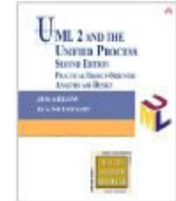


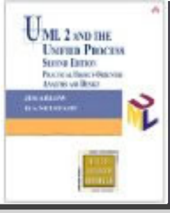
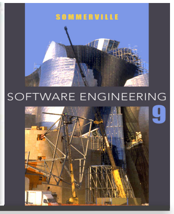
Lecture 6

HIGH-LEVEL DESIGN

PB007 Software Engineering I
Faculty of Informatics, Masaryk University

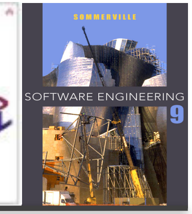
Fall 2017

Purpose of system design

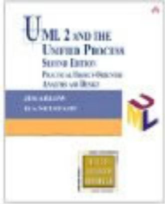
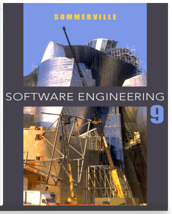
- Refine how the system's functions are to be implemented and how non-functional requirements are to be ensured
- Decide on strategic design issues such as concurrency, redundancy, persistence, distribution etc. to end with a design satisfying both functional and non-functional requirements
- Create policies to deal with tactical design issues

Design best practices



- A system design consists of a **collection of decisions** that help to control different attributes of software quality.
  - The design aims to ensure achievement of system functionality, but whenever there are different ways to achieve the functionality, the impact of each design decision on software quality becomes the issue.
- Quality-driven design decisions are often known as **tactics**, which isolate and describe design best practices with respect to a specific quality attribute.
  - **Design patterns** are a specific and very popular tactic used during low-level design.

Outline

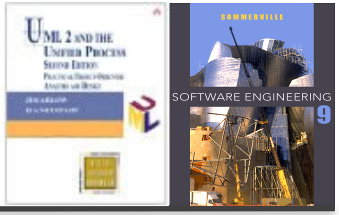
✧ Design for reliability and availability

✧ Design for security

✧ Design for performance, modifiability and usability

✧ UML Class Diagram in Design

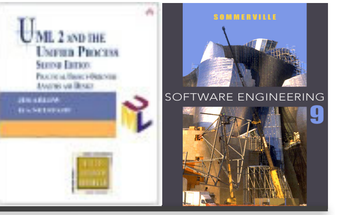
- Design classes
- Design relationships



Design for Reliability and Availability

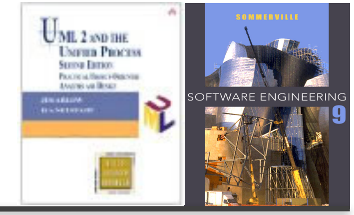
Lecture 6/Part 1

Software reliability and availability



- In general, software customers expect all software to be dependable. However, for **non-critical applications**, they may be willing to accept some system failures.
- Some applications (critical systems) have **very high dependability requirements** and special software engineering techniques may be used to achieve this.
  - Medical systems
  - Telecommunications and power systems
  - Aerospace systems

Dependability achievement



✧ Fault avoidance

- **The development process** is organised so that faults in the system are detected and repaired before delivery to the customer.
- **Verification and validation** techniques are used to discover and remove faults in a system before it is deployed.

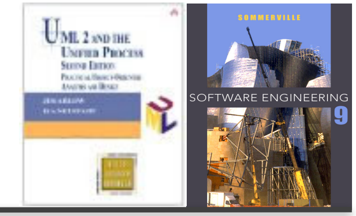
✧ Fault detection

- **Run-time techniques** to detect faults and failures, such as acceptance tests, ping/echo, heartbeat.

✧ Fault tolerance

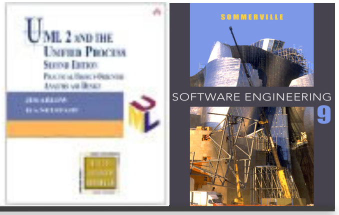
- The system is designed so that faults in the delivered software do not result in system failure.

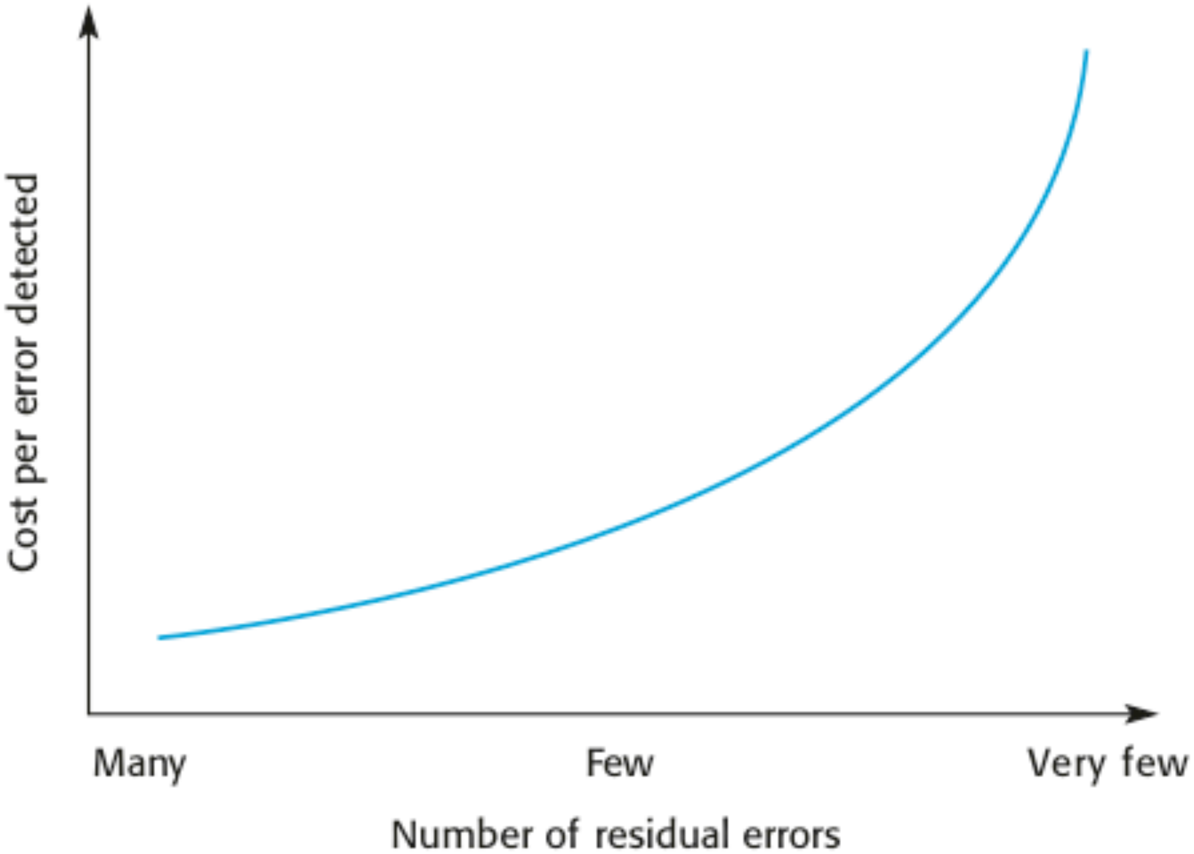
Dependable processes for fault avoidance

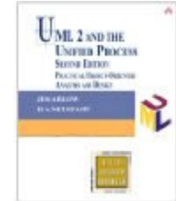


- To ensure a minimal number of software faults, it is important to have a well-defined, **repeatable software process**.
- The process should not depend entirely on individual skills; rather can be enacted by different people.
- **QA engineers** use information about the process to check if good software engineering practice has been used.
- It is clear that the process activities should include significant effort devoted to **verification and validation**.

Fault discovery and its costs



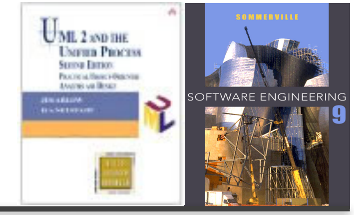




Run-time fault detection tactics

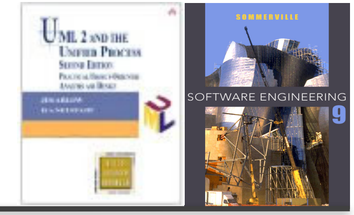
- **Monitoring and event processing.** Collection, logging and processing of events that may signal problems.
- **Acceptance tests.** Acceptance checking for individual methods and code fragments, raising signals for fault handling (possibly with an exception).
- **Ping/echo.** One component issues a ping and expects to receive back an echo, within a predefined time.
- **Heartbeat (dead man timer).** In this case one component emits a heartbeat message periodically and another component listens for it. If the heartbeat fails, the fault correction component is notified.

Fault tolerance



- In critical situations, software systems must be fault tolerant.
  - Fault tolerance is required where there are high availability requirements or where system failure costs are very high.
- **Fault tolerance** means that the system can continue in operation in spite of software failure.
  - Even if the system has been proved to conform to its specification, it must also be fault tolerant as there may be specification errors or the validation may be incorrect.
- **Dependable systems architectures** are used in situations where fault tolerance is essential.
  - These architectures are generally all based on **redundancy and diversity**.

Diversity and redundancy



✧ **Redundancy**

- Keep more than 1 version of a critical component available so that if one fails then a backup is available.
- E.g. switch to backup servers automatically if failure occurs.

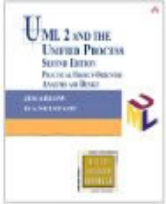
✧ **Diversity**

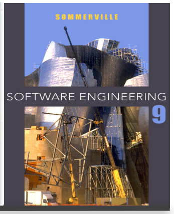
- Provide the same functionality in different ways so that they will not fail in the same way.
- E.g. different servers may be implemented using different operating systems (e.g. Windows and Linux).

✧ However, adding diversity and redundancy adds complexity and this can increase the chances of error.

- Some engineers advocate simplicity and extensive V & V is a more effective route to software dependability.

Fault tolerance and recovery tactics (1)





✧ Exception handling

- Detection, signaling and propagation of the information about system faults, including the handling on an appropriate place.
- Decide well on the responsibilities in the system.

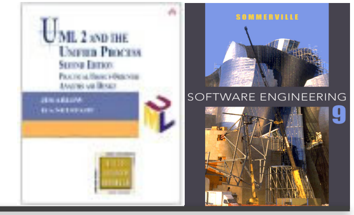
✧ Checkpoint/rollback

- Recording of a consistent state (either periodically or in response to specific events), to which the system can be restored.

✧ Recovery capabilities

- The system should be able to perform a clean-up after a major failure, so that no disturbances remain in the system.

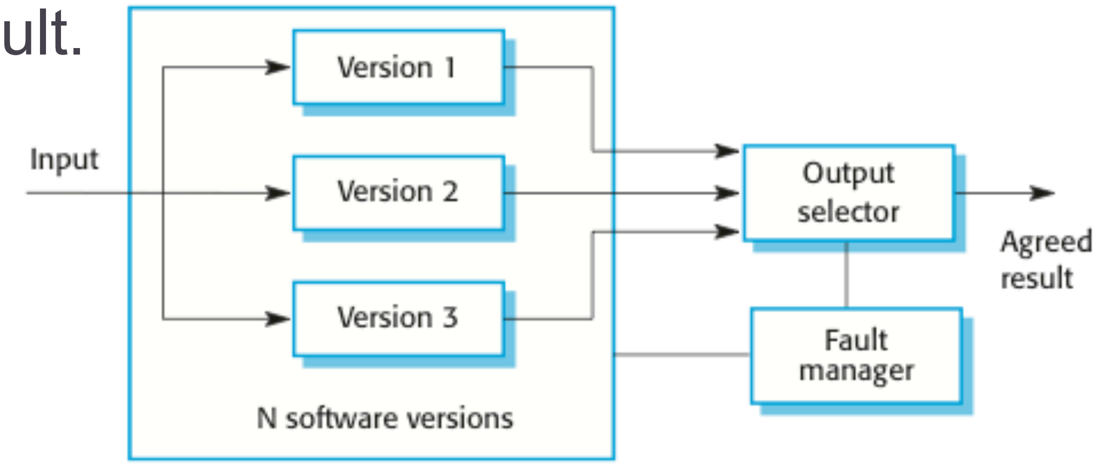
Fault tolerance and recovery tactics (2)



- **Active redundancy** (hot restart)
  - All redundant components respond to events in parallel. Consequently, they are all in the same state. The response from only one component is used (usually the first to respond).
- **Passive redundancy** (warm restart/dual redundancy)
  - One component (the primary) responds to events and informs the other components (the standbys) of state updates they must make.
- **Voting**
  - Processes running on redundant processors each take equivalent input and compute a simple output value that is sent to a voter to choose non-deviant result.

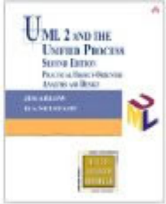
N-version programming pattern

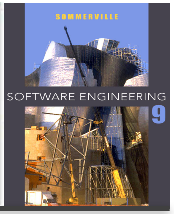
- Multiple versions of a software system carry out computations at the same time.
  - The versions should be designed and implemented by different teams, to avoid repeating the same mistake.
- The results are compared using a voting system and the majority result is taken
  to be the correct result.



Which of the tactics are involved here?

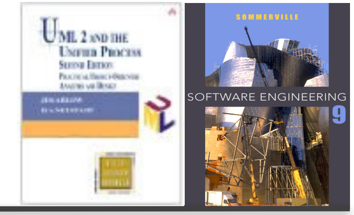
Protection systems



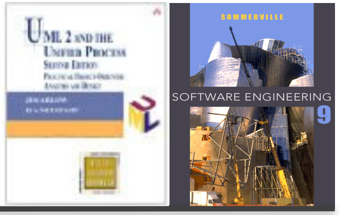


- A specialized **monitoring and control system** that is associated with another system, which can take **emergency action** if a failure occurs.
  - System to stop a train if it passes a red light
  - System to shut down a reactor if temperature is too high
- Protection systems are **redundant** because they include monitoring and control capabilities that replicate those in the controlled software.
- Protection systems should be **diverse** and use different technology from the controlled software.

Key points



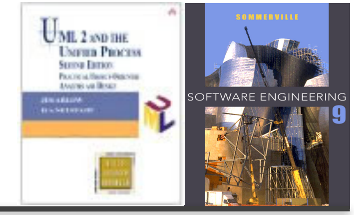
- ✧ Reliability and availability achievement
  - Fault avoidance, fault detection and fault tolerance
- ✧ Fault avoidance
  - Repeatable software development process
  - Testing
- ✧ Fault detection
  - Monitoring, acceptance tests, ping/echo, heartbeat
- ✧ Fault tolerance
  - Redundancy and diversity
  - N-version programming



Design for Security

Lecture 6/Part 2

Design for security



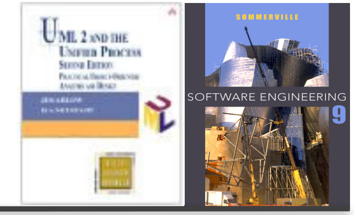
✧ Two fundamental issues have to be considered when designing an architecture for security.

- **Protection**
  - How should the system be organised so that critical assets can be protected against external attack?
- **Distribution**
  - How should system assets be distributed so that the effects of a successful attack are minimized?

✧ These are potentially conflicting

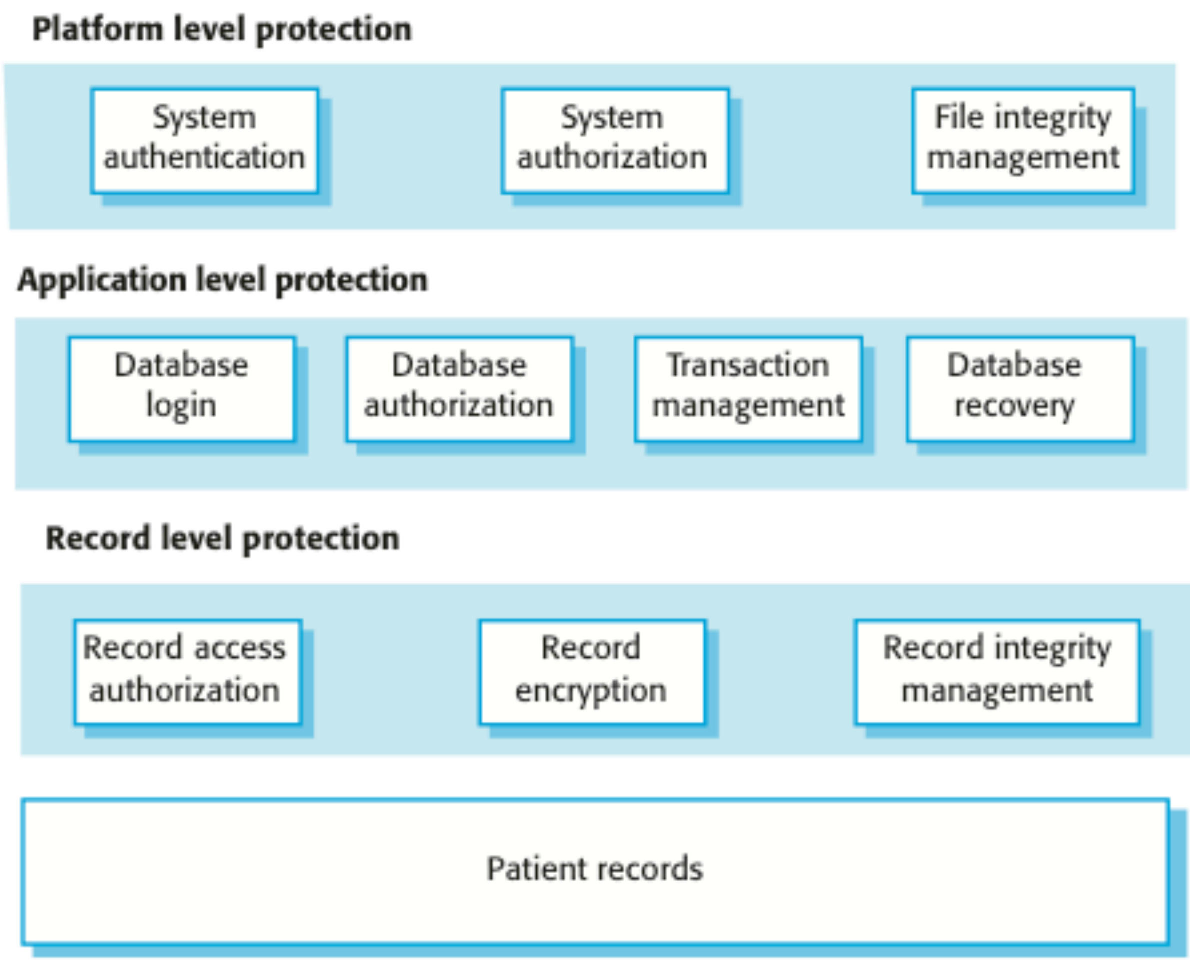
- If assets are distributed, then they are more expensive to protect. If assets are protected, then usability and performance requirements may be compromised.

Protection

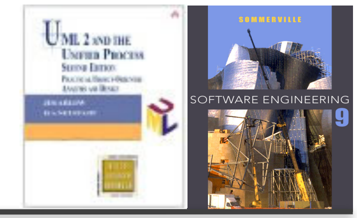


- ✧ **Platform-level protection**
  - Top-level controls on the platform on which a system runs.
- ✧ **Application-level protection**
  - Specific protection mechanisms built into the application itself e.g. additional password protection.
- ✧ **Record-level protection**
  - Protection that is invoked when access to specific information is requested
- ✧ These lead to a layered protection architecture

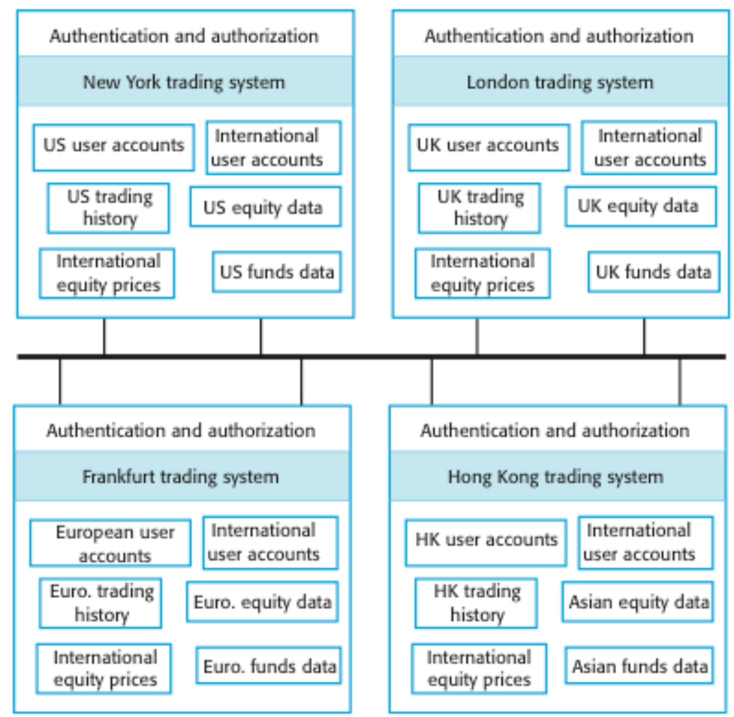
A layered protection architecture



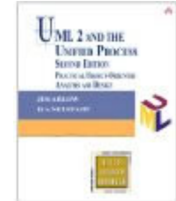
Distribution

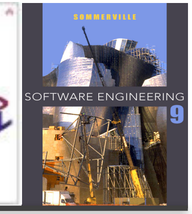


- Distributing assets means that attacks on one system do not necessarily lead to complete loss of system service
- Each platform has separate protection features and may be different from other platforms so that they do not share a common vulnerability
- Distribution is particularly important if the risk of denial of service attacks is high



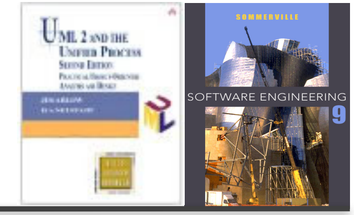
Security guidelines





Security tactics
Base security decisions on an explicit security policy
Avoid a single point of failure
Fail securely
Balance security and usability
Log user actions
Use redundancy and diversity to reduce risk
Compartmentalize your assets
Design for recoverability
Design for deployment
Validate all inputs

Security guidelines 1-3



✧ Base decisions on an explicit security policy

- Define a security policy for the organization that sets out the fundamental security requirements that should apply to all organizational systems.

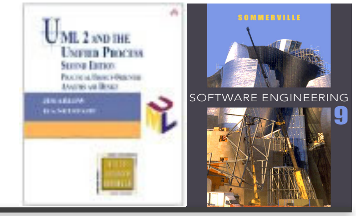
✧ Avoid a single point of failure

- Ensure that a security failure can only result when there is more than one failure in security procedures. For example, have password and question-based authentication.

✧ Fail securely

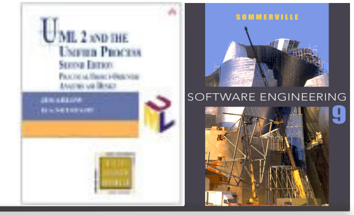
- When systems fail, for whatever reason, ensure that sensitive information cannot be accessed by unauthorized users even although normal security procedures are unavailable.

Security guidelines 4-6



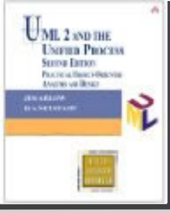
- Balance security and usability
  - Try to avoid security procedures that make the system difficult to use. Sometimes you have to accept weaker security to make the system more usable.
- Log user actions
  - Maintain a log of user actions that can be analyzed to discover who did what. If users know about such a log, they are less likely to behave in an irresponsible way.
- Use redundancy and diversity to reduce risk
  - Keep multiple copies of data and use diverse infrastructure so that an infrastructure vulnerability cannot be the single point of failure.

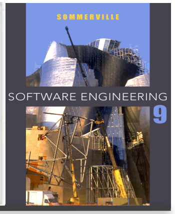
Security guidelines 7-10



- Compartmentalize your assets
  - Organize the system so that assets are in separate areas and users only have access to the information that they need rather than all system information.
- Design for recoverability
  - Design the system to simplify recoverability after a successful attack.
- Design for deployment
  - Design the system to avoid deployment problems
- Validate all inputs
  - Check that all inputs are within range so that unexpected inputs cannot cause problems.

System survivability

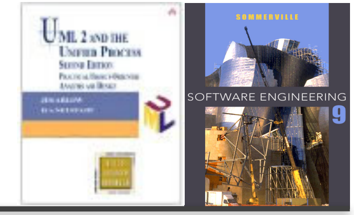




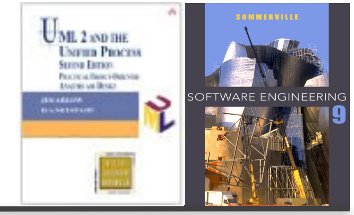
- Survivability = system ability to deliver essential services whilst it is under attack or after part of it was damaged.

- **Resistance**
  - Avoiding problems by building capabilities into the system to resist attacks
- **Recognition**
  - Detecting problems by building capabilities into the system to detect attacks and failures and assess the resultant damage
- **Recovery**
  - Tolerating problems by building capabilities into the system to deliver services whilst under attack

Key points



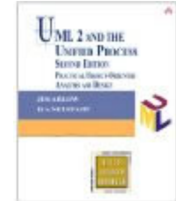
- Design for security involves
  - Protection
  - Distribution
- Layered protection architecture
  - Platform-level protection
  - Application-level protection
  - Record-level protection
- Security guidelines
- Survivability



Design for Performance, Modifiability and Usability

Lecture 6/Part 3

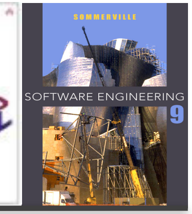
Performance tactics



- **Introduce concurrency.** If requests can be processed in parallel, the blocked time can be reduced. But it is necessary to understand well the concurrency effects.
- **Control the use of resources.** This includes both **computational** resources and **data**. Namely, balance the load, control access, scheduling (via priority), cache, maintain multiple copies to reduce contention.
- **Increase available resources.** Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency.

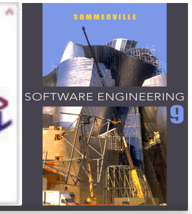
What is the role of vertical and horizontal scalability here?

Modifiability tactics – Defer binding time



- **Runtime registration** supports plug-and-play operation at the cost of additional overhead to manage the registration. Publish/subscribe registration, for example, can be implemented at either runtime or load time.
- **Configuration files** are intended to set parameters at startup.
- **Polymorphism** allows late binding of method calls.
- **Component replacement** allows load time binding.
- **Adherence to defined protocols** allows runtime binding of independent processes.

Modifiability tactics – Prevent ripple effects

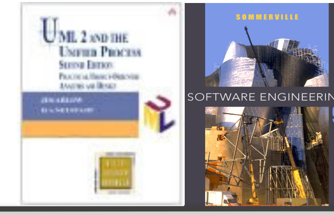


- **A ripple effect** from a modification is the necessity of making changes to modules not directly affected by it.
  - For instance, if module A is changed to accomplish a particular modification, then module B is changed only because of the change to module A. B has to be modified because it depends, in some sense, on A.
- **Hide information.** Information hiding is the decomposition of the responsibilities for an entity (a system or some decomposition of a system) into smaller pieces and choosing which information to make visible.

Usability tactics – Design-time tactics

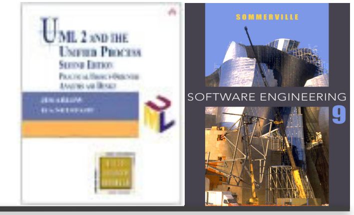
- **Separate the user interface from the rest of the application.** Localizing expected changes is the rationale for semantic coherence.
- Since the user interface is expected to change frequently both during the development and after deployment, maintaining the user interface code separately will localize changes to it.

Usability tactics – Runtime tactics



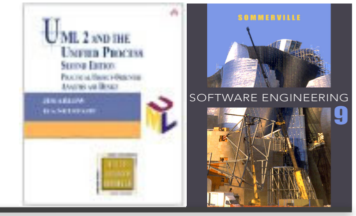
- **Maintain a model of the task.** The task model is used to determine context so the system can have some idea of what the user is attempting and provide various kinds of assistance.
  - For example, knowing that sentences usually start with capital letters would allow an application to correct a lower-case letter in that position.
- **Maintain a model of the user.** The model determines the user's knowledge of the system, the user's behavior in terms of expected response time, and other aspects specific to a user or a class of users.
  - For example, maintaining a user model allows the system to pace scrolling so that pages do not fly past faster than they can be read.
- **Maintain a model of the system.** The model determines the expected system behavior so that appropriate feedback can be given to the user.

Quality conflicts

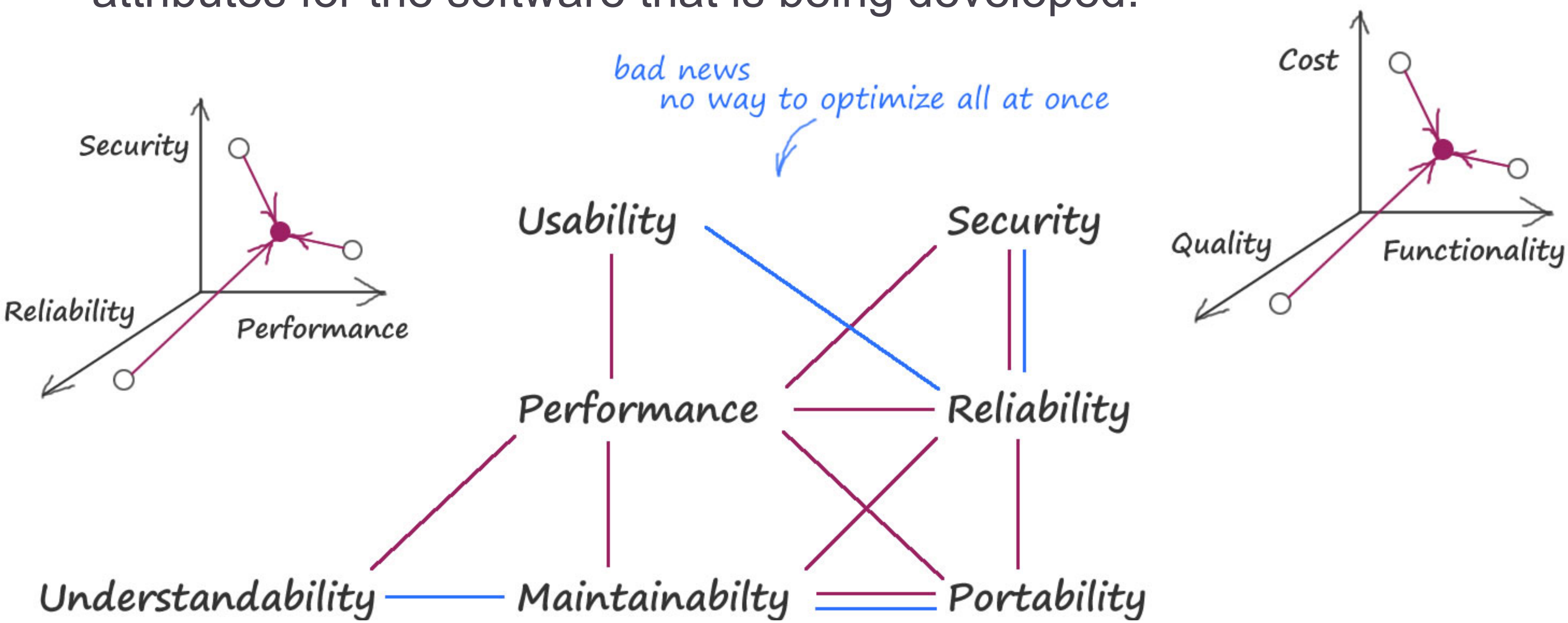


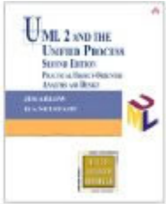
- ✧ Within complex systems, quality attributes can never be achieved in isolation.
  - The achievement of any one will have an effect, sometimes positive and sometimes negative, on the achievement of others.
- ✧ For example, almost every quality attribute negatively affects performance.
  - Reliability. Redundancy together with a voting schema delays system response.
  - Portability. The main technique for achieving portable software is to isolate system dependencies, which introduces overhead into the system's execution.

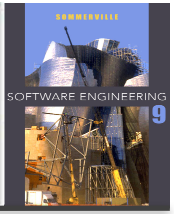
Quality conflicts



- It is not possible for any system to be optimized for all of these attributes.
- The quality plan should therefore define the most important quality attributes for the software that is being developed.



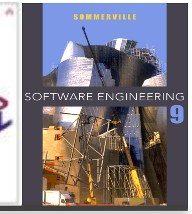




UML Class Diagram in Design

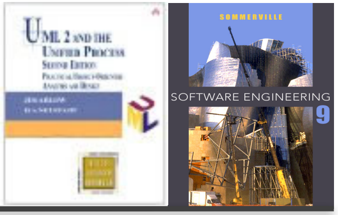
Lecture 6/Part 4

Design model



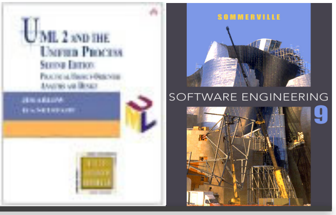
- Design model is a refinement of an analysis model to such a degree that it can be implemented
  - In MDD design models include all implementation details and can be automatically translated into code
- In OO design models:
  - All attributes are completely specified
  - Analysis operations become fully 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

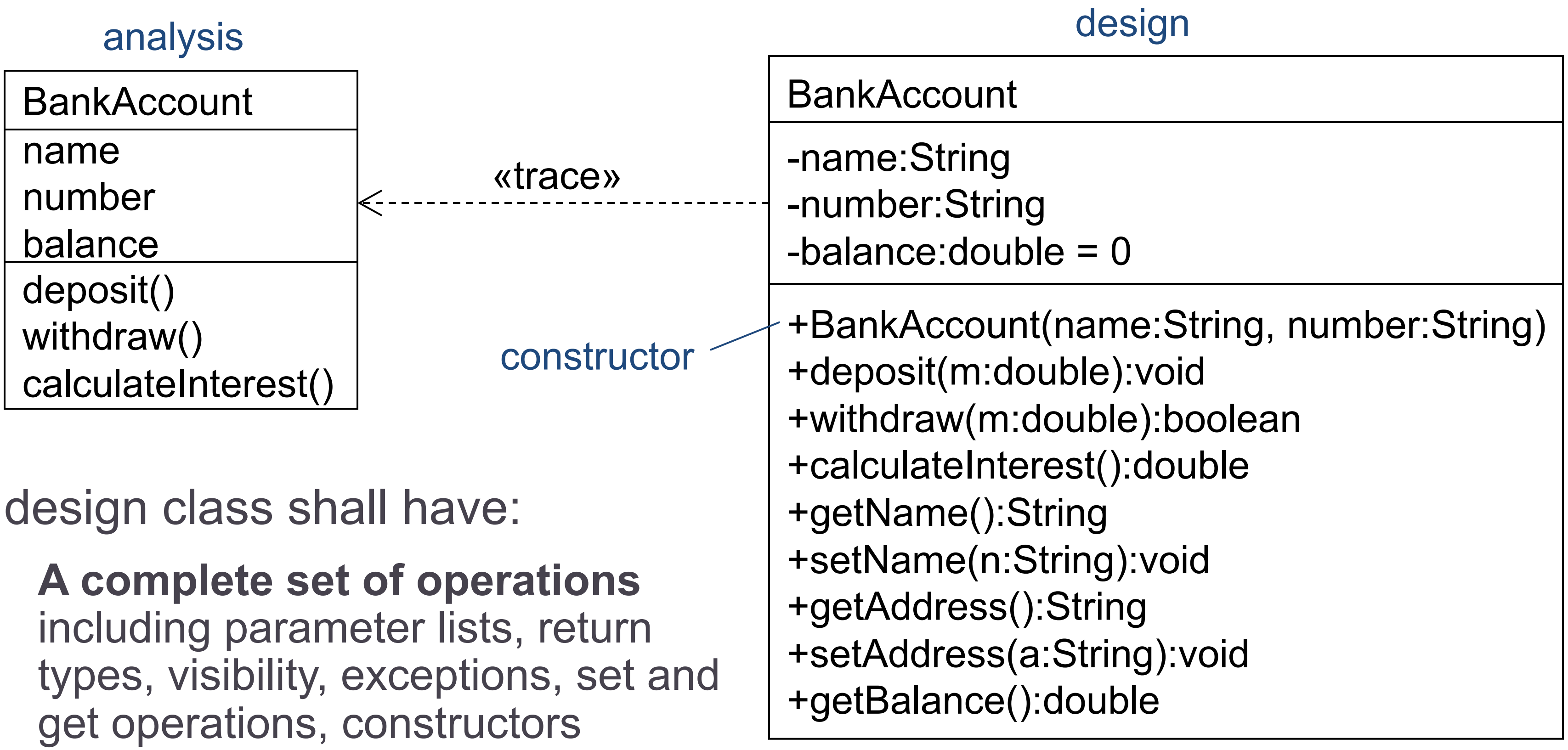
Analysis vs. design model



- 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
- Otherwise, we can make it with only a design model

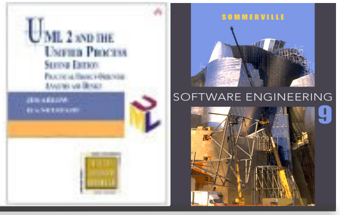
Anatomy of a design class



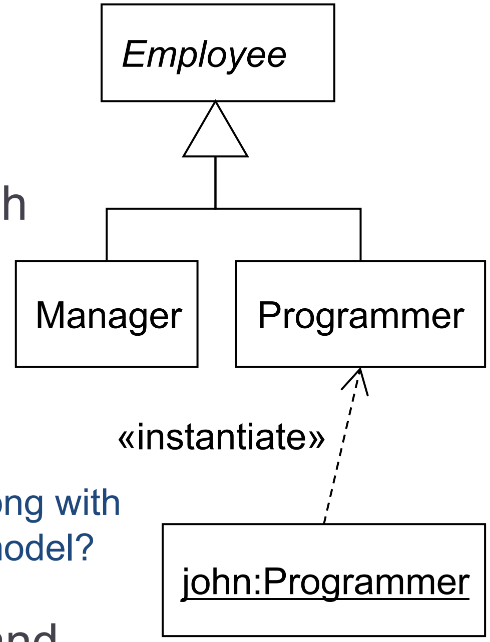


- A design class shall have:
  - **A complete set of operations** including parameter lists, return types, visibility, exceptions, set and get operations, constructors
  - **A complete set of attributes** including types and default values

High cohesion, low coupling

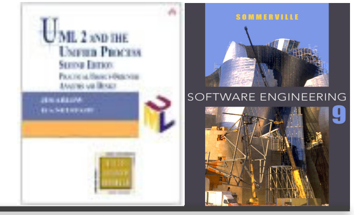


- High cohesion
  - Each class should have a set of operations that support the intent of the class, no more and no less
  - Each class should model a single abstract concept
- Low coupling
  - A particular class should be associated with just enough other classes to allow it to realise its responsibilities
  - 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

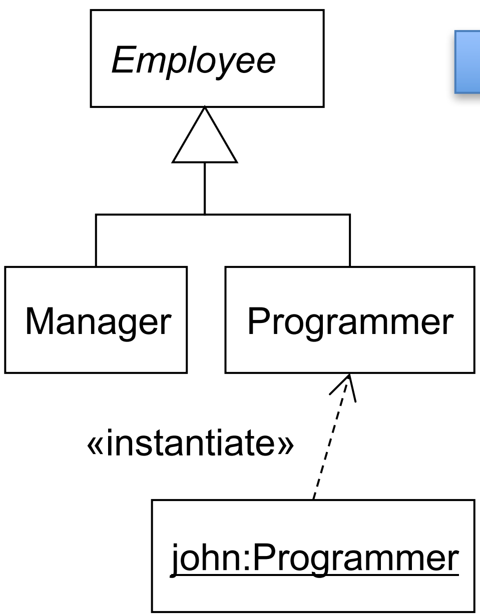


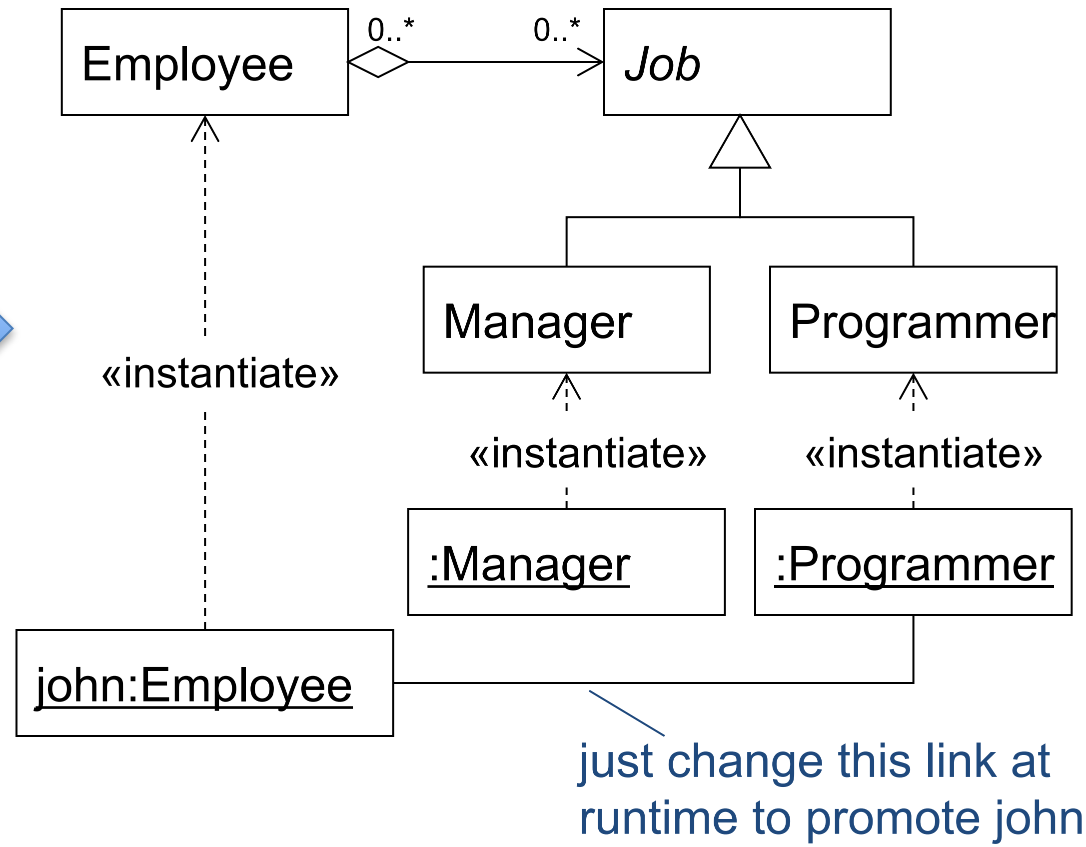
What is wrong with this model?

Aggregation vs. inheritance

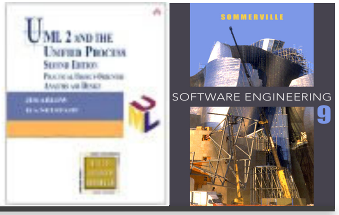


- An employee **has** a job, not **is** a job.
- An employee can have more jobs.

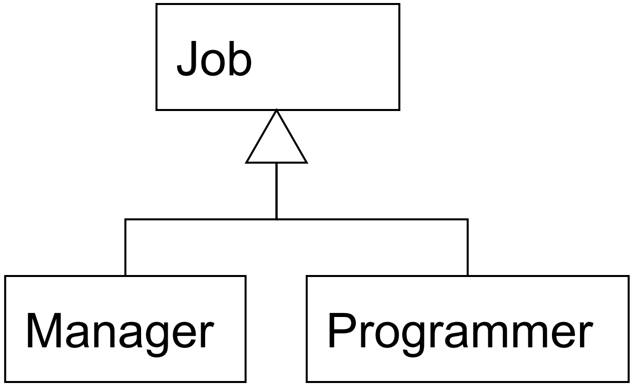




Inheritance vs. interface realization

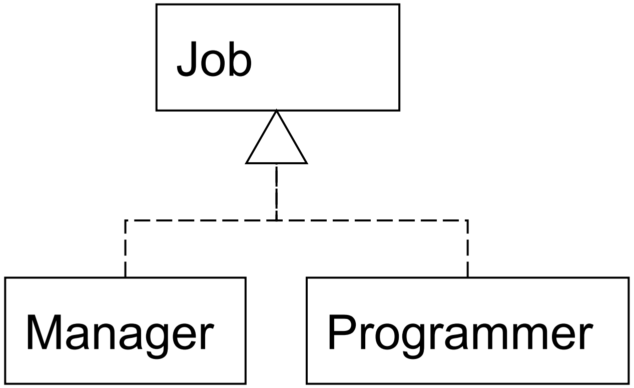


- With **inheritance** we get two things:
  - Interface – the public operations of the base classes
  - Implementation – the attributes, relationships, operations of the class



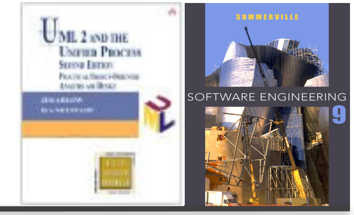
Use inheritance when we want to *inherit implementation*.

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



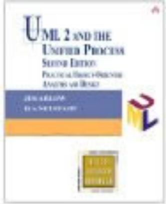
Use interface realization when we want to *define a contract*.

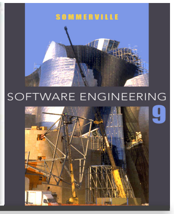
Key points (design classes)



- 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

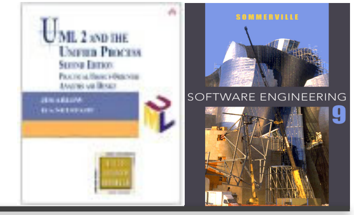
Design relationships

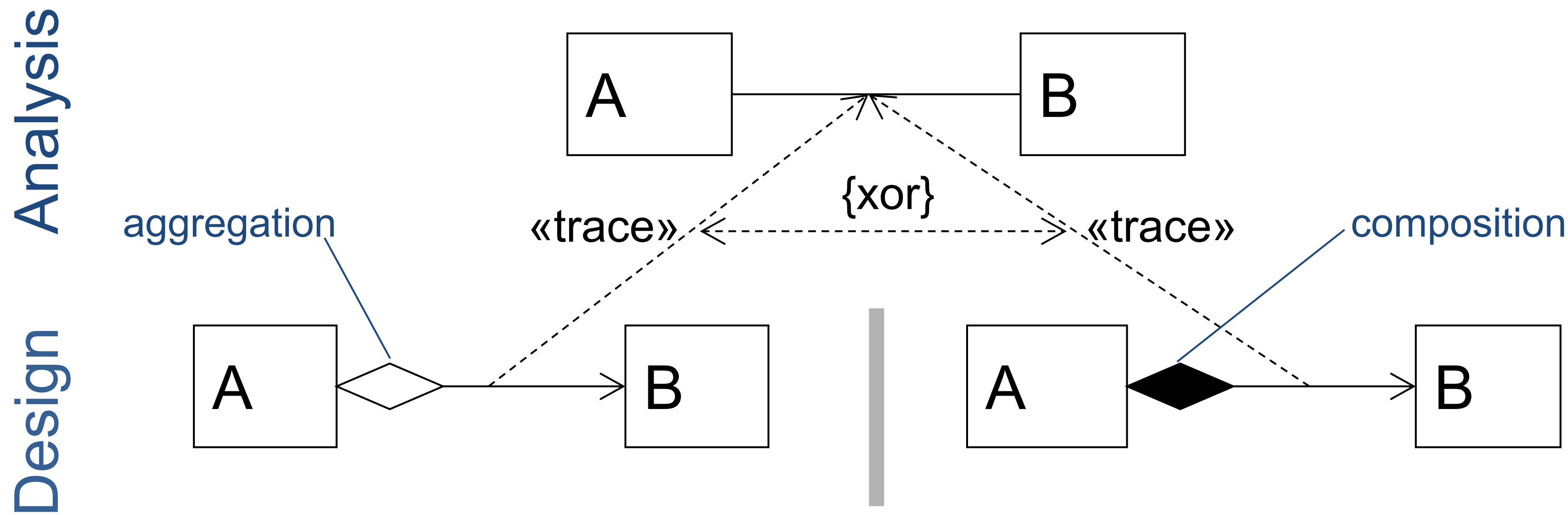




- Refining analysis associations to design associations involves several procedures:
  - refining associations to aggregation or composition
  - implementing one-to-many associations
  - implementing many-to-one associations
  - implementing many-to-many associations
  - implementing bidirectional associations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

Aggregation and composition

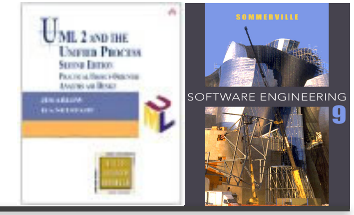




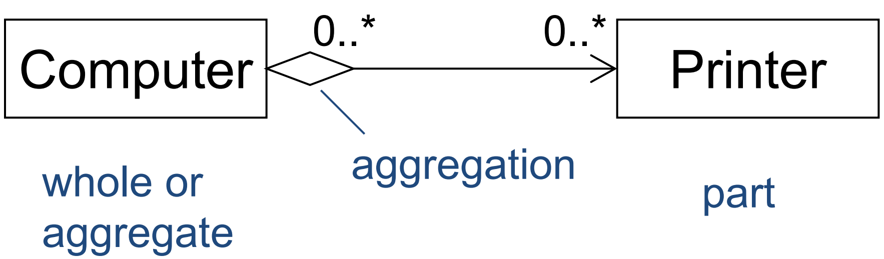
Some objects are weakly related like a computer and its peripherals

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

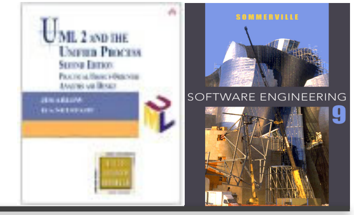
At any one point in time a Printer is connected to 0 or more Computers

The Printer exists even if there are no Computers

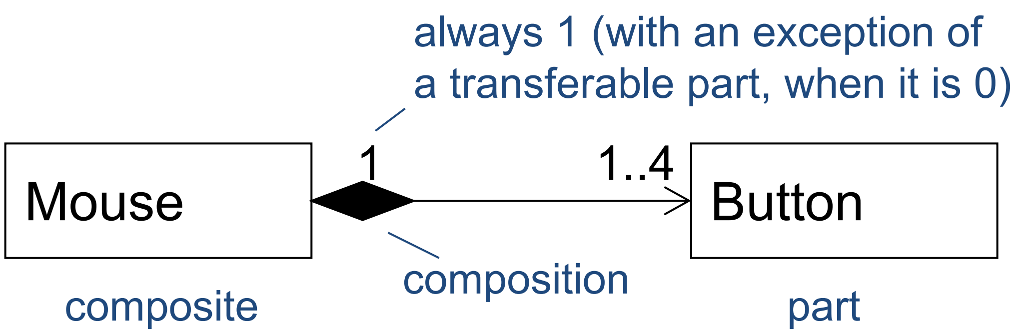
The Printer is independent of the Computer

- 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



composition is a strong form of aggregation

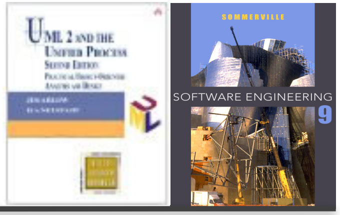


The buttons have no independent existence. If we destroy the mouse, we destroy the buttons. They are an integral part of the mouse

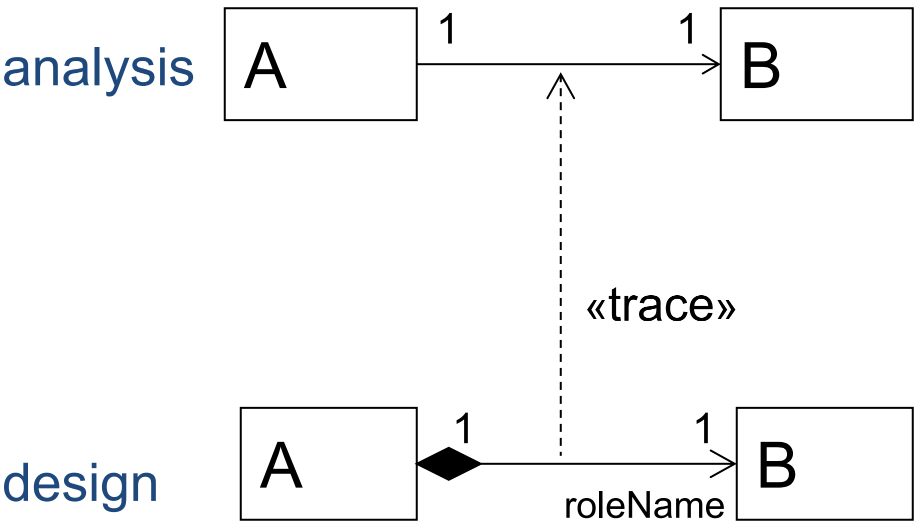
Each button can belong to exactly 1 mouse

- 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 (the exception above)

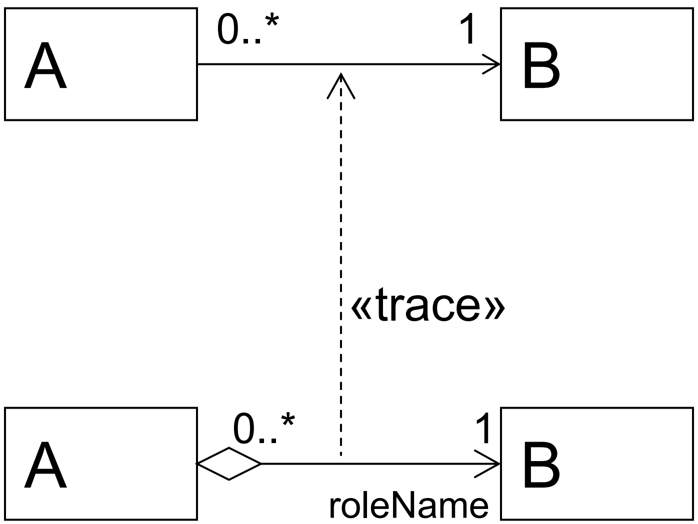
One-to-one and many-to-one associations



One to one

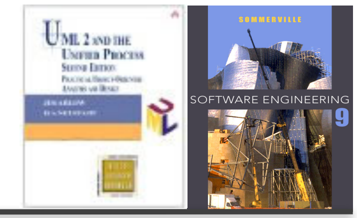


Many to one

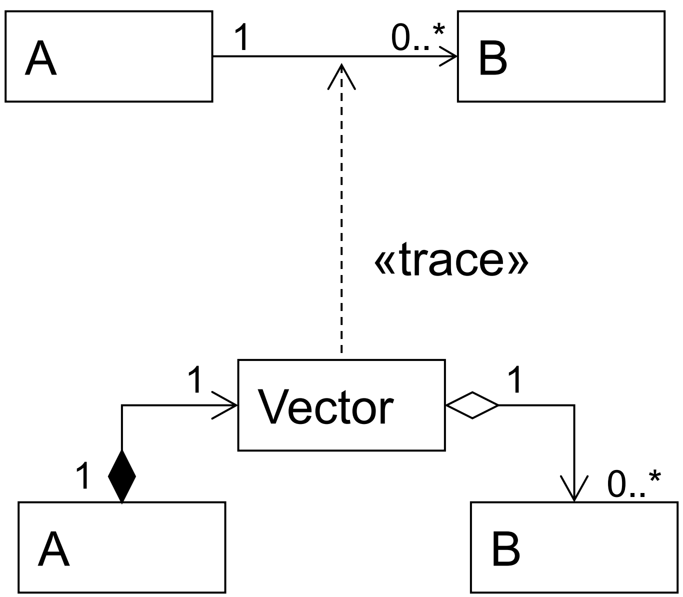


- One-to-one associations in analysis usually imply single ownership and usually refine to compositions
- Many-to-one relationships in analysis imply shared ownership and are refined to aggregations

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

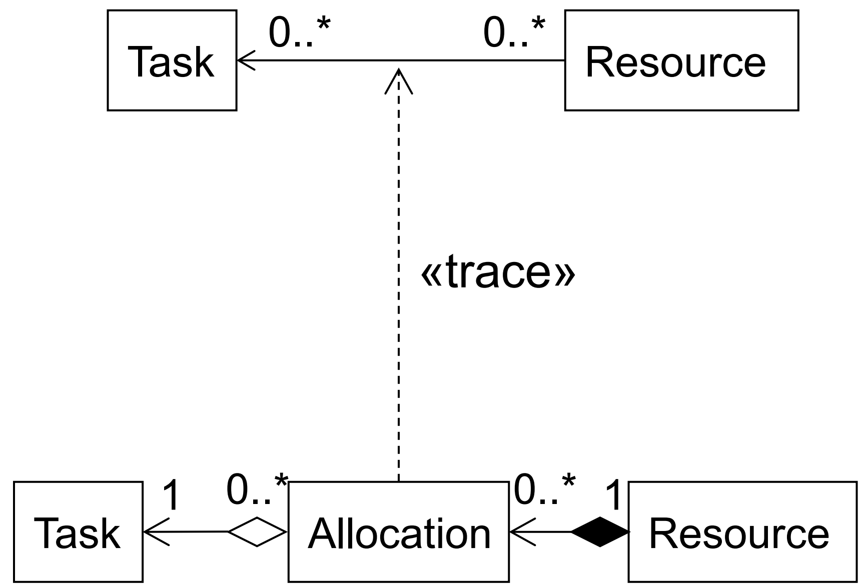


One to many



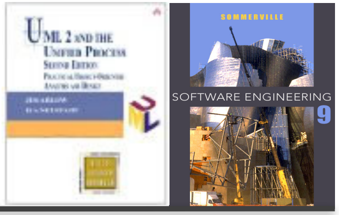
- Collection classes instances store a collection of object references to objects of the target and provide methods for operating the collection
- In Java in the java.util library

Many to many

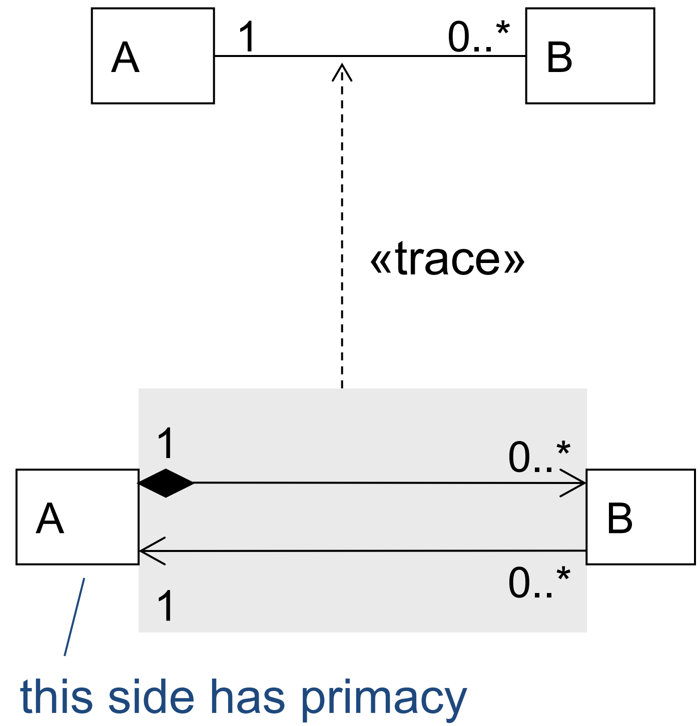


- Many-to-many associations may be (but do not have to be) refined into intermediate design classes.

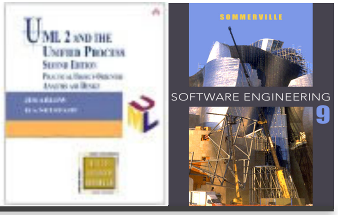
Bi-directional associations



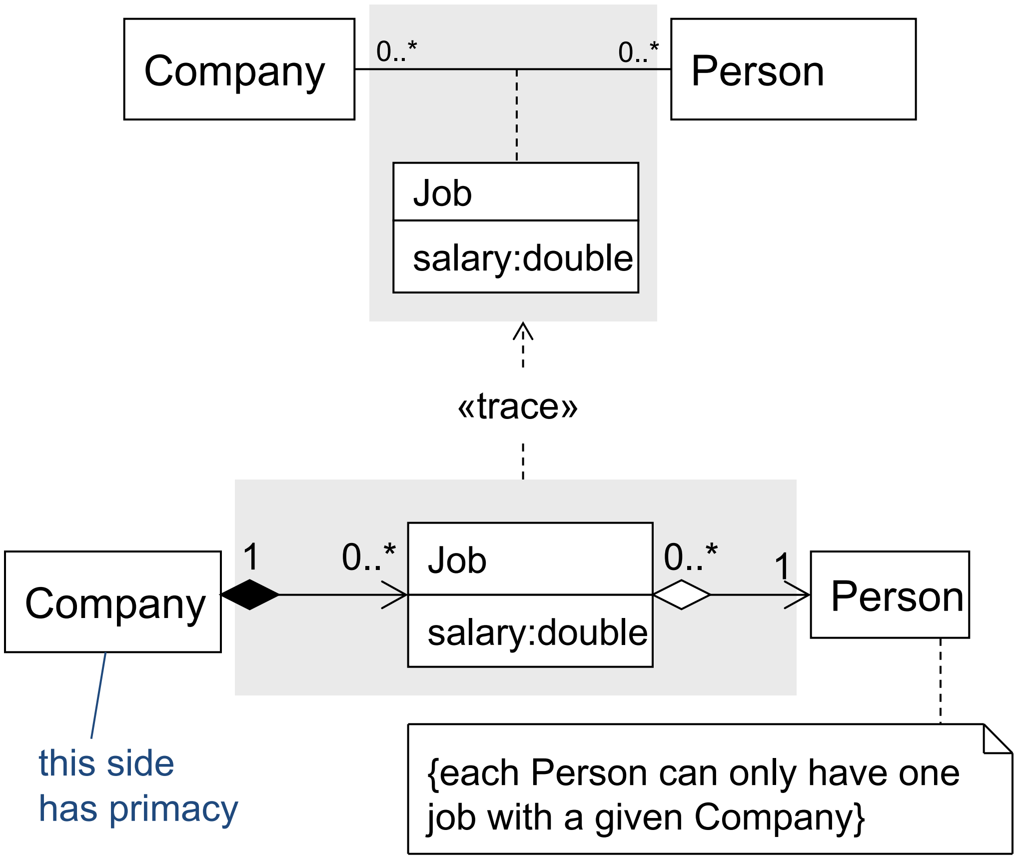
- There is no commonly used OO language that directly supports bi-directional associations
- We must resolve each bi-directional associations into two unidirectional associations
- Again, we must decide which side of the association should have primacy and use composition, aggregation and navigability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



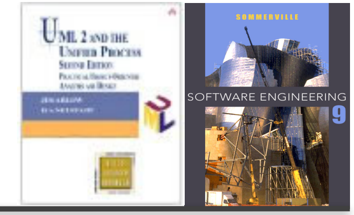
Association classes



- 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