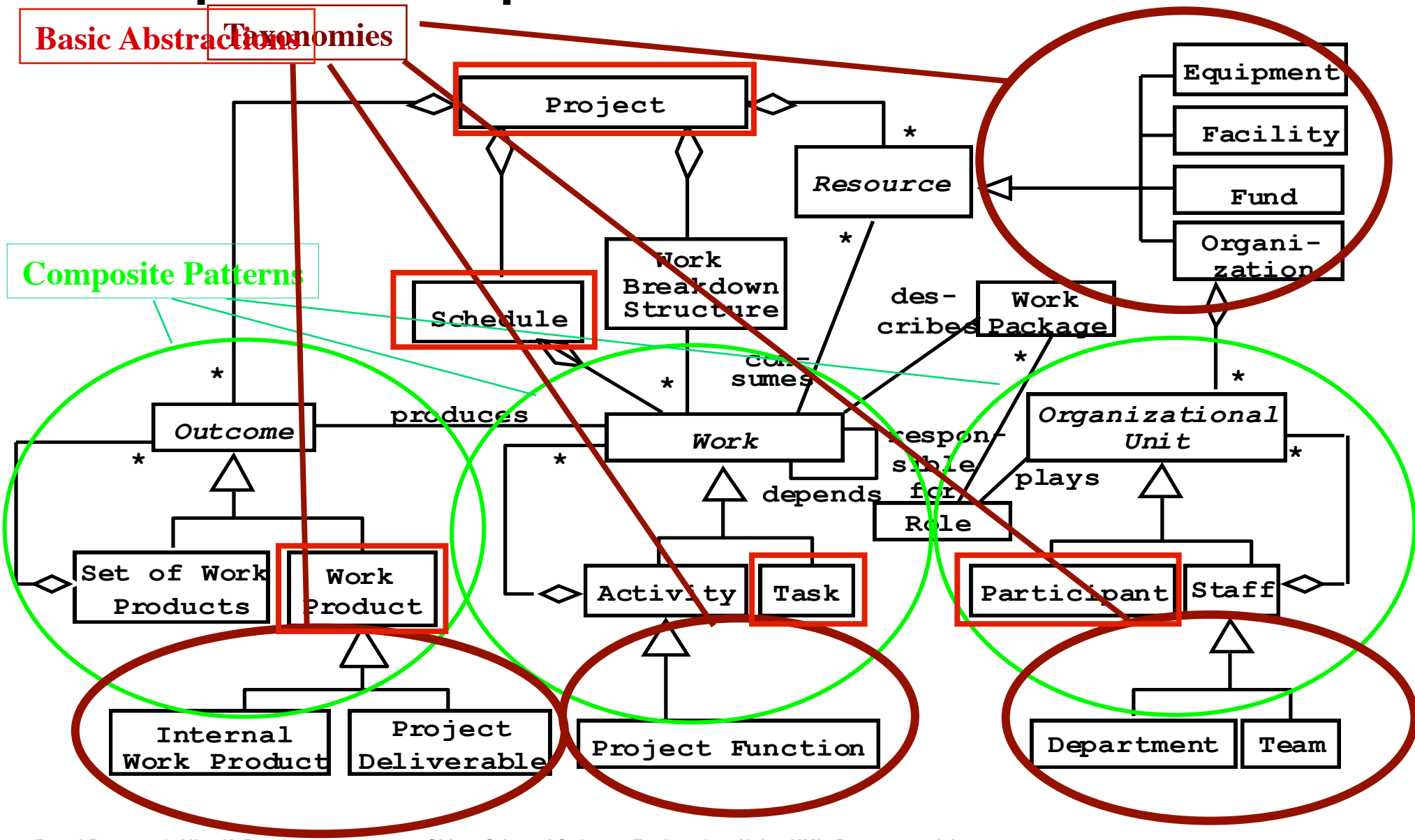
- The *Graphic* Class represents both primitives (Line, Circle) and their containers (Picture)



Reducing the Complexity of Models

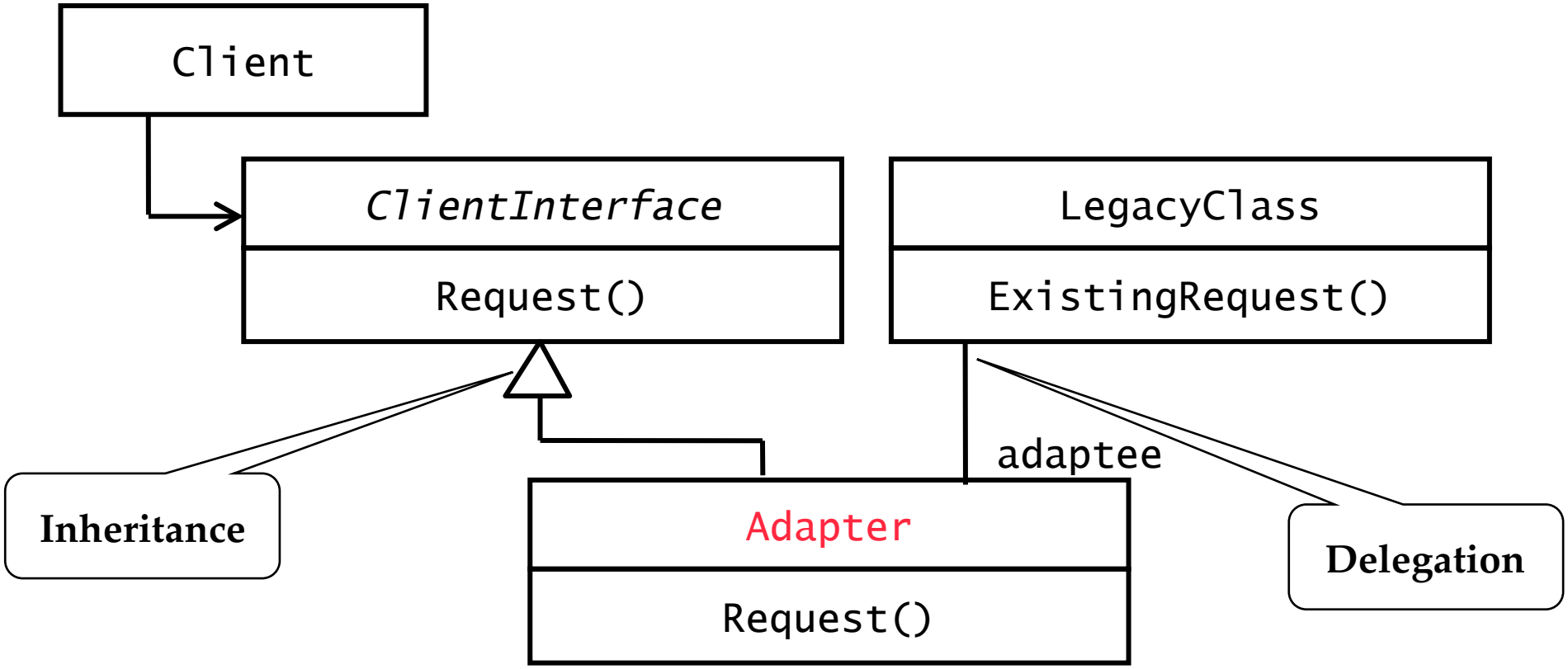
- To communicate a complex model we use navigation and reduction of complexity
  - We do not simply use a picture from the CASE tool and dump it in front of the user
  - The key is to navigate through the model so the user can follow it
- We start with a very simple model
  - Start with the key abstractions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

Example: A Complex Model



Many design patterns use a combination of inheritance and delegation

Adapter Pattern



The adapter pattern uses inheritance as well as delegation:

- **Interface inheritance is used to specify the interface of the Adapter class.**
- **Delegation is used to bind the Adapter and the Adaptee**

Adapter Pattern

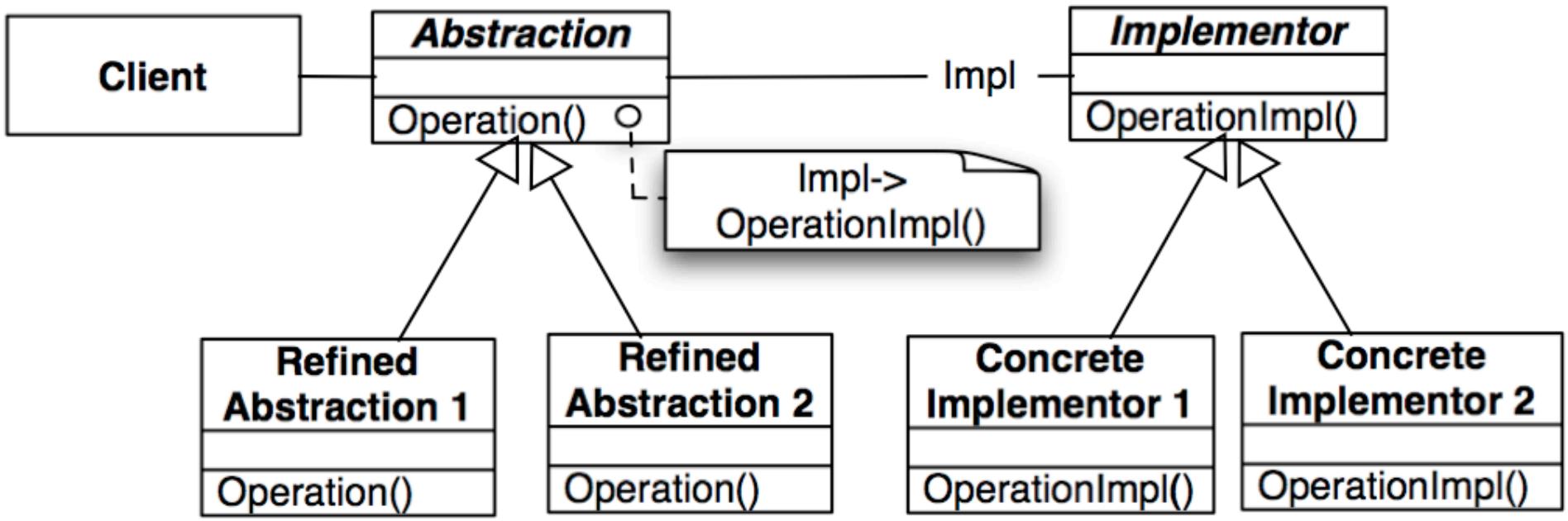
- The adapter pattern lets classes work together that couldn't otherwise because of incompatible interfaces
  - "Convert the interface of a class into another interface expected by a client class."
  - Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- 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 cover only object adapters (and call them adapters).

Bridge Pattern

- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”
  - Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
  - Beyond encapsulation, to insulation
- Also know as a Handle/Body pattern

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

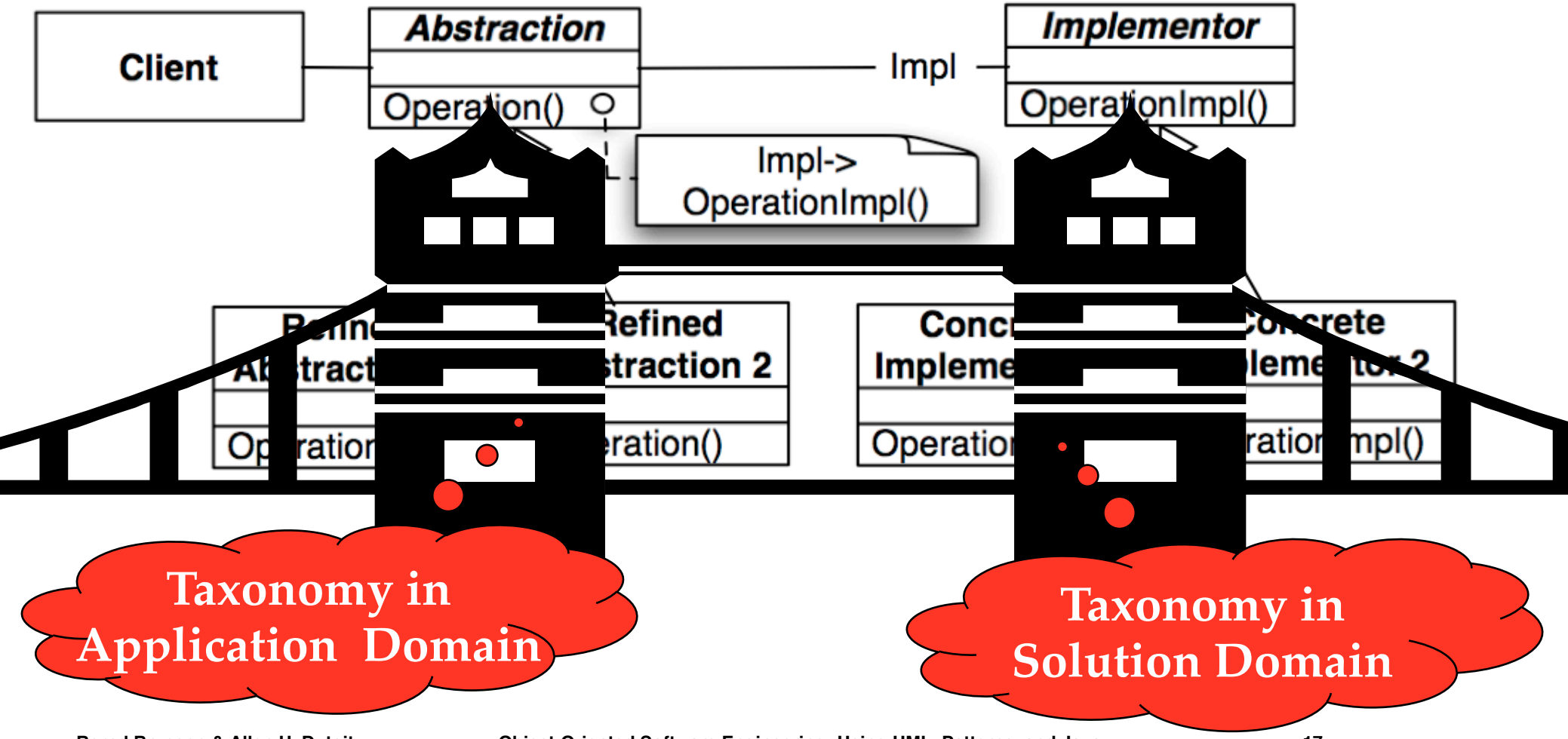
Bridge Pattern



Taxonomy in Application Domain

Taxonomy in Solution Domain

Why the Name Bridge Pattern?



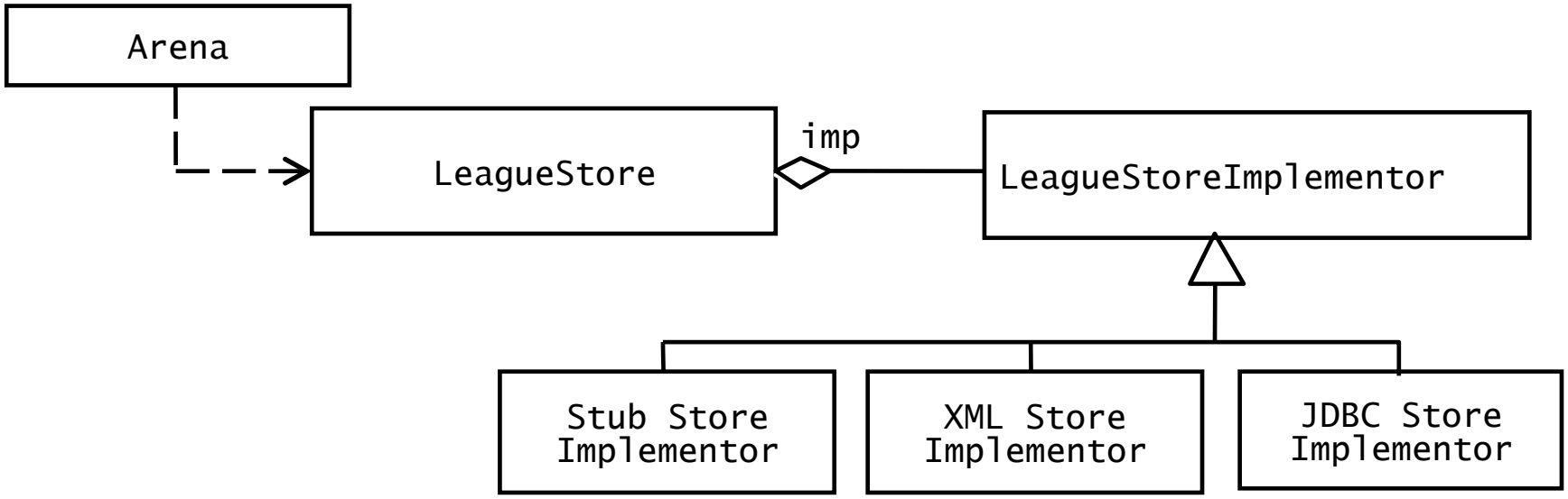
Motivation for the Bridge Pattern

- Decouples an abstraction from its implementation so that the two can vary independently
- This allows to bind one from many different implementations of an interface to a client dynamically
- Design decision that can be realized any time during the runtime of the system
  - However, usually the binding occurs at start up time of the system (e.g. in the constructor of the interface class)

Using a Bridge

- The bridge pattern can be used to provide multiple implementations under the same interface

Example use of the Bridge Pattern: Support multiple Database Vendors

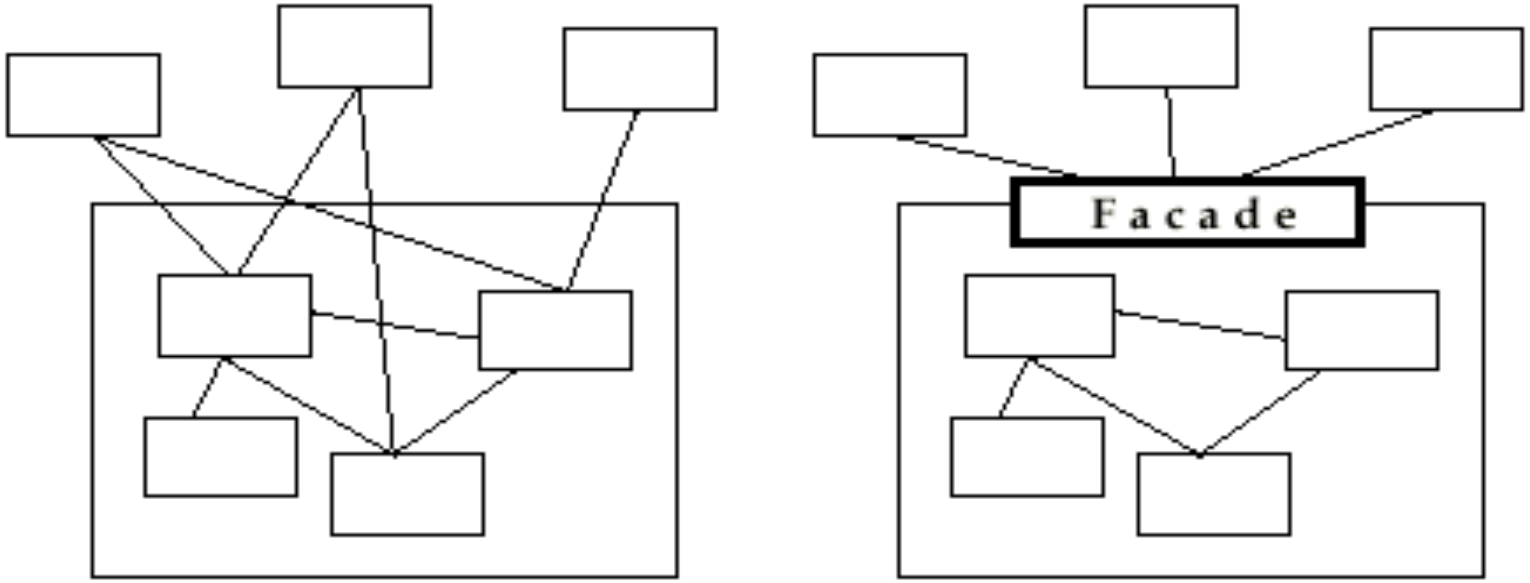


Adapter vs Bridge

- 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).
    - "Inheritance followed by delegation"
  - 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.
    - "Delegation followed by inheritance"

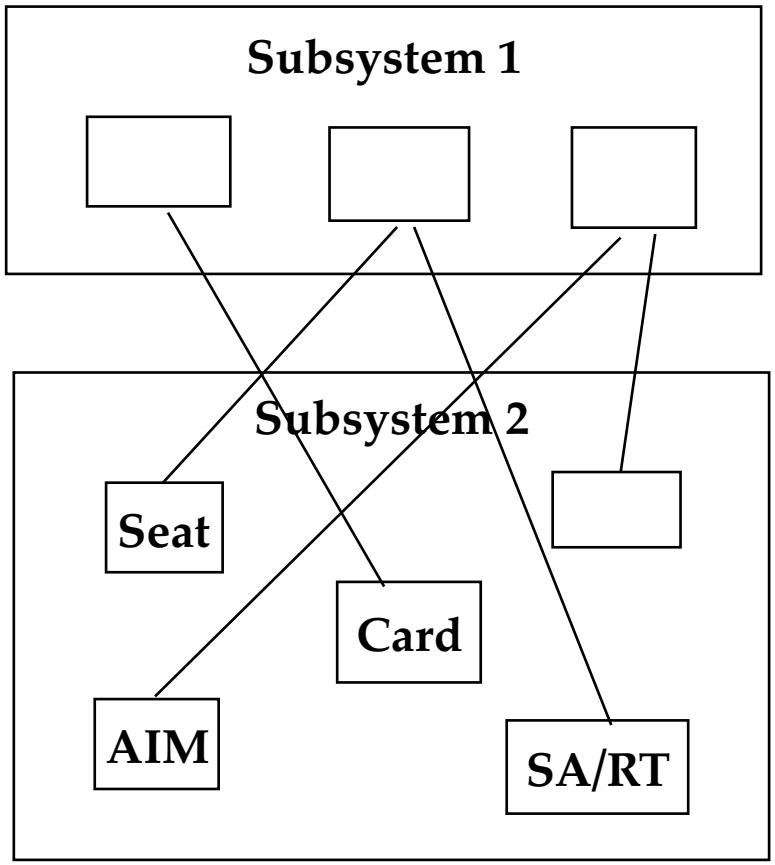
Facade Pattern

- 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)
-



Design Example

- Subsystem 1 can look into the Subsystem 2 (vehicle subsystem) and call on any component or class operation at will.
- This is “Ravioli Design”
- 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



Subsystem Design with 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 this subsystem structure
- 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!

When should you use these Design Patterns?

- A façade should be offered by all subsystems in a software system who a services
  - The façade delegates requests to the appropriate components within the subsystem. The façade usually does not have to be changed, when the components are changed
- The adapter design pattern should be used to interface to existing components
  - Example: A smart card software system should use an adapter for a smart card reader from a specific manufacturer
- The bridge design pattern should be used to interface to a set of objects
  - where the full set of objects is not completely known at analysis or design time.
  - when a subsystem or component must be replaced later after the system has been deployed and client programs use it in the field.

Summary

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