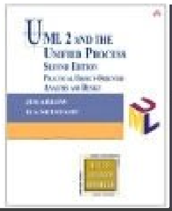
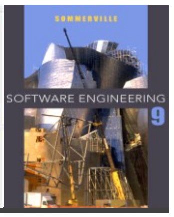
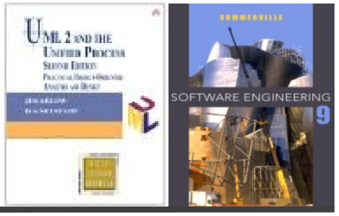



Low-level Design and Implementation Issues

Lecture 8

Outline

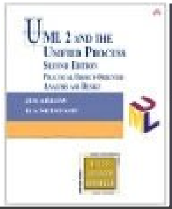
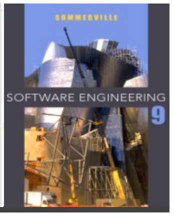


✧ Low-level design

✧ Implementation issues

✧ UML Class Diagram in Design

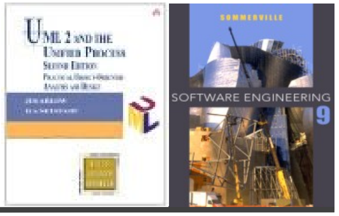
- Design classes
- Design relationships

Low-level Design

Lecture 8/Part 1

Low-level design

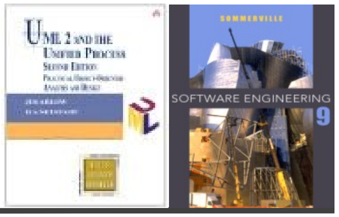


Purpose:

- Include all **code-level details** into the model
- Decide how exactly the system shall be **implemented**
- Typically an **implicit part of implementation**

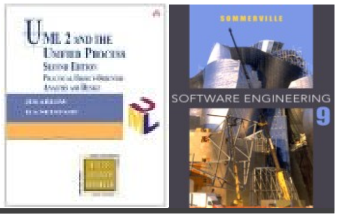
- **Techniques**
  - Design patterns
  - SOLID principles
  - Guidelines for dependable/testable/.. programming

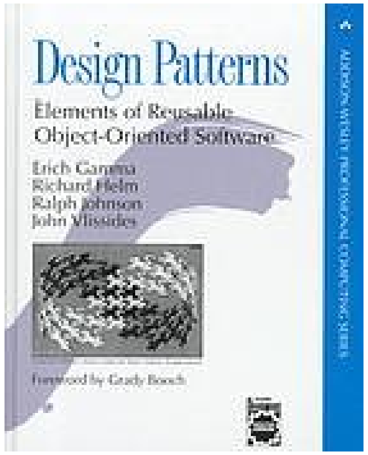
Design patterns



- ✧ A design pattern is a way of reusing abstract knowledge about a problem and its solution in object-oriented world.
  - Pattern descriptions make use of object-oriented characteristics such as inheritance, polymorphism and interface realization.
- ✧ A pattern is a description of the problem and the essence of its solution.
  - Not a concrete design but a **template** for a design solution that can be **instantiated in different ways**.
- ✧ It should be sufficiently abstract to be reusable in different settings.

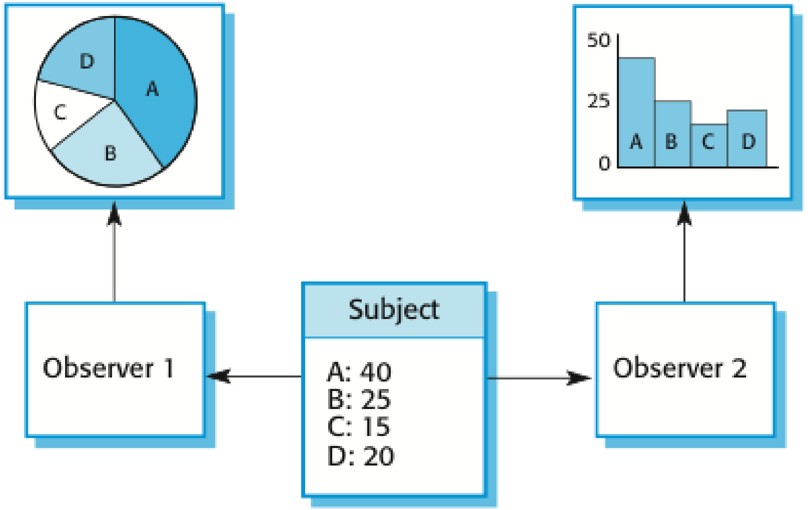
The “Gang of Four” design patterns



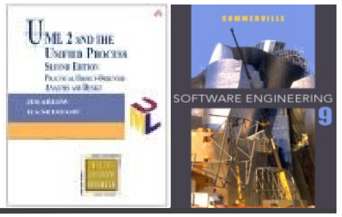


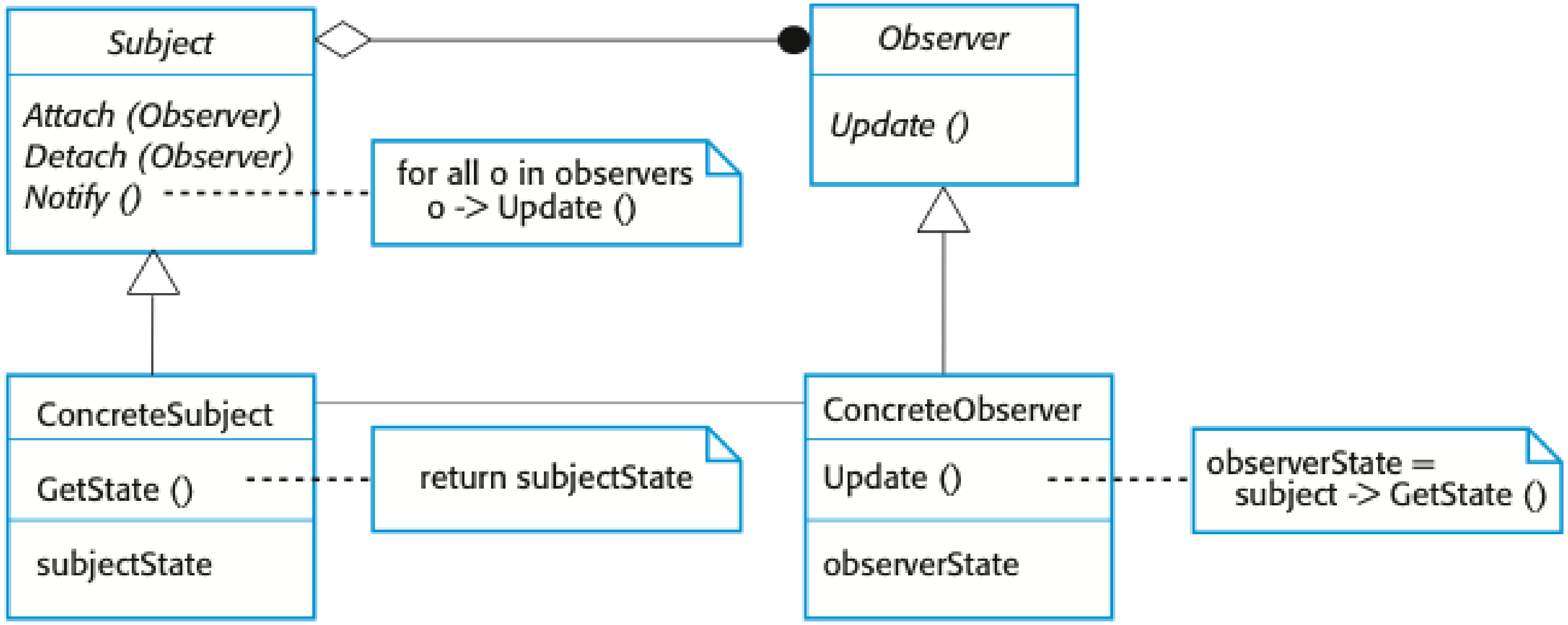
- ✧ Introduced in a book by GoF in 1995
- ✧ Collection of 23 classic software design patterns divided into three groups:
  - Creational
  - Structural
  - Behavioral

- ✧ **Observer pattern**
  - Behavioral pattern
  - Separates the display of object state from the object itself when multiple displays of state are needed.

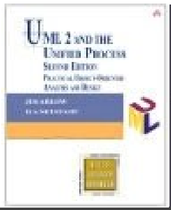


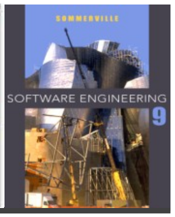
A UML model of the Observer pattern





Design problems

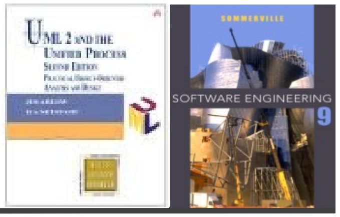




✧ Be aware that any design problem you are facing **may have an associated pattern** that can be applied.

- Tell several objects that the state of some other object has changed (Observer pattern).
- Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
- Allow classes with incompatible interfaces to work together by wrapping a new interface around that of an already existing class (Adapter pattern).
- Reduce the cost of creating and manipulating a large number of similar objects (Flyweight pattern).
- Restrict object creation for a class to only one instance (Singleton pattern).

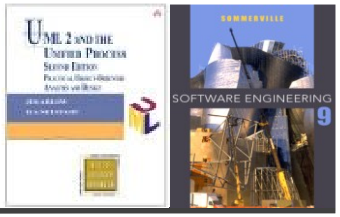
SOLID principles



- The “first five principles” identified by Robert C. Martin in the early 2000s that stands for five basic principles of object-oriented programming and design.

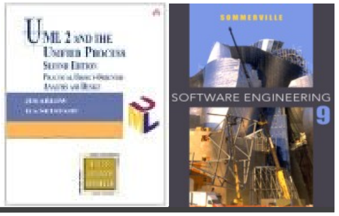
- **S**ingle responsibility
- **O**pen/closed
- **L**iskov substitution
- **I**nterface segregation
- **D**ependency inversion

Single responsibility principle

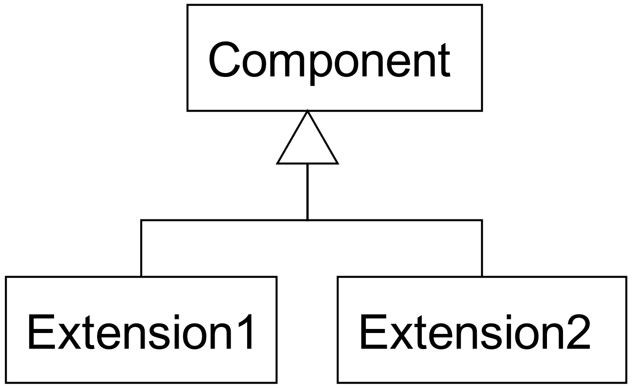


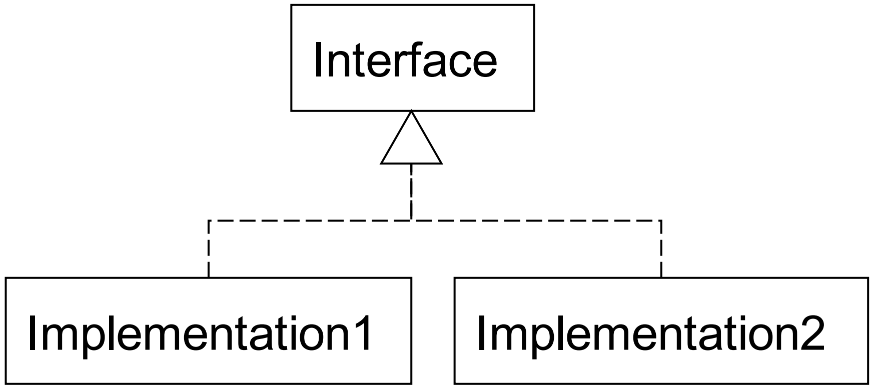
- The principle states that every class should have a **single responsibility**, and that responsibility should be **entirely encapsulated** by the class.
- A **responsibility** can be understood as a reason to change, so a class or module should have one, and only one, reason to change.
- As an example, consider a module that compiles and prints a report. Such a module can be changed for two reasons – because the **content** or the **format** changes.
  - If there is a change to the report compilation process, there is greater danger that the printing code breaks.

Open/closed principle

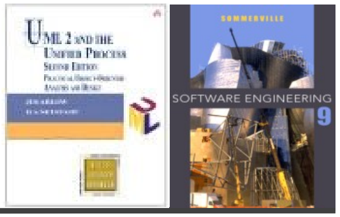


- The principle states that software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.
- Use **inheritance** and **interfaces** to avoid code changes when extending system functionality.

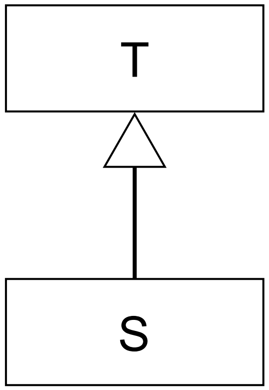


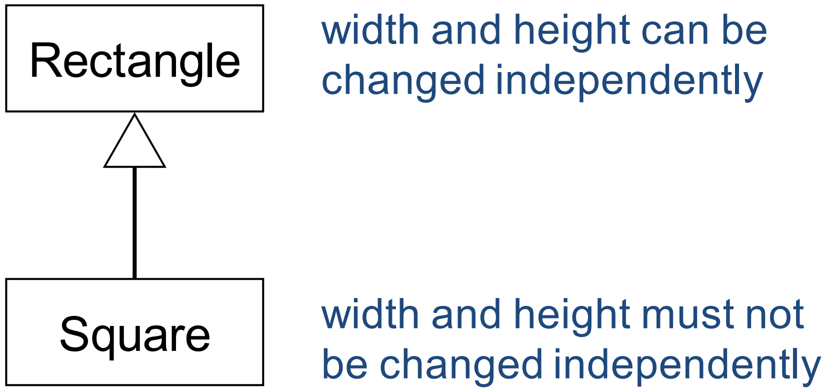


Liskov substitution principle

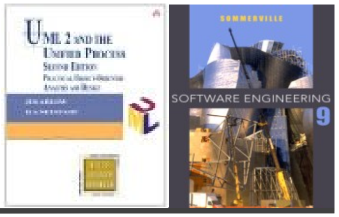


- The principle states that, in a computer program, if S is a subtype of T, then **objects of type T may be replaced with objects of type S** without altering any of the desirable properties of that program (correctness, task performed, etc.).

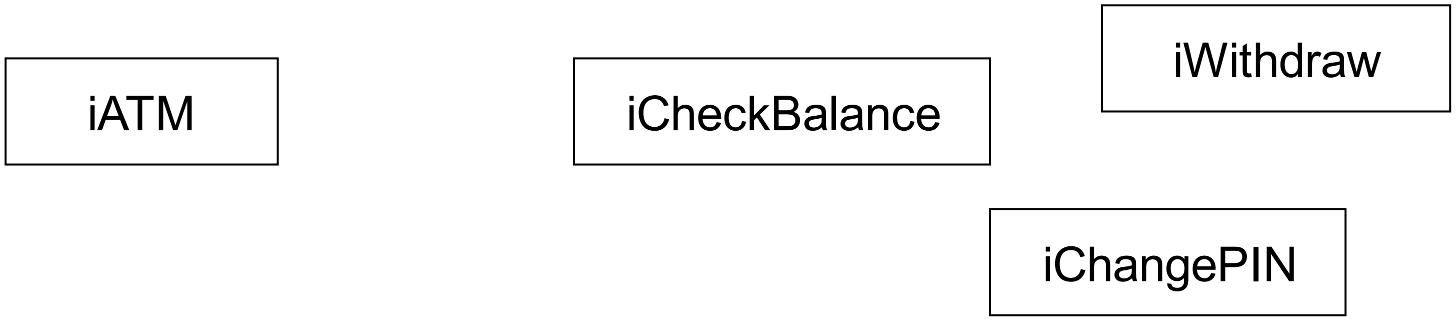



Rectangle
width and height can be changed independently
Square
width and height must not be changed independently

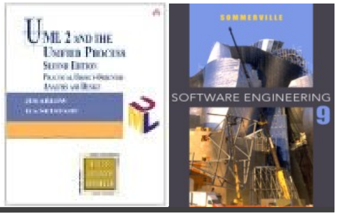
Interface segregation principle



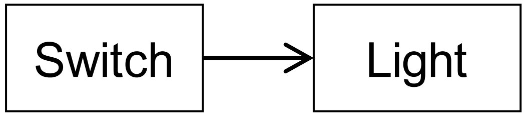
- ✧ The principle states that no client should be forced to **depend on methods it does not use**.
- ✧ ISP splits large interfaces into smaller and more specific “role” interfaces so that clients will only have to know about the methods that are of interest to them.
- ✧ ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

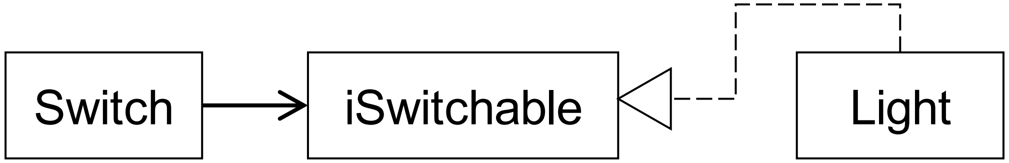


Dependency inversion principle

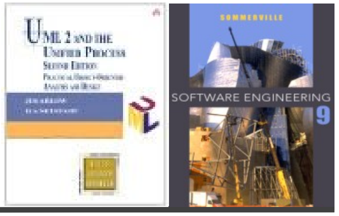


- The principle refers to a specific form of **decoupling** where conventional dependency relationships **established from high-level modules to low-level modules are inverted**. The principle states:
- **A.** High-level modules should not depend on low-level modules. Both should depend on abstractions.
- **B.** Abstractions should not depend upon details. Details should depend upon abstractions.



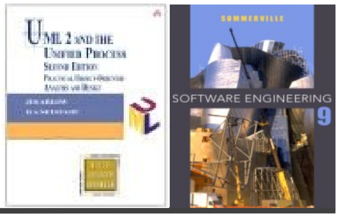


Clean code by Robert C. Martin



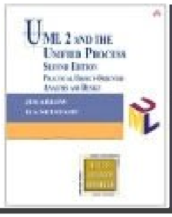
- ✧ A handbook of agile software craftsmanship
- ✧ Guidelines for:
  - Meaningful names
  - Functions
  - Comments
  - Formatting
  - Objects and data structures
  - Error handling
  - Concurrency
  - … and others
- ✧ Smells and heuristics

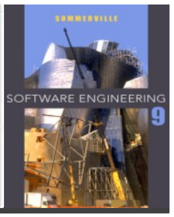
Design for non-functional qualities



- ✧ Design patterns help us to implement specific functionality while maintaining high code quality
  - Respect of design patterns improves system maintainability
- ✧ What if also other non-functional qualities are of high importance?
- ✧ Are there any “patterns” for dependability, performance, testability, etc.?

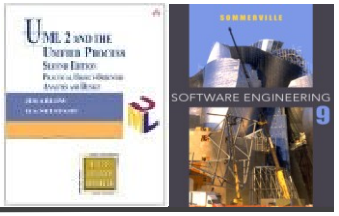
Dependable programming guidelines





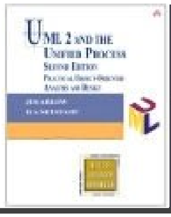
1. Limit the visibility of information in a program
2. Check all inputs for validity
3. Provide a handler for all exceptions
4. Minimize the use of error-prone constructs
5. Provide restart capabilities
6. Check array bounds
7. Include timeouts when calling external components
8. Name all constants that represent real-world values

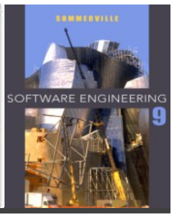
Limit the visibility of information in a program



- ✧ Program components should only be allowed access to data that they need for their implementation.
- ✧ This means that accidental corruption of parts of the program state by these components is impossible.
- ✧ You can control visibility by using abstract data types where the data representation is private and you only allow access to the data through predefined operations such as get () and put ().

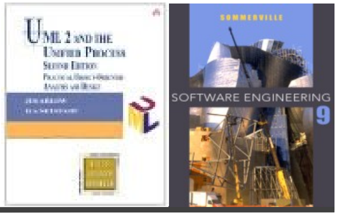
Check all inputs for validity





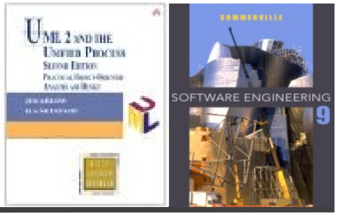
- ✧ All program take inputs from their environment and make assumptions about these inputs.
- ✧ However, program specifications rarely define what to do if an input is not consistent with these assumptions.
- ✧ Consequently, many programs behave unpredictably when presented with unusual inputs and, sometimes, these are threats to the security of the system.
- ✧ Consequently, you should always check inputs before processing against the assumptions made about these inputs.

Validity checks



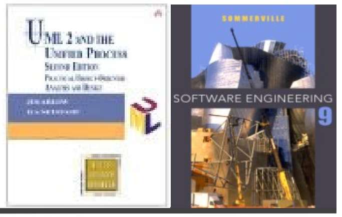
- ✧ Range checks
  - Check that the input falls within a known range.
- ✧ Size checks
  - Check that the input does not exceed some maximum size e.g. 40 characters for a name.
- ✧ Representation checks
  - Check that the input does not include characters that should not be part of its representation e.g. names do not include numerals.
- ✧ Reasonableness checks
  - Use information about the input to check if it is reasonable rather than an extreme value.

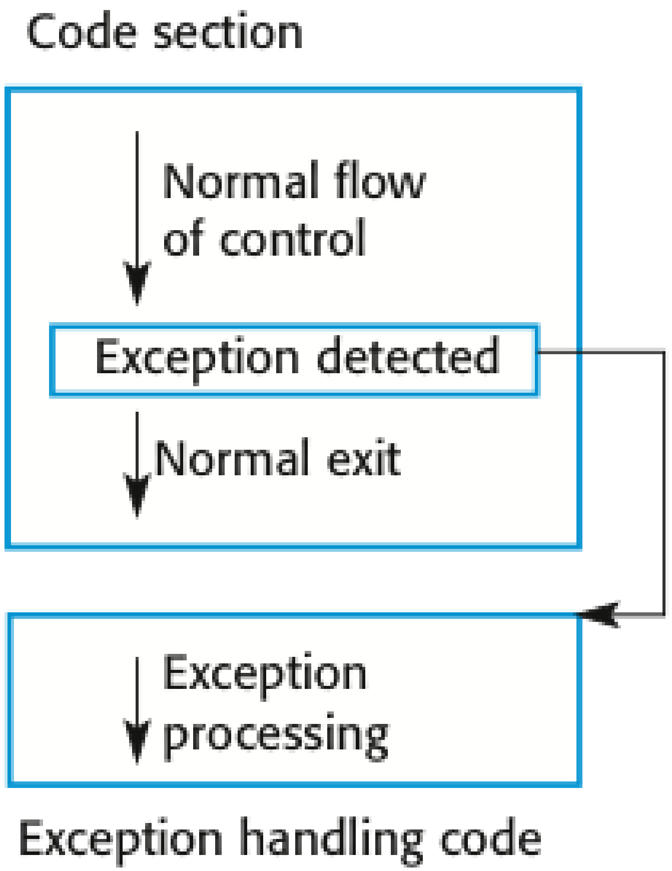
Provide a handler for all exceptions



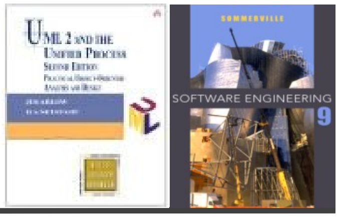
- A program exception is an error or some unexpected event such as a power failure.
- Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.
- Using normal control constructs to detect exceptions needs many additional statements to be added to the program. This adds a significant overhead and is potentially error-prone.

Exception handling



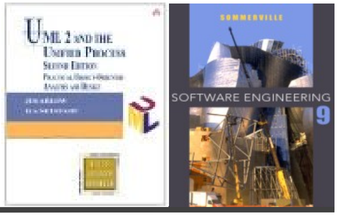


Exception handling



- ✧ Three possible exception handling strategies
  - Signal to a calling component that an exception has occurred and provide information about the type of exception.
  - Carry out some alternative processing to the processing where the exception occurred. This is only possible where the exception handler has enough information to recover from the problem that has arisen.
  - Pass control to a run-time support system to handle the exception.
- ✧ Exception handling is a mechanism to provide some fault tolerance

Minimize the use of error-prone constructs

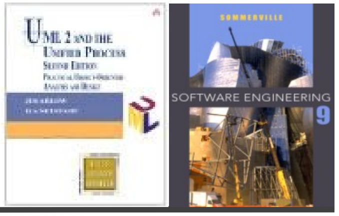


✧ Program faults are usually a consequence of human error because programmers lose track of the relationships between the different parts of the system

✧ This is exacerbated by error-prone constructs in programming languages that are inherently complex or that don’t check for mistakes when they could do so.

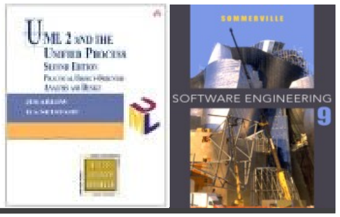
✧ Therefore, when programming, you should try to avoid or at least minimize the use of these error-prone constructs.

Error-prone constructs



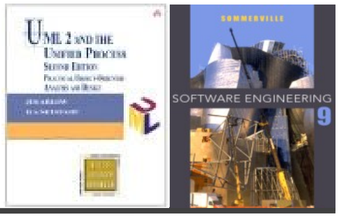
- ✧ Unconditional branch (goto) statements
- ✧ Floating-point numbers
  - Inherently imprecise. The imprecision may lead to invalid comparisons.
- ✧ Pointers
  - Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change.
- ✧ Dynamic memory allocation
  - Run-time allocation can cause memory overflow.

Error-prone constructs



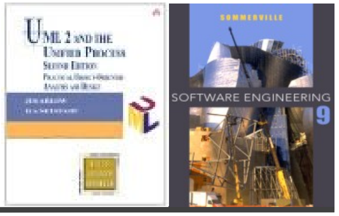
- Parallelism
  - Can result in subtle timing errors because of unforeseen interaction between parallel processes.
- Recursion
  - Errors in recursion can cause memory overflow as the program stack fills up.
- Interrupts
  - Interrupts can cause a critical operation to be terminated and make a program difficult to understand.
- Inheritance
  - Code is not localised. This can result in unexpected behaviour when changes are made and problems of understanding the code.

Error-prone constructs



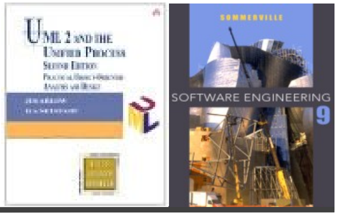
- ✧ Aliasing
  - Using more than 1 name to refer to the same state variable.
- ✧ Unbounded arrays
  - Buffer overflow failures can occur if no bound checking on arrays.
- ✧ Default input processing
  - An input action that occurs irrespective of the input.
  - This can cause problems if the default action is to transfer control elsewhere in the program. In incorrect or deliberately malicious input can then trigger a program failure.

Provide restart capabilities



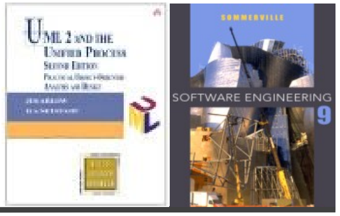
- ✧ For systems that involve long transactions or user interactions, you should always provide a restart capability that allows the system to restart after failure without users having to redo everything that they have done.
- ✧ Restart depends on the type of system
  - Keep copies of forms so that users don't have to fill them in again if there is a problem
  - Save state periodically and restart from the saved state

Check array bounds



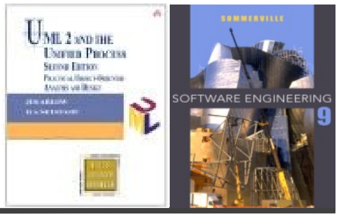
- ✧ In some programming languages, such as C, it is possible to address a memory location outside of the range allowed for in an array declaration.
- ✧ This leads to the well-known 'bounded buffer' vulnerability where attackers write executable code into memory by deliberately writing beyond the top element in an array.
- ✧ If your language does not include bound checking, you should therefore always check that an array access is within the bounds of the array.

Include timeouts when calling external components



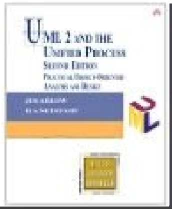
- ✧ In a distributed system, failure of a remote computer can be ‘silent’ so that programs expecting a service from that computer may never receive that service or any indication that there has been a failure.
- ✧ To avoid this, you should always include timeouts on all calls to external components.
- ✧ After a defined time period has elapsed without a response, your system should then assume failure and take whatever actions are required to recover from this.

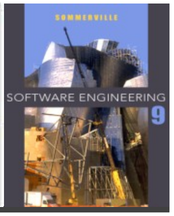
Name all constants that represent real-world values



- ✧ Always give constants that reflect real-world values (such as tax rates) names rather than using their numeric values and always refer to them by name
- ✧ You are less likely to make mistakes and type the wrong value when you are using a name rather than a value.
- ✧ This means that when these ‘constants’ change (for sure, they are not really constant), then you only have to make the change in one place in your program.

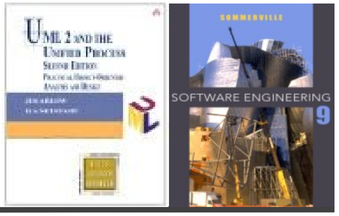
Low-level performance tactics



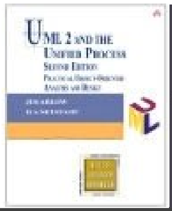
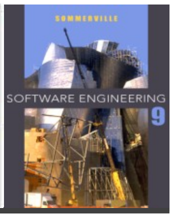


- ✧ Reduce the resources required for processing an event stream.
  - Increase computational efficiency.
  - Reduce computational overhead.
- ✧ Reduce the number of events processed.
  - Manage event rate.
  - Control frequency of sampling.
- ✧ Control the use of resources.
  - Bound execution times.
  - Bound queue sizes.

Testability tactics



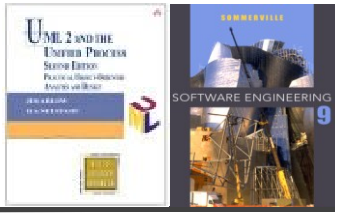
- **Record/playback.** The information crossing an interface during normal operation can be saved and accessed.
- **Separate interface from implementation.** This allows substitution of implementations for testing purposes.
- **Specialize access routes/interfaces.** Having specialized testing interfaces allows the capturing or specification of variable values for a component through a test harness, independently from its normal execution.
- **Built-in monitors.** The component can maintain state, performance load, capacity, security, or other information accessible through an interface.

Implementation Issues

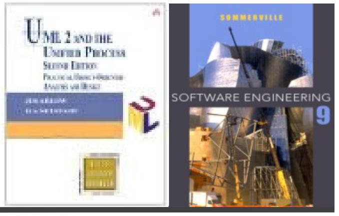
Lecture 8/Part 2

Implementation issues



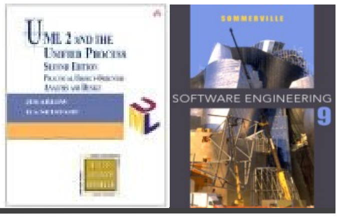
- ✧ Some implementation issues that are often not covered in programming texts:
  - **Reuse** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - **Configuration management** During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - **Host-target development** Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

Reuse



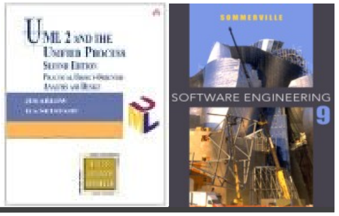
- ✧ From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- ✧ Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- ✧ An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

Reuse levels



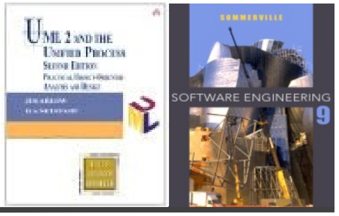
- The abstraction level
  - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Reuse costs



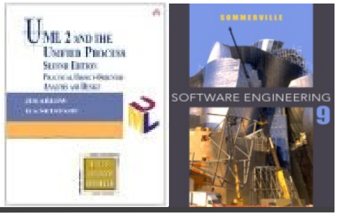
- ✧ The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- ✧ Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- ✧ The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- ✧ The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Configuration management



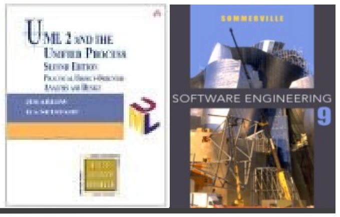
- ✧ Configuration management is the name given to the general process of managing a changing software system.
- ✧ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

Configuration management activities



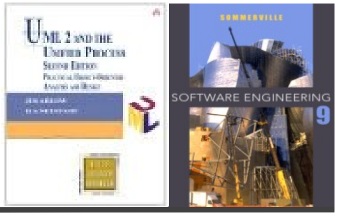
- Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Host-target development

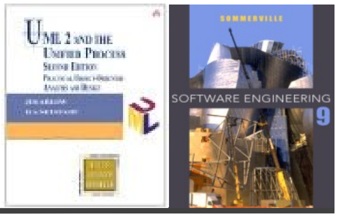


- ✧ Most software is developed on one computer (the host), but runs on a separate machine (the target).
- ✧ More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- ✧ Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Key points



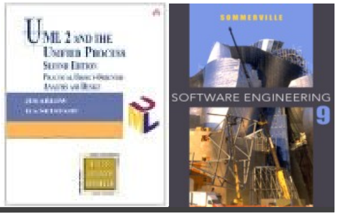
- ✧ When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- ✧ Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- ✧ Most software development is host-target development. You use a development environment on a host machine to develop the software, which is transferred to a target machine for execution.



UML Class Diagram in Design

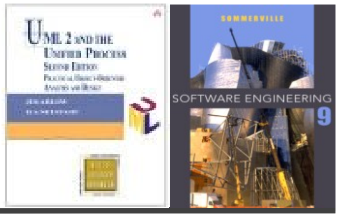
Lecture 8/Part 3

Design model



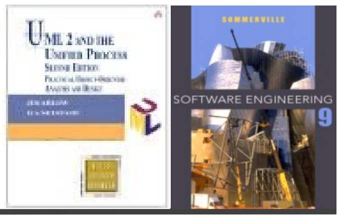
- ✧ Design model is a refinement of an analysis model to such a degree that it can be implemented
  - In MDD design models include all implementation details and can be automatically translated into code
- ✧ In OO design models:
  - All attributes are completely specified
  - Analysis operations become fully specified methods
  - Many new classes are added to include implementation details, such as utility classes (String, Date, Time, etc.), middleware classes (DB access, communication, etc.) or GUI classes (Applet, Button, etc.)
- ✧ Design models are programming-language specific
  - Multiple inheritance, templates, nested classes, collections

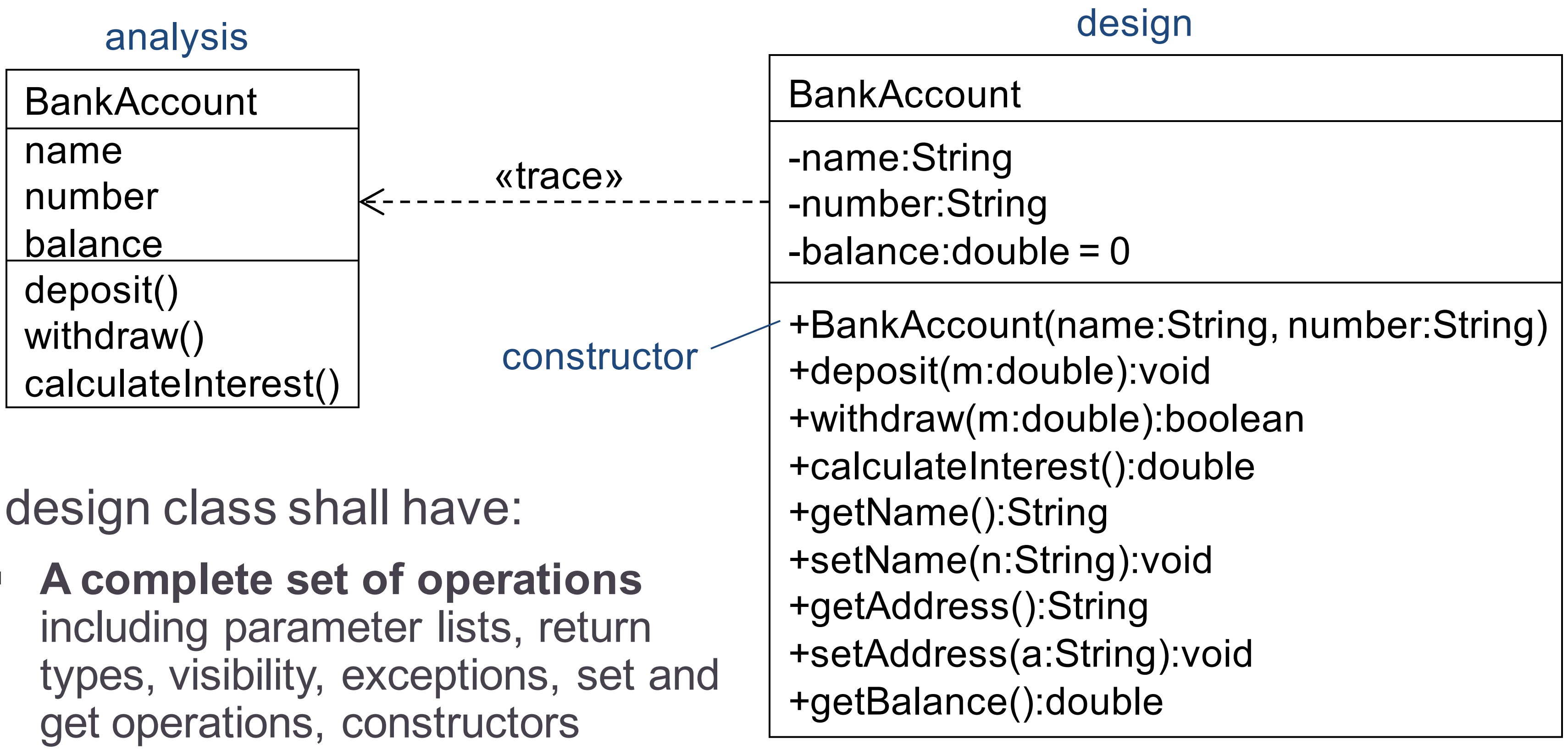
Analysis vs. design model



- A design model may contain 10 to 100 times as many classes as the analysis model
  - The analysis model helps us to see the big picture without getting lost in implementation details
- We need to maintain both models if:
  - It is a big system (>200 design classes)
  - It has a long expected lifespan
  - It is a strategic system
  - We are outsourcing construction of the system
- We can make do with only a design model if:
  - It is a small system
  - It has a short lifespan
  - It is not a strategic system

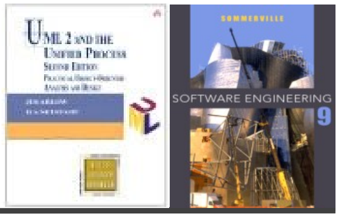
Anatomy of a design class





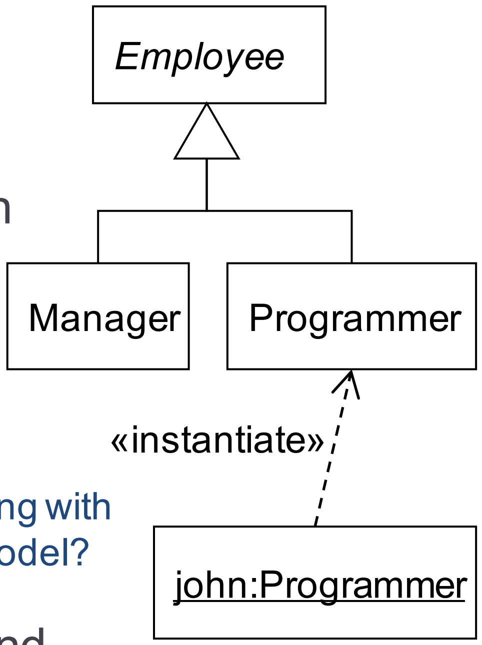
✧ A design class shall have:

- **A complete set of operations** including parameter lists, return types, visibility, exceptions, set and get operations, constructors
- **A complete set of attributes** including types and default values



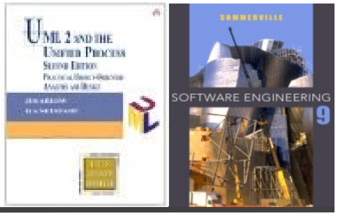
High cohesion, low coupling

- High cohesion
  - Each class should have a set of operations that support the intent of the class, no more and no less
  - Each class should model a single abstract concept
- Low coupling
  - A particular class should be associated with just enough other classes to allow it to realise its responsibilities
  - Only associate classes if there is a true semantic link between them – never to only reuse code!
  - Use aggregation rather than inheritance
- Primitive operations
  - Each operation shall implement a single functionality, and each functionality shall be implemented by single operation

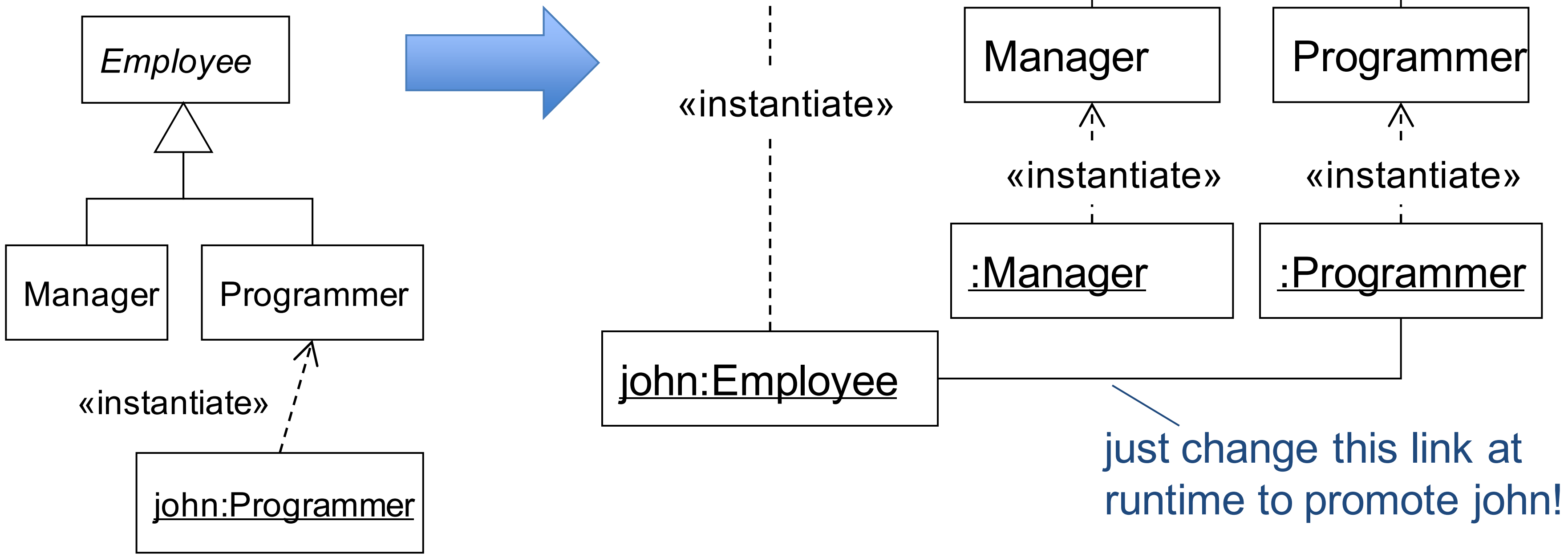


What is wrong with this model?

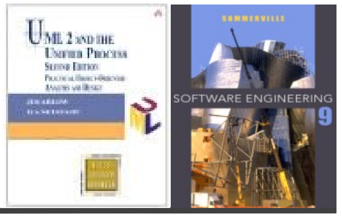
Aggregation vs. inheritance



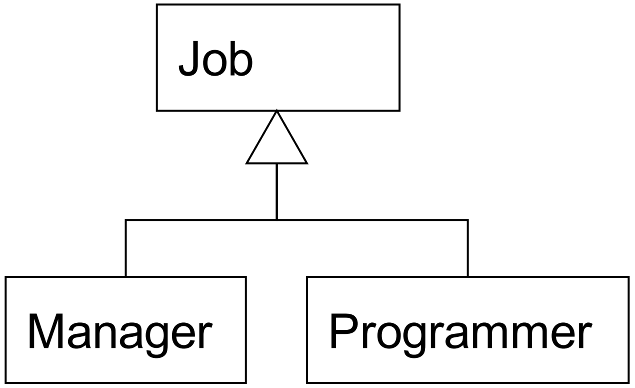
- An employee **has** a job, not **is** a job.
- An employee can have more jobs.



Inheritance vs. interface realization

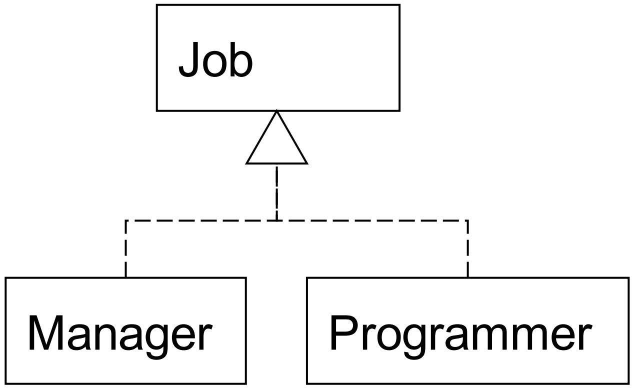


- ✧ With **inheritance** we get two things:
  - Interface – the public operations of the base classes
  - Implementation – the attributes, relationships, operations of the class



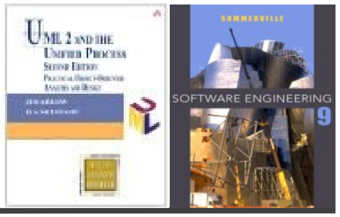
Use inheritance when we want to *inherit implementation*.

- ✧ With **interface** we get one thing:
  - Interface – a set of public operations, attributes and relationships that have no implementation



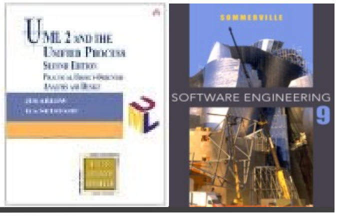
Use interface realization when we want to *define a contract*.

Key points (design classes)



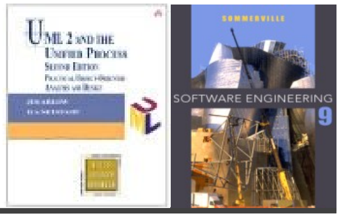
- ✧ Design classes come from:
  - A refinement of analysis classes (i.e. the business domain)
  - From the solution domain
- ✧ Design classes must be well-formed:
  - High cohesion
  - Low coupling
  - Primitive operations
- ✧ Don’t overuse inheritance
  - Use inheritance for "is kind of"
  - Use aggregation for "is role played by"
  - Use interfaces rather than inheritance to define contracts

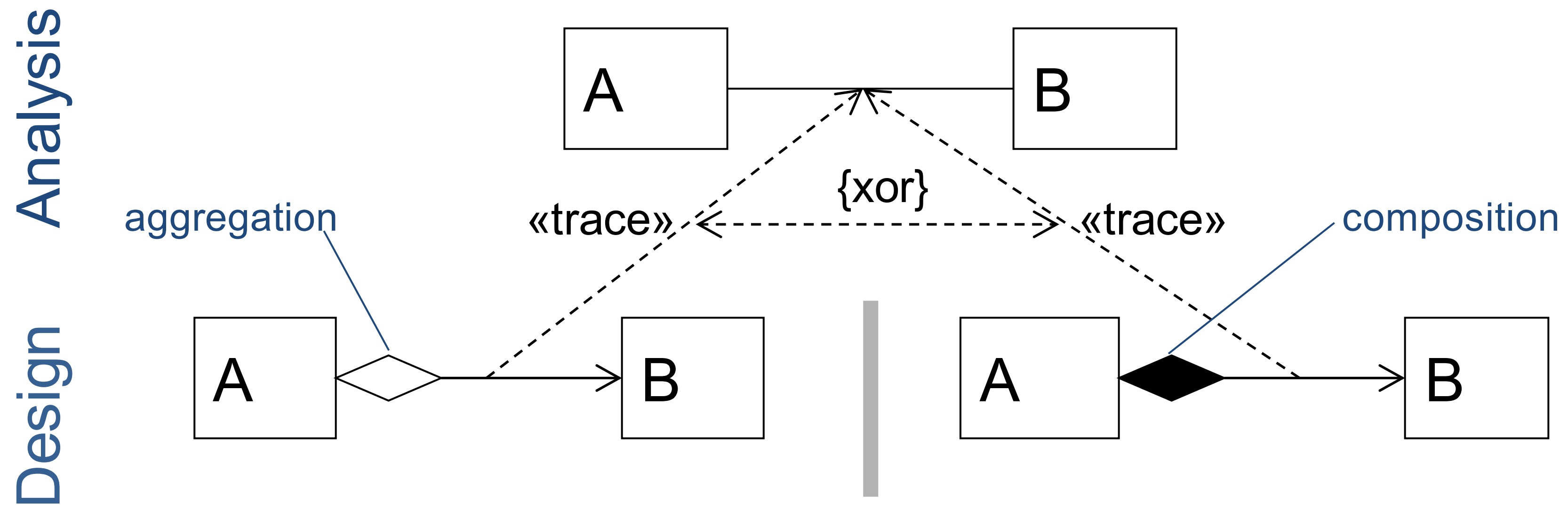
Design relationships

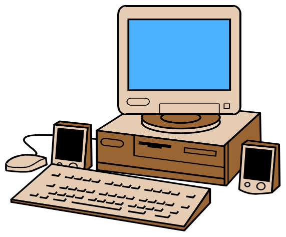


- Refining analysis associations to design associations involves several procedures:
  - refining associations to aggregation or composition
  - implementing one-to-many associations
  - implementing many-to-one associations
  - implementing many-to-many associations
  - implementing bidirectional associations
  - implementing association classes
- All design associations must have:
  - navigability
  - multiplicity on both ends

Aggregation and composition



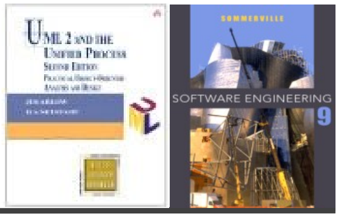




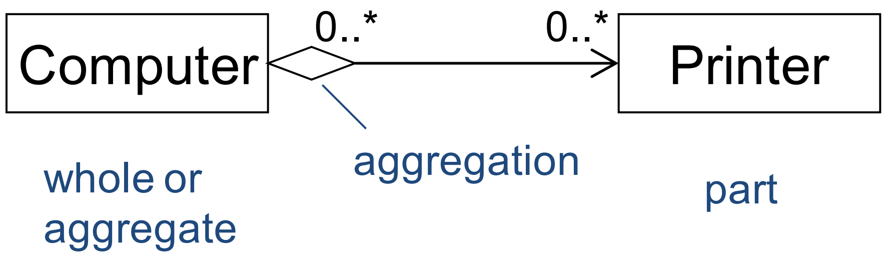
Some objects are weakly related like a computer and its peripherals

Some objects are strongly related like a tree and its leaves

Aggregation semantics



aggregation is a *whole–part* relationship



A Computer may be attached to 0 or more Printers

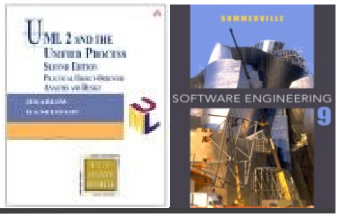
At any one point in time a Printer is connected to 0 or more Computers

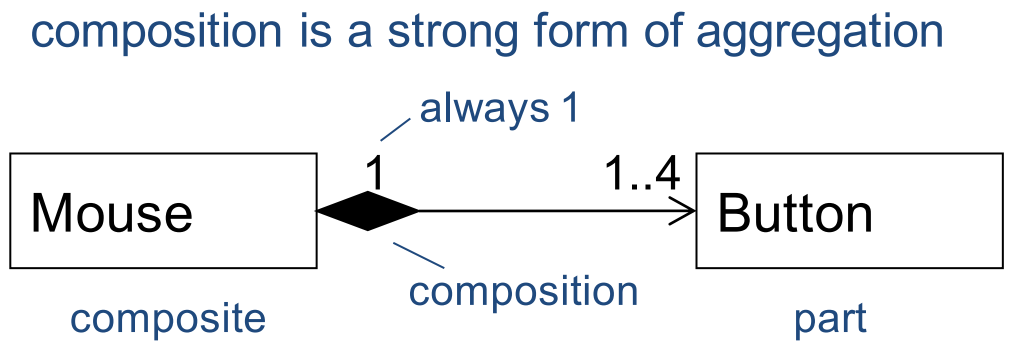
The Printer exists even if there are no Computers

The Printer is independent of the Computer

- The aggregate can (sometimes) exist independently of the parts
- The parts can (sometimes) exist independently of the aggregate
- It is possible to have shared ownership of the parts by several aggregates

Composition semantics



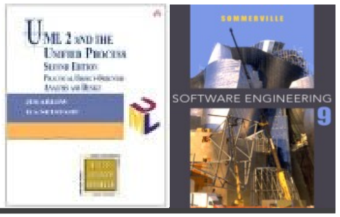


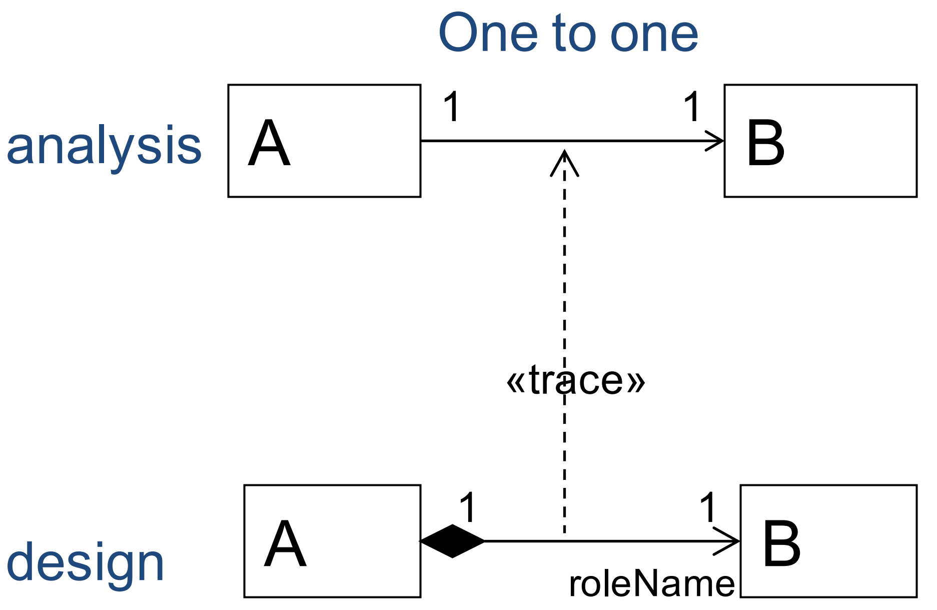
The buttons have no independent existence. If we destroy the mouse, we destroy the buttons. They are an integral part of the mouse

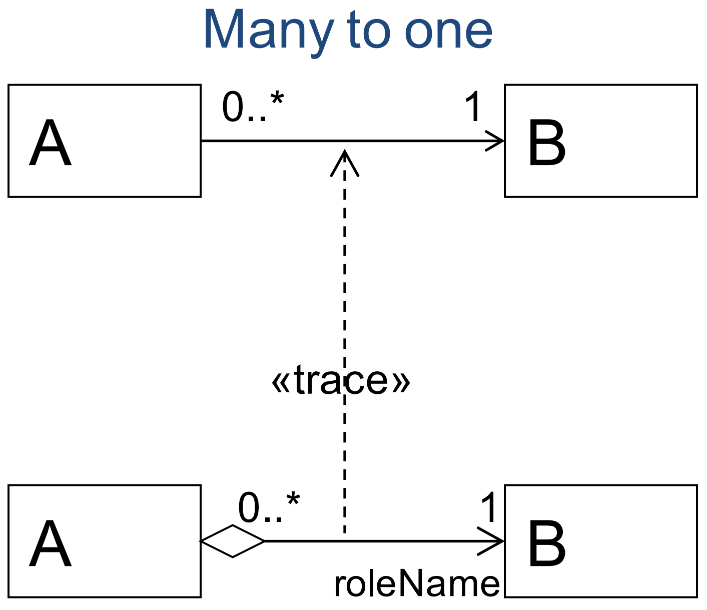
Each button can belong to exactly 1 mouse

- ✧ The parts belong to exactly 1 whole at a time
- ✧ The composite has sole responsibility for the disposition of all its parts. This means responsibility for their creation and destruction
- ✧ If the composite is destroyed, it must either destroy all its parts, OR give responsibility for them over to some other object

One-to-one and many-to-one associations



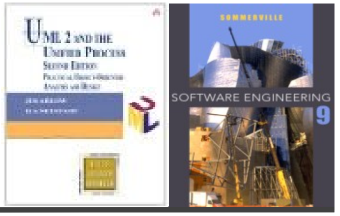




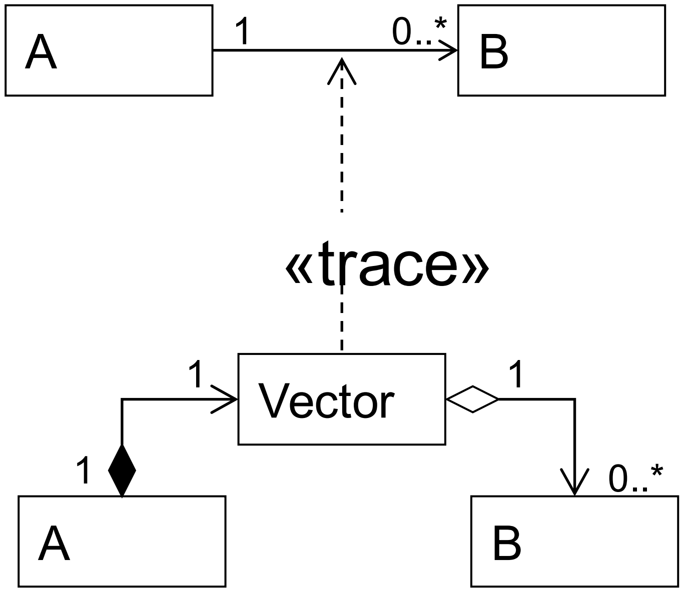
- One-to-one associations in analysis usually imply single ownership and usually refine to compositions

- Many-to-one relationships in analysis imply shared ownership and are refined to aggregations

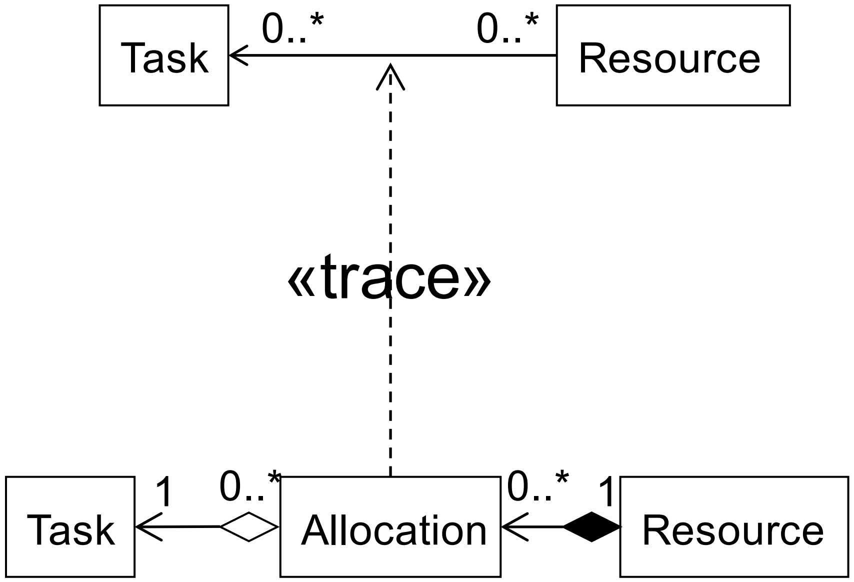
One-to-many and many-to-many associations



One to many



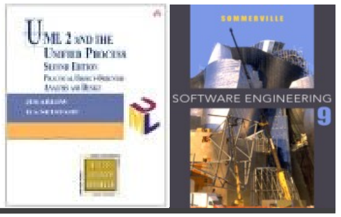
Many to many



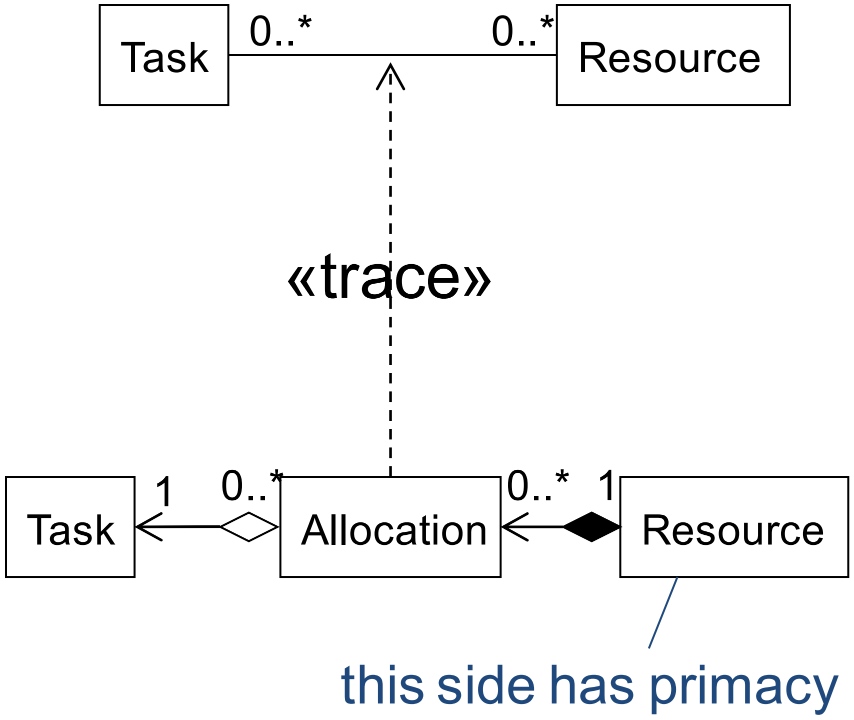
- Collection classes instances store a collection of object references to objects of the target and provide methods for operating the collection
- In Java in the java.util library

- Many-to-many associations shall be reified into intermediate design classes

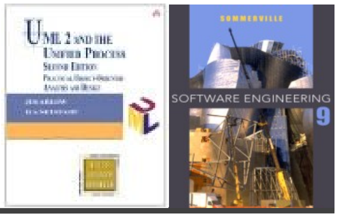
Many to many associations



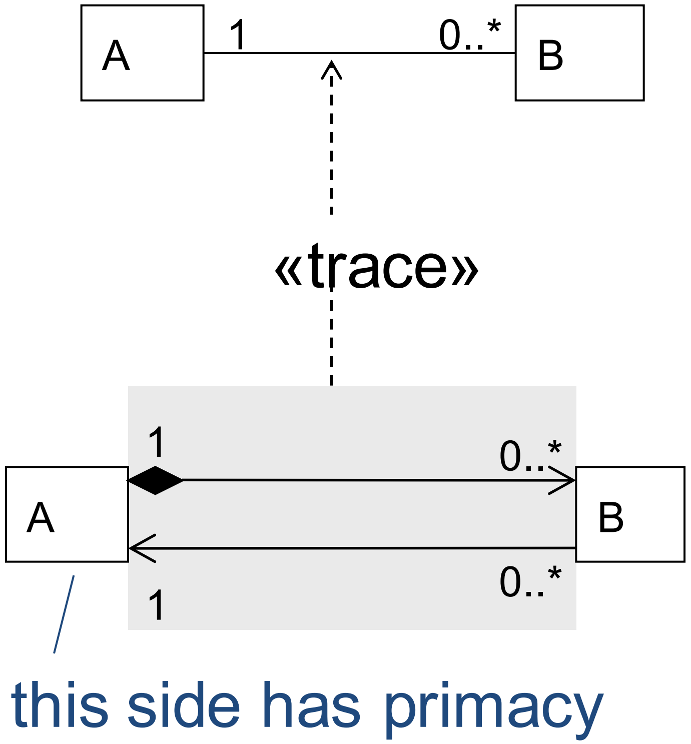
- There is no commonly used OO language that directly supports many-to-many associations
- We must reify such associations into design classes
- Again, we must decide which side of the association should have primacy and use composition, aggregation and navigability accordingly



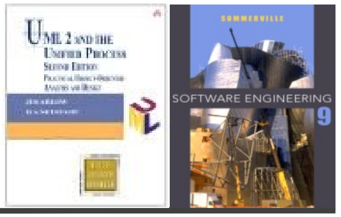
Bi-directional associations



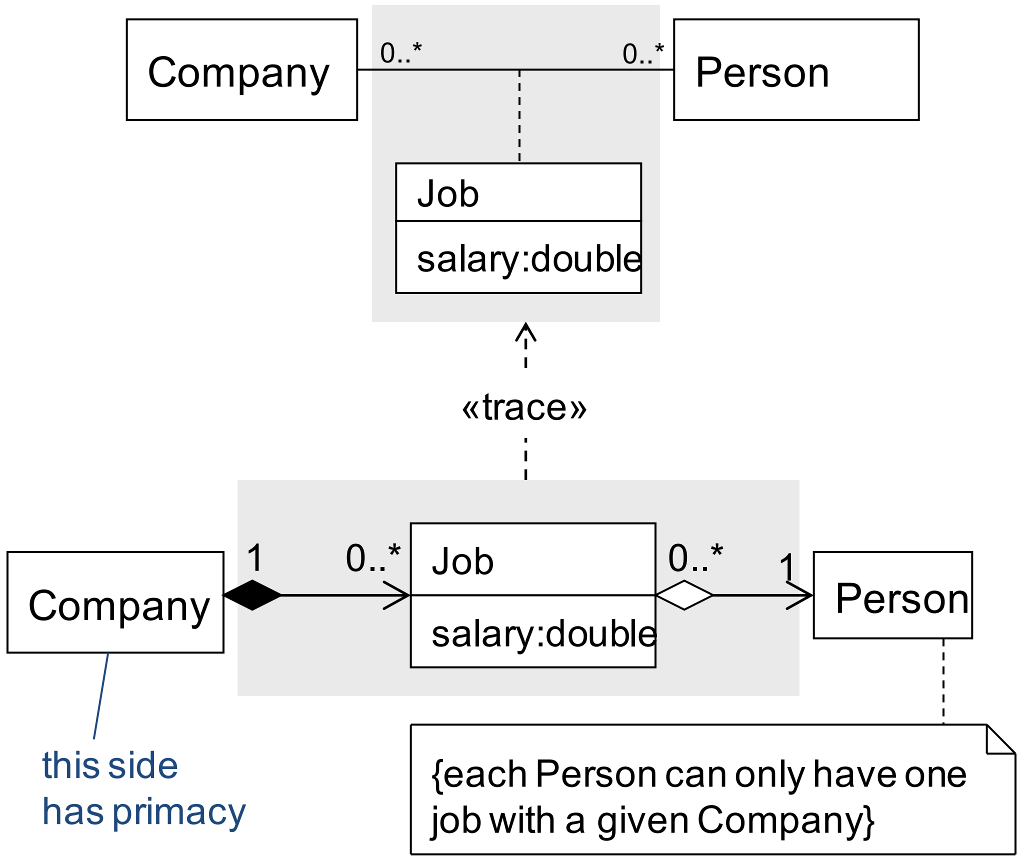
- ✧ There is no commonly used OO language that directly supports bi-directional associations
- ✧ We must resolve each bi-directional associations into two unidirectional associations
- ✧ Again, we must decide which side of the association should have primacy and use composition, aggregation and navigability accordingly



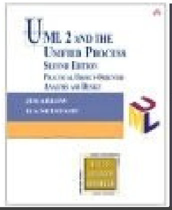
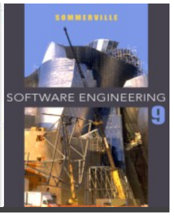
Association classes



- There is no commonly used OO language that directly supports association classes
- Refine all association classes into a design class
- Decide which side of the association has primacy and use composition, aggregation and navigability accordingly



Key points (design relationships)

- ✧ In this section we have seen how we take the incompletely specified associations in an analysis model and refine them to:
  - Aggregation
    - Whole-part relationship
    - Parts are independent of the whole
    - Parts may be shared between wholes
  - Composition
    - A strong form of aggregation
    - Parts are entirely dependent on the whole
    - Parts may not be shared
- ✧ One-to-many, many-to-many, bi-directional associations and association classes are refined in design