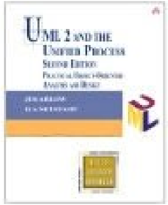
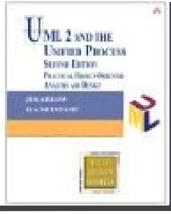
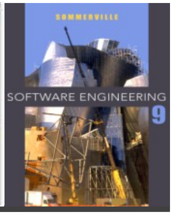


System Design

Lecture 7

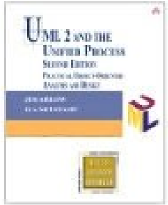
Outline





- ✧ Introduction to system design
- ✧ Design for dependability
- ✧ Design for security
- ✧ Design for performance, modifiability, testability and usability

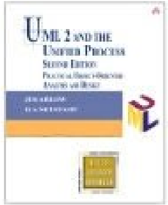
- ✧ UML Class Diagram in Design
  - Design classes
  - Design relationships



Introduction to System Design

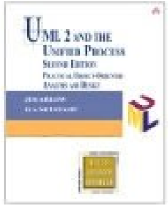
Lecture 7/Part 1

Design purpose



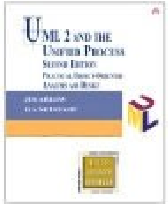
- Decide how the system's functions are to be implemented and how non-functional requirements are to be ensured
- Decide on strategic design issues such as persistence, distribution etc.
- Create policies to deal with tactical design issues

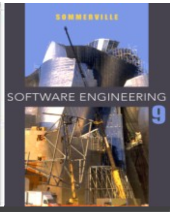
Design model



- 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 including type, visibility and default values
  - Analysis operations become fully specified operations (methods) with a return type and parameter list
  - Many new classes are added to include implementation details, such as utility classes, middleware classes or GUI classes
- Design models are programming-language specific
  - Multiple inheritance, templates, nested classes, collections

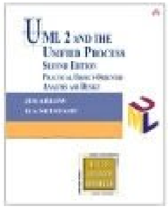
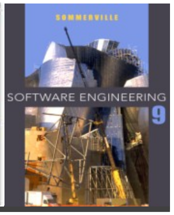
Analysis vs. design model





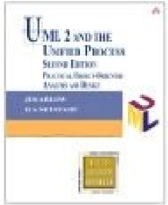
- ✧ A design model may contain 10 to 100 times as many classes as the analysis model
  - The analysis model helps us to see the big picture without getting lost in implementation details
- ✧ We need to maintain both models if:
  - It is a big system (>200 design classes)
  - It has a long expected lifespan
  - 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

Design best practices

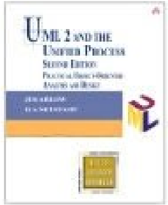
- A system design consists of a collection of decisions that help to control different attributes of software quality.
    - The design aims to ensure achievement of system functionality, but whenever there are different ways to achieve the functionality, the impact of each design decision on software 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 example.

Design patterns



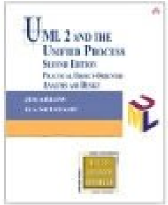
- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

Pattern elements



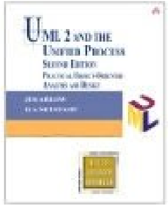
- Name
  - A meaningful pattern identifier.
- Problem description.
- Solution description.
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- Consequences
  - The results and trade-offs of applying the pattern.

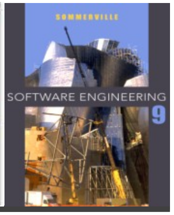
The Observer pattern

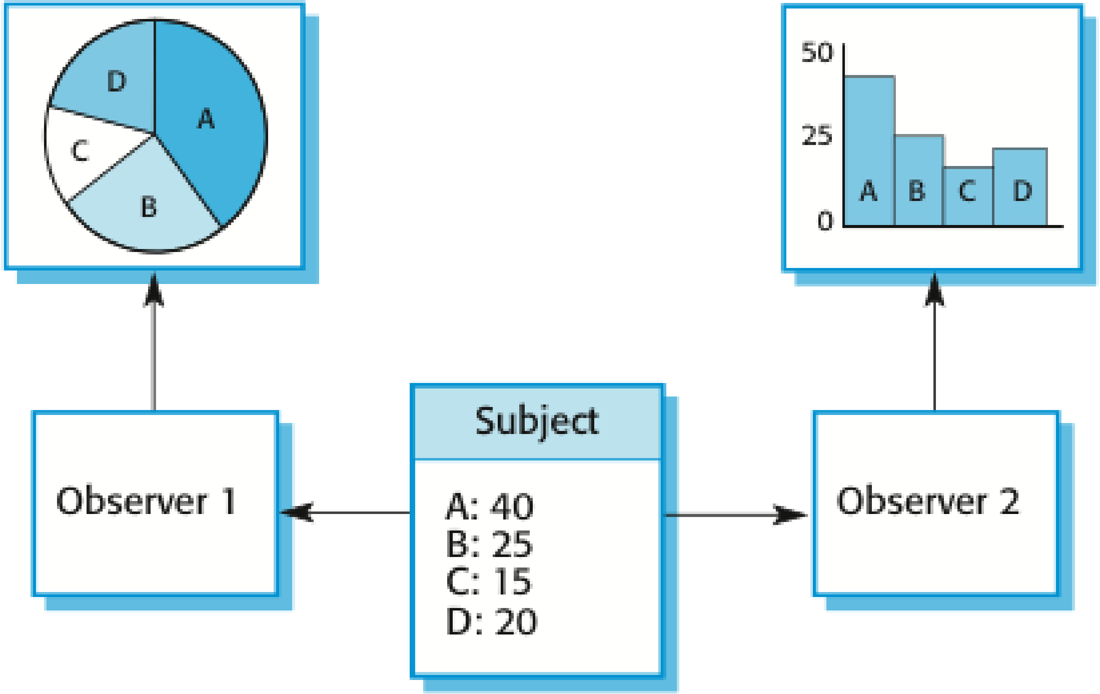


- ✧ Name
  - Observer.
- ✧ Description
  - Separates the display of object state from the object itself.
- ✧ Problem description
  - Used when multiple displays of state are needed.
- ✧ Solution description
  - See slide with UML description.
- ✧ Consequences
  - Optimisations to enhance display performance are impractical.

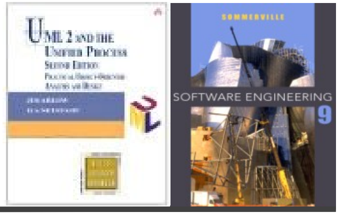
Multiple displays using the Observer pattern

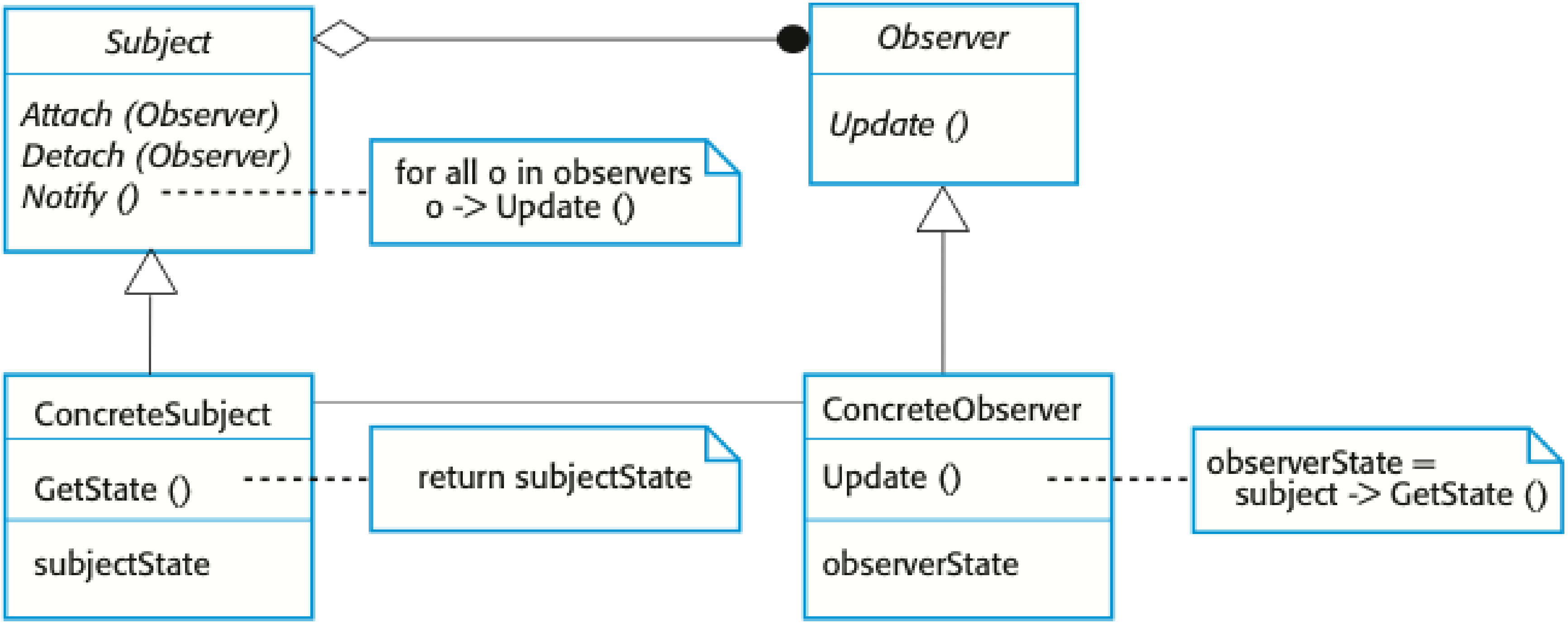




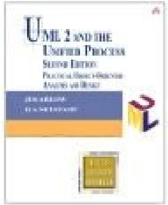


A UML model of the Observer pattern



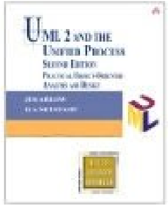


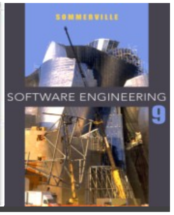
Design problems



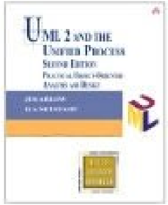
- To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
  - Tell several objects that the state of some other object has changed (Observer pattern).
  - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
  - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
  - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

Design for non-functional qualities





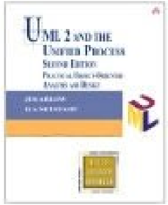
- Design patterns help us to implement specific functionality while maintaining high code quality
  - Respect of design patterns improves system maintainability
- What if also other non-functional qualities are of high importance?
- Are there any “patterns” for dependability, performance, security, etc.?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



Design for Dependability

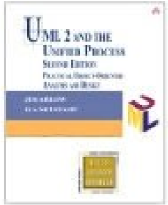
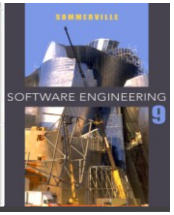
Lecture 7/Part 2

Topics covered



- ✧ Dependable processes
  - How the use of dependable processes leads to dependable systems
- ✧ Redundancy and diversity
  - Fundamental approaches to achieve fault tolerance.
- ✧ Dependable systems architectures
  - Architectural patterns for software fault tolerance

Software dependability

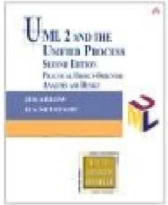



- In general, software customers expect all software to be dependable. However, for non-critical applications, they may be willing to accept some system failures.
- Some applications (critical systems) have very high dependability requirements and special software engineering techniques may be used to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

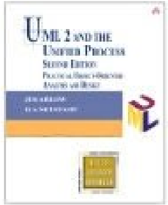
Dependability achievement

- ✧ Fault avoidance
  - ▪ The system is developed in such a way that human error is avoided and thus system faults are minimised.
  - ▪ The development process is organised so that faults in the system are detected and repaired before delivery to the customer.
- ✧ Fault detection
  - ▪ Verification and validation techniques are used to discover and remove faults in a system before it is deployed.
- ✧ Fault tolerance
  - ▪ The system is designed so that faults in the delivered software do not result in system failure.

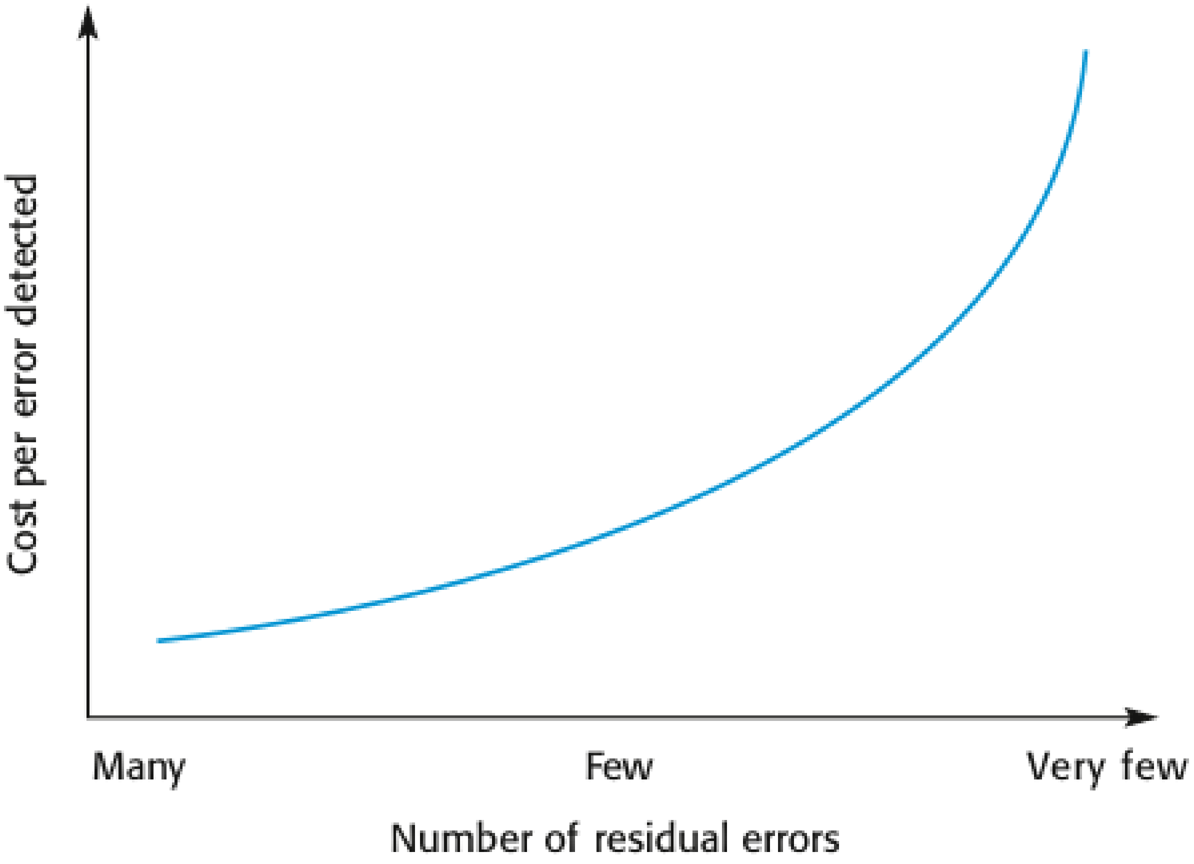
Dependable processes for fault avoidance



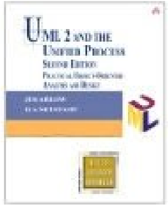
- To ensure a minimal number of software faults, it is important to have a well-defined, repeatable software process.
- A well-defined repeatable process is one that does not depend entirely on individual skills; rather can be enacted by different people.
- Regulators use information about the process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

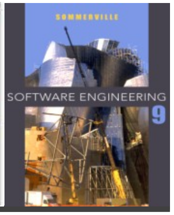


Static fault detection and its costs



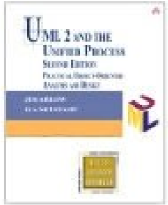
Dynamic fault detection tactics





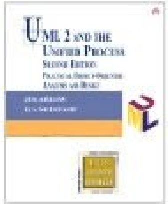
- **Ping/echo.** One component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny.
- **Heartbeat (dead man timer).** In this case one component emits a heartbeat message periodically and another component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified.
- **Exceptions.** One method for recognizing faults is to encounter an exception, which is raised when one of the fault classes is recognized.

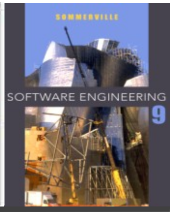
Fault tolerance



- 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.

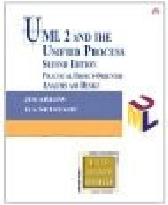
Diversity and redundancy





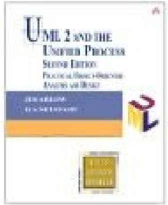
- ✧ Redundancy
  - Keep more than 1 version of a critical component available so that if one fails then a backup is available.
  - E.g. switch to backup servers automatically if failure occurs.
- ✧ Diversity
  - Provide the same functionality in different ways so that they will not fail in the same way.
  - 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.

Dependable system architectures



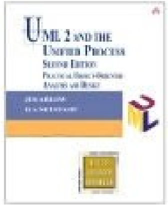
- ✧ Dependable systems architectures are used in situations where fault tolerance is essential. These architectures are generally all based on redundancy and diversity.
- ✧ Examples of situations where dependable architectures are used:
  - Flight control systems, where system failure could threaten the safety of passengers
  - Reactor systems where failure of a control system could lead to a chemical or nuclear emergency
  - Telecommunication systems, where there is a need for 24/7 availability.

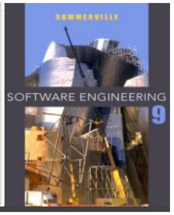
Fault tolerance and recovery tactics (1)



- **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.
- **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), and the rest are discarded.
- **Passive redundancy** (warm restart/dual redundancy/ triple redundancy). One component (the primary) responds to events and informs the other components (the standbys) of state updates they must make.

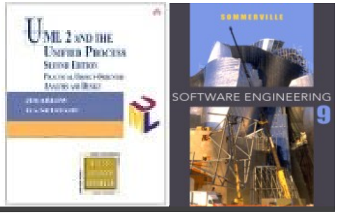
Fault tolerance and recovery tactics (2)





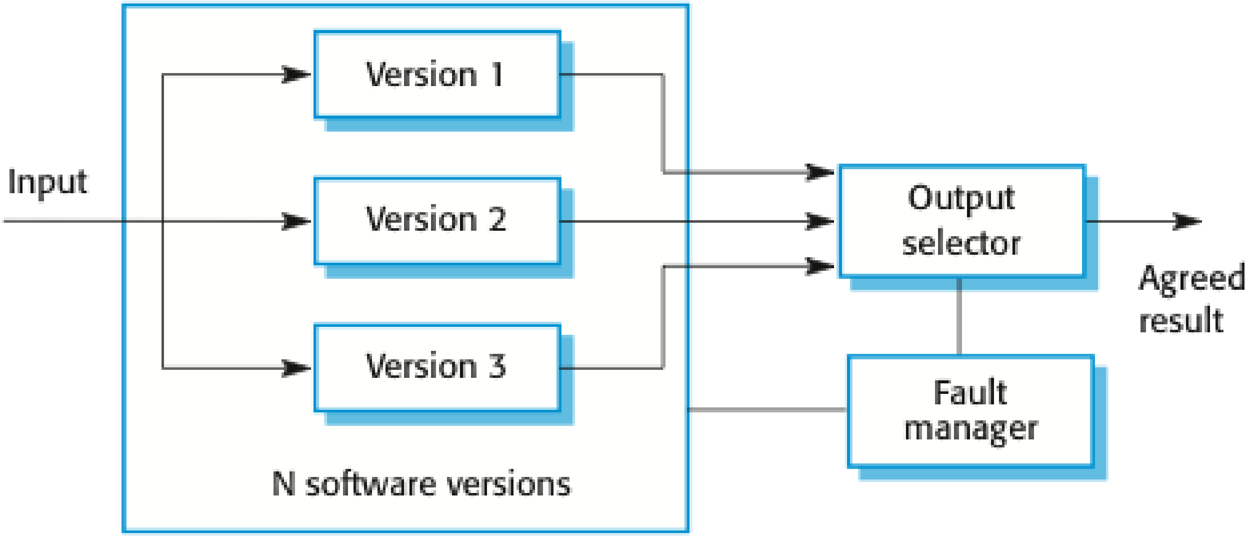
- **Spare.** A standby spare computing platform is configured to replace many different failed components. It must be rebooted to the appropriate software configuration and have its state initialized when a failure occurs.
- **Shadow operation.** A previously failed component may be run in "shadow mode" for a short time to make sure that it mimics the behavior of the working components before restoring it to service.
- **Checkpoint/rollback.** A checkpoint is a recording of a consistent state created either periodically or in response to specific events, to which the system can be restored.

N-version programming pattern

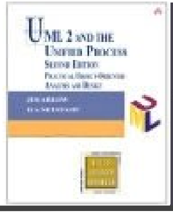


- Combines different dependability tactics
- Multiple versions of a software system carry out computations at the same time.
  - There should be an odd number of versions involved, typically 3.
  - The versions should be designed and implemented by different teams, since it is assumed that different teams are unlikely to make the same mistakes.
- The results are compared using a voting system and the majority result is taken to be the correct result.
- Approach derived from the notion of triple-modular redundancy, as used in hardware systems.

N-version programming

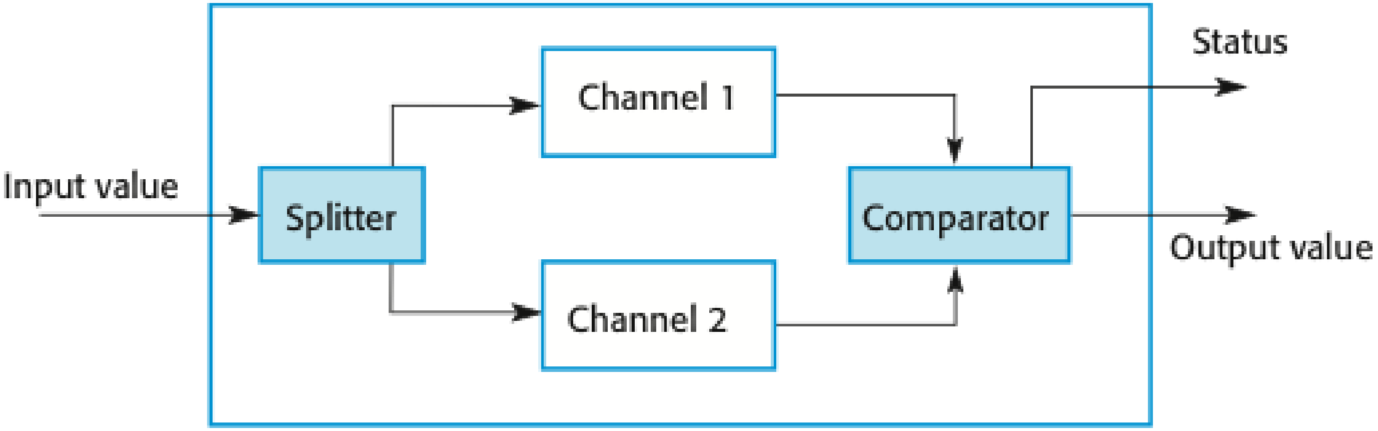


Self-monitoring architectures

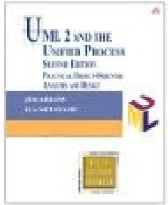


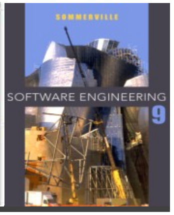
✧ Multi-channel architectures with diverse SW and HW in each channel.

- The same computation is carried out on each channel and the results compared.
- The system monitors its own operations and takes action if inconsistencies are detected.



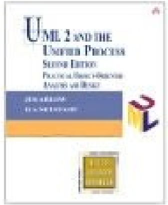
Protection systems





- A specialized system that is associated with some other control 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/pressure are too high
- Protection systems are redundant because they include monitoring and control capabilities that replicate those in the control software.
- Protection systems should be diverse and use different technology from the control software.

Dependable programming



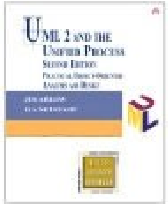
✧ Good programming practices can be adopted that help reduce the incidence of program faults.

✧ These programming practices support

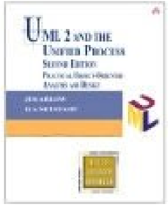
- Fault avoidance
- Fault detection
- Fault tolerance

✧ Dependable programming tactics will be detailed in the next lecture where we discuss the Implementation stage.

Key points



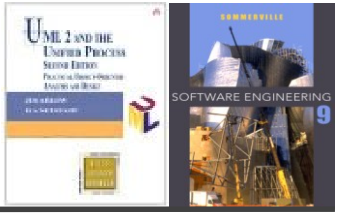
- ✧ Dependability in a program can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment, and by including fault tolerance facilities.
- ✧ The use of redundancy and diversity in hardware, software processes and software systems is essential for the development of dependable systems.
- ✧ The use of a well-defined, repeatable process is essential if faults in a system are to be minimized.
- ✧ Dependable system architectures are system architectures that are designed for fault tolerance. Architectural styles that support fault tolerance include protection systems, self-monitoring architectures and N-version programming.



Design for Security

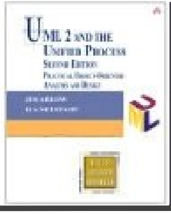
Lecture 7/Part 3

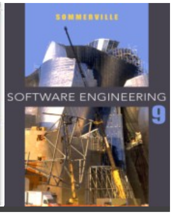
Topics covered



- Architectural design
- Design guidelines for security
  - Guidelines that help you design a secure system
- System survivability
  - Allow the system to deliver essential services when under attack

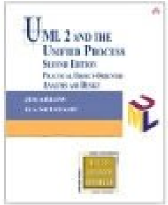
Architectural design

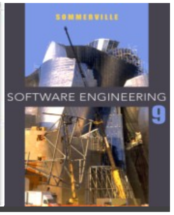




- Two fundamental issues have to be considered when designing an architecture for security.
  - Protection
    - How should the system be organised so that critical assets can be protected against external attack?
  - Distribution
    - How should system assets be distributed so that the effects of a 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.

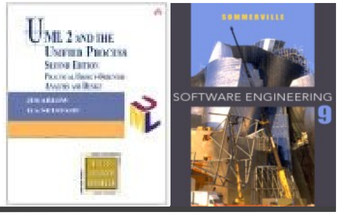
Protection

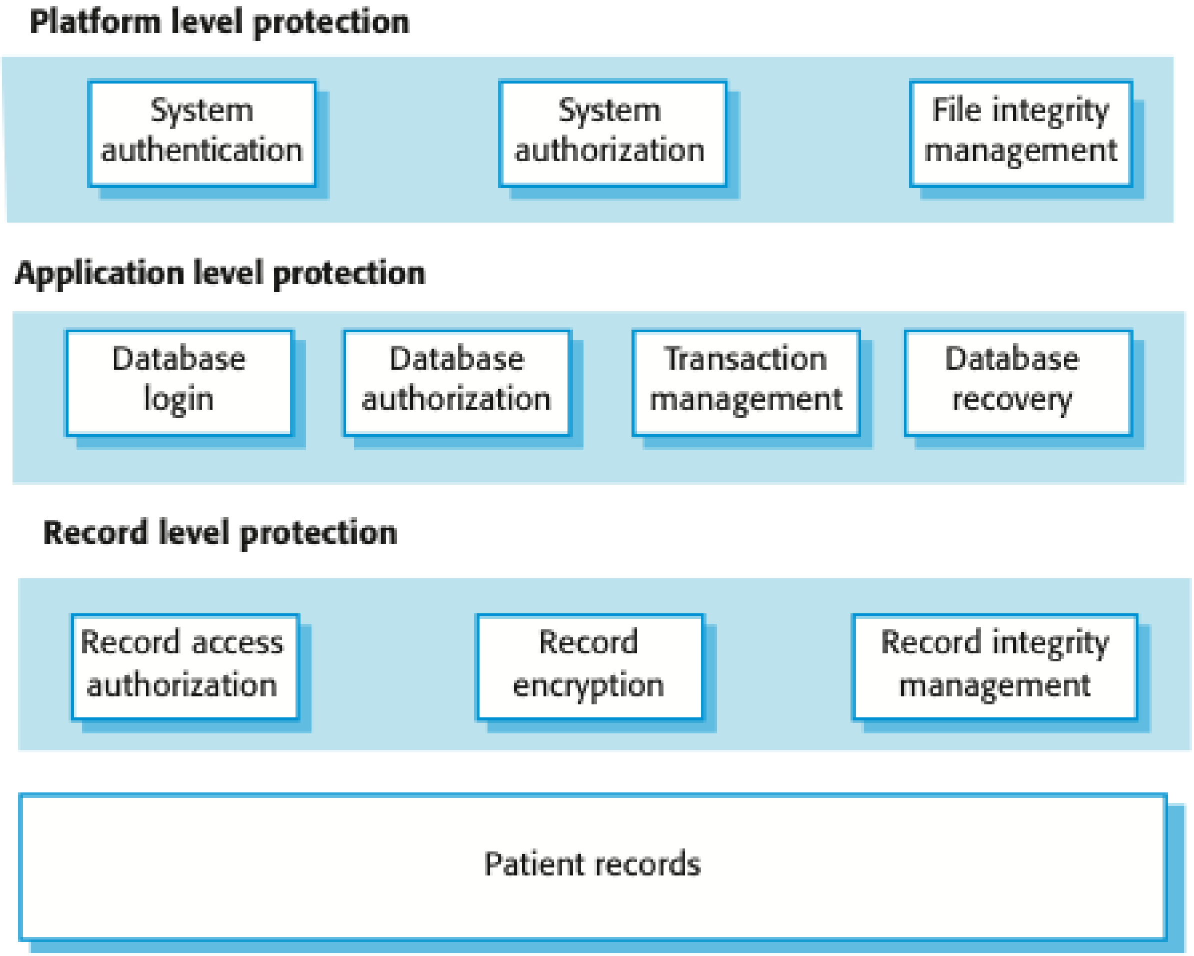




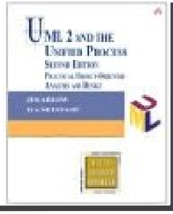
- ✧ Platform-level protection
  - Top-level controls on the platform on which a system runs.
- ✧ Application-level protection
  - Specific protection mechanisms built into the application itself e.g. additional password protection.
- ✧ Record-level protection
  - Protection that is invoked when access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

A layered protection architecture



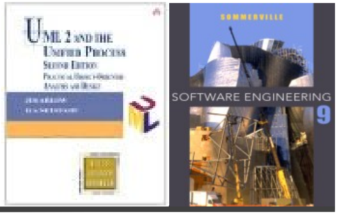


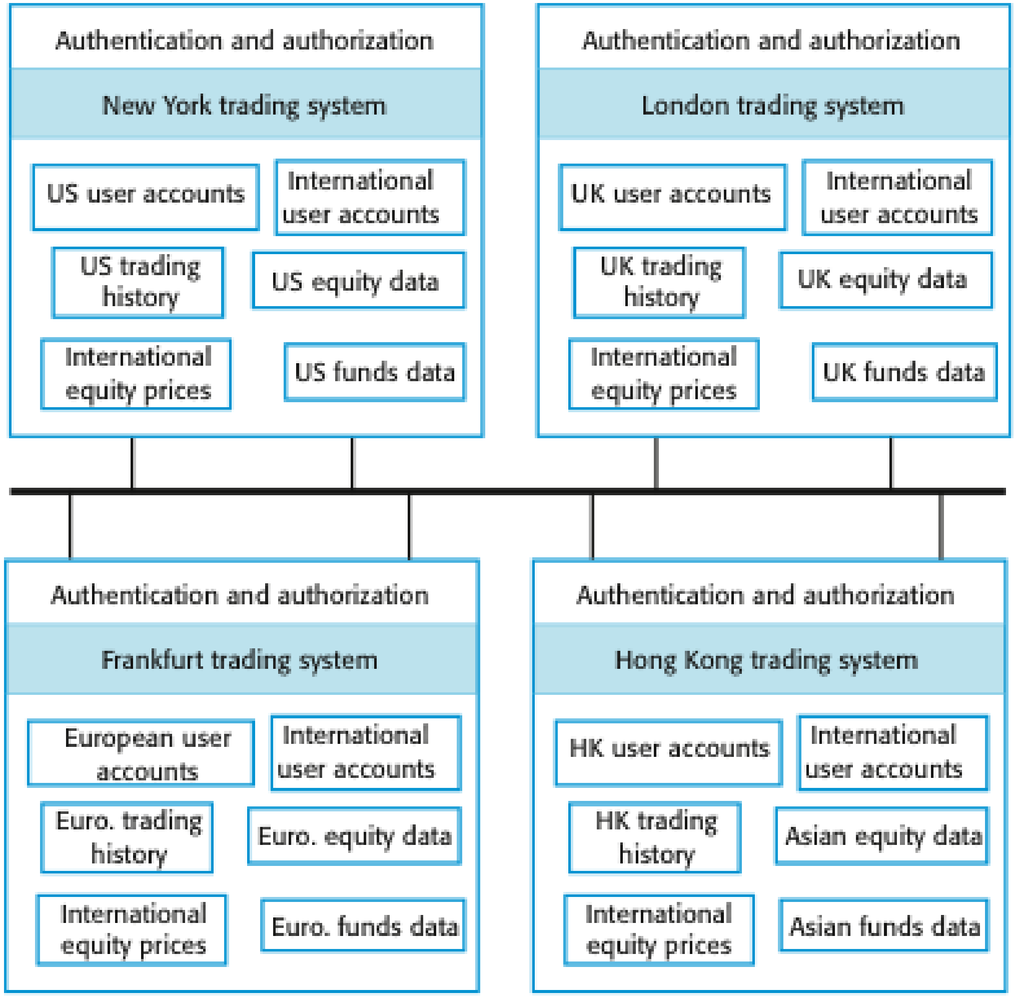
Distribution



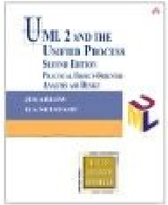
- Distributing assets means that attacks on one system do not necessarily lead to complete loss of system service
- Each platform has separate protection features and may be different from other platforms so that they do not share a common vulnerability
- Distribution is particularly important if the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

Distributed assets in an equity trading system



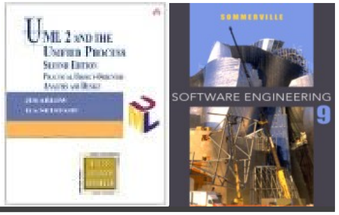


Security tactics



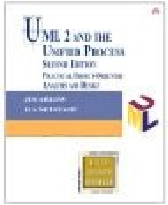
- ✧ Security tactics encapsulate good practice in secure systems design
- ✧ Security tactics serve two purposes:
  - They raise awareness of security issues in a software engineering team. Security is considered when design decisions are made.
  - They can be used as the basis of a review checklist that is applied during the system validation process.
- ✧ Tactics described here are applicable during software specification and design

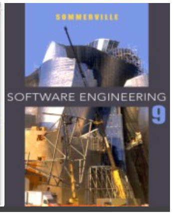
Tactics for secure systems engineering



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
Validate all inputs
Compartmentalize your assets
Design for deployment
Design for recoverability

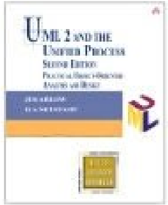
Design guidelines 1-3





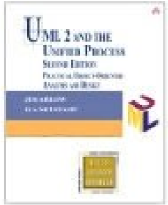
- Base decisions on an explicit security policy
  - Define a security policy for the organization that sets out the fundamental security requirements that should apply to all organizational systems.
- Avoid a single point of failure
  - Ensure that a security failure can only result when there is more 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.

Design guidelines 4-6



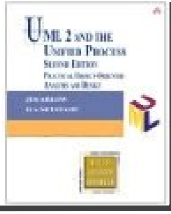
- ✧ Balance security and usability
  - Try to avoid security procedures that make the system difficult to use. Sometimes you have to accept weaker security to make the system more usable.
- ✧ Log user actions
  - Maintain a log of user actions that can be analyzed to discover who did what. If users know 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.

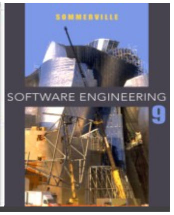
Design guidelines 7-10



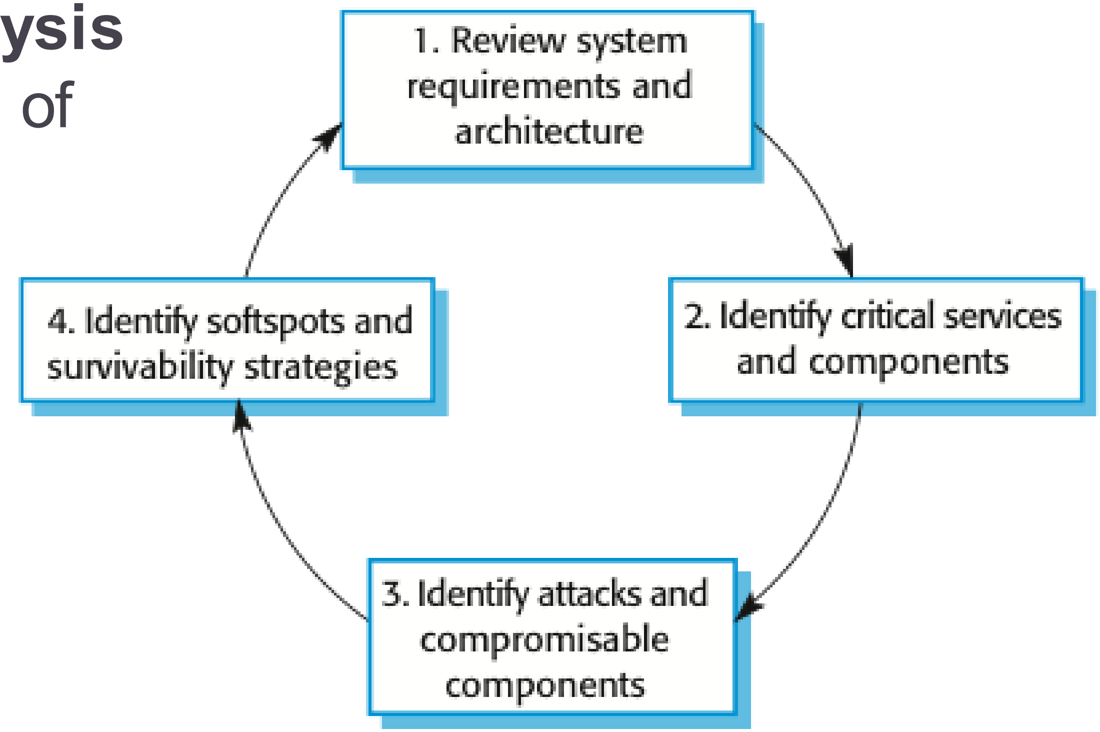
- Validate all inputs
  - Check that all inputs are within range so that unexpected inputs cannot cause problems.
- Compartmentalize your assets
  - Organize the system so that assets are in separate areas and users only have access to the information that they need rather than all system information.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

System survivability





- Survivability is an emergent system property that reflects the systems ability to deliver essential services whilst it is under attack or after part of the system was damaged
- **Survivability analysis** and should be part of the security engineering process



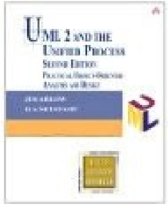
1. Review system requirements and architecture

2. Identify critical services and components

3. Identify attacks and compromisable components

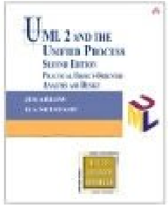
4. Identify softspots and survivability strategies

Survivability strategies

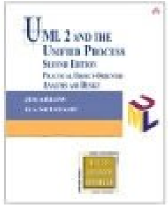


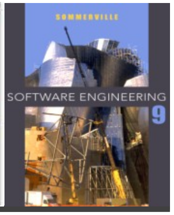
- Resistance
  - Avoiding problems by building capabilities into the system to resist attacks
- Recognition
  - Detecting problems by building capabilities into the system to detect attacks and failures and assess the resultant damage
- Recovery
  - Tolerating problems by building capabilities into the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

Key points



- ✧ Design for security involves architectural design, following good design practice and minimising the introduction of system vulnerabilities
- ✧ General security guidelines sensitize designers to security issues and serve as review checklists
- ✧ System survivability reflects the ability of a system to deliver services whilst under attack or after part of the system has been damaged.

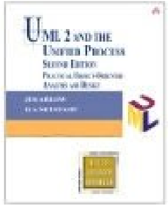


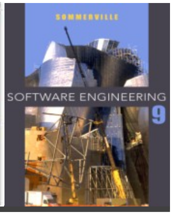


Design for Performance, Modifiability, Testability and Usability

Lecture 7/Part 4

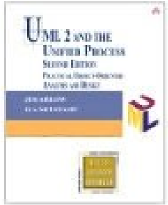
Outline

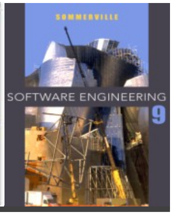




- ✧ Performance tactics
  - Resource demand
  - Resource management
  - Resource arbitration
- ✧ Modifiability tactics
  - Localize changes
  - Prevention of ripple effect
  - Defer binding time
- ✧ Testability tactics
  - Manage input/output
  - Internal monitoring
- ✧ Usability tactics

Performance tactics – Resource demand





✧ Reduce the resources required for processing an event stream.

- Increase computational efficiency.
- Reduce computational overhead.

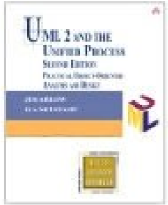
✧ Reduce the number of events processed.

- Manage event rate.
- Control frequency of sampling.

✧ Control the use of resources.

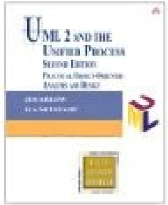
- Bound execution times.
- Bound queue sizes.

Performance tactics – Resource management



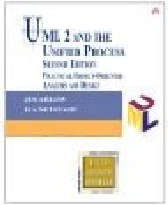
- **Introduce concurrency.** If requests can be processed in parallel, the blocked time can be reduced.
- **Maintain multiple copies** of either data or computations. The purpose of replicas is to reduce the contention that would occur if all computations took place on a central server.
- **Increase available resources.** Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency.

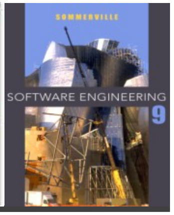
Performance tactics – Resource arbitration



- The selection of **optimal scheduling strategy** for each resource influences optimal resource usage, minimizes the number of resources used, minimizes latency, maximizes throughput, prevents starvation, and so forth.
- A scheduling policy conceptually has two parts: a **priority assignment** and **dispatching**.
- All scheduling policies assign priorities.
  - In some cases the assignment is as simple as first-in/first-out.
  - In other cases, it can be tied to the deadline of the request or its semantic importance.

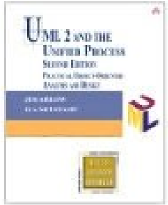
Modifiability tactics – Localize modifications

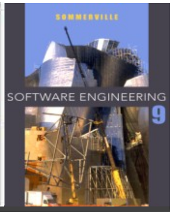




- **Maintain semantic coherence.** The goal is to ensure that all the responsibilities in a module work together without excessive reliance on other modules.
- **Generalize the module.** Making a module more general allows it to compute a broader range of functions on input.
- **Limit possible options.** Modifications, especially within a product line, may be far ranging and hence affect many modules. Restricting the possible options will reduce the effect of these modifications.

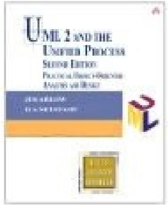
Modifiability tactics – Prevent ripple effects





- ✧ 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 private and which to make public.

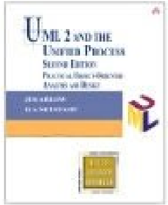
Modifiability tactics – Prevent ripple effects



- **Maintain existing interfaces.** If B depends on the name and signature of an interface of A, maintaining this interface and its syntax allows B to remain unchanged.
- **Restrict communication paths.** Restrict the modules with which a given module shares data via data production and consumption.
- **Use an intermediary.** If B has any type of dependency on A other than semantic, it is possible to insert an intermediary between B and A that manages activities associated with the dependency.

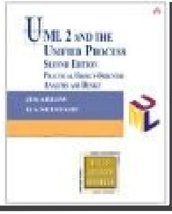
© Software Architecture in Practice
by L. Bass, P. Clements and R. Kazman

Modifiability tactics – Defer binding time



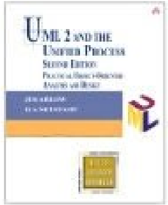
- **Runtime registration** supports plug-and-play operation at the cost of additional overhead to manage the registration. Publish/subscribe registration, for example, can be implemented at either runtime or load time.
- **Configuration files** are intended to set parameters at startup.
- **Polymorphism**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

Testability tactics – Manage input/output



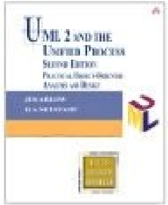
- ✧ **Record/playback.** The information crossing an interface during normal operation is saved in some repository and represents output from one component and input to another.
- ✧ **Separate interface from implementation.** Separating the interface from the implementation allows substitution of implementations for various 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 as well as independently from its normal execution.

Testability tactics – Internal monitoring



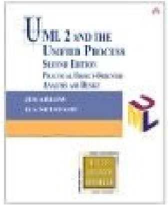
- ✧ **Built-in monitors.** The component can maintain state, performance load, capacity, security, or other information accessible through an interface.
- ✧ This interface can be a permanent interface of the component or it can be introduced temporarily via an instrumentation technique such as aspect-oriented programming or preprocessor macros.
- ✧ A common technique is to record events when monitoring states have been activated.

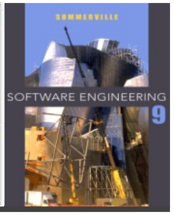
Usability tactics – Design-time tactics



- **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.

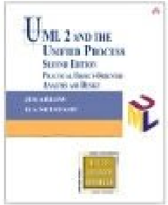
Usability tactics – Runtime tactics





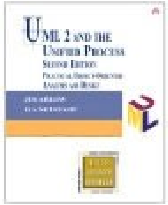
- **Maintain a model of the task.** The task model is used to determine context so the system can have some idea of what the user is attempting and provide various kinds of assistance.
  - For example, knowing that sentences usually start with capital letters would allow an application to correct a lower-case 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.
  - Portability. The main technique for achieving portable software is to isolate system dependencies, which introduces overhead into the system's execution, typically as process or procedure boundaries, and this hurts performance.
  - Reliability. Redundancy together with a voting schema delays system response.

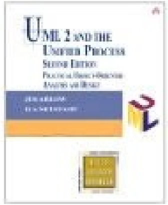
Quality conflicts



✧ 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.

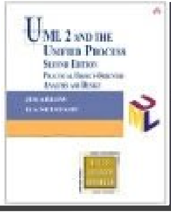
✧ The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

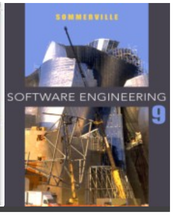


UML Class Diagram in Design

Lecture 7/Part 5

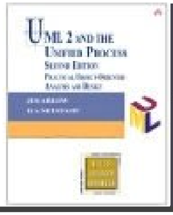
What are design classes?

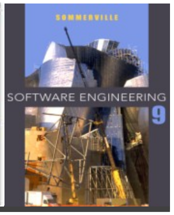


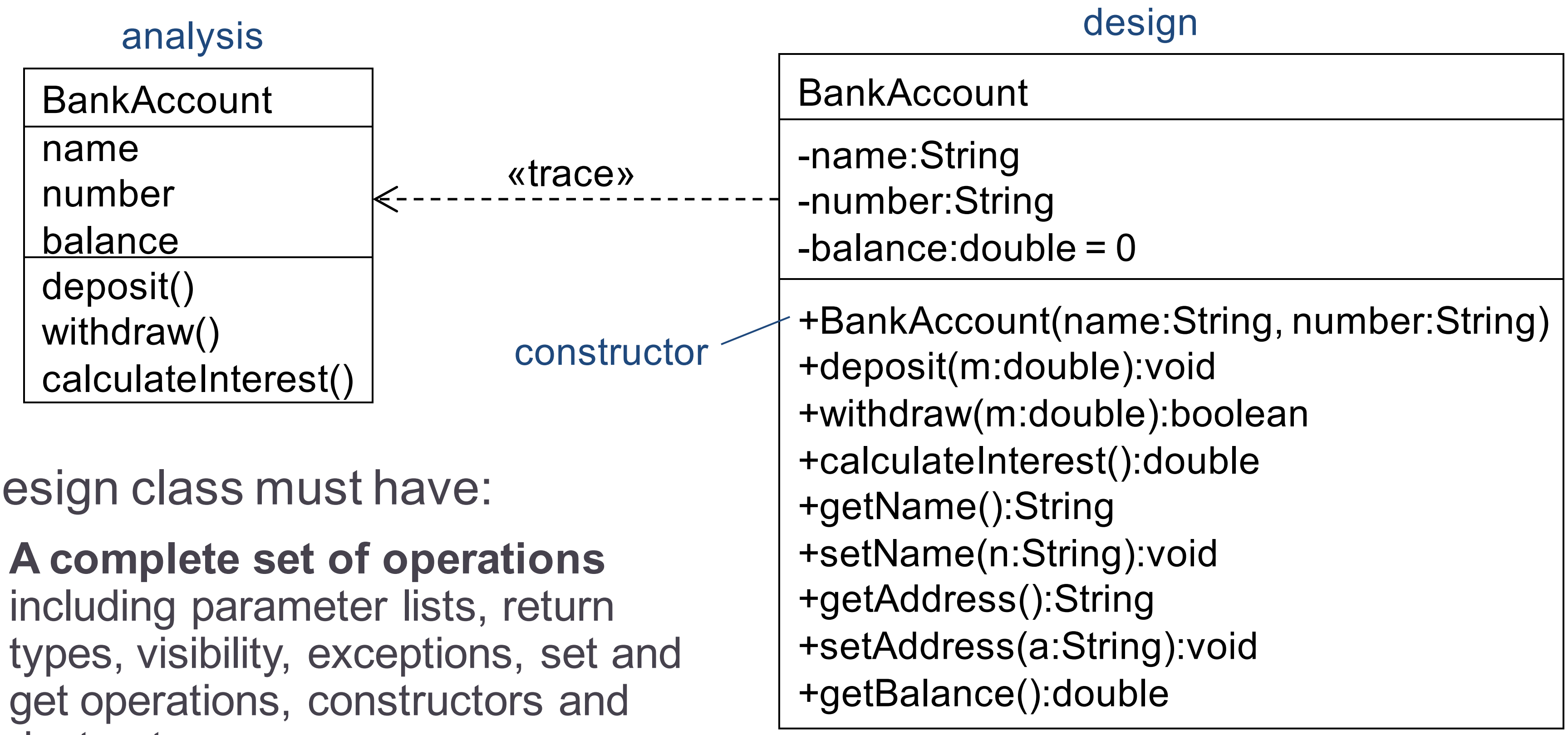


- Design classes are classes whose specifications have been completed to such a degree that they can be implemented
  - Specifies an actual piece of code
- Design classes arise from analysis classes:
  - Remember – analysis classes arise from the **problem domain only**
  - A refinement of analysis classes to include implementation details
  - All attributes are completely specified including type, visibility and default values
  - Analysis operations become fully specified operations (methods) with a return type and parameter list
- Design classes arise from the solution domain
  - Utility classes – String, Date, Time etc.
  - Middleware classes – database access, comms etc.
  - GUI classes – Applet, Button etc.

Anatomy of a design class







analysis

BankAccount
name
number
balance
deposit()
withdraw()
calculateInterest()

«trace»

constructor

design

BankAccount
-name:String
-number:String
-balance:double = 0
+BankAccount(name:String, number:String)
+deposit(m:double):void
+withdraw(m:double):boolean
+calculateInterest():double
+getName():String
+setName(n:String):void
+getAddress():String
+setAddress(a:String):void
+getBalance():double

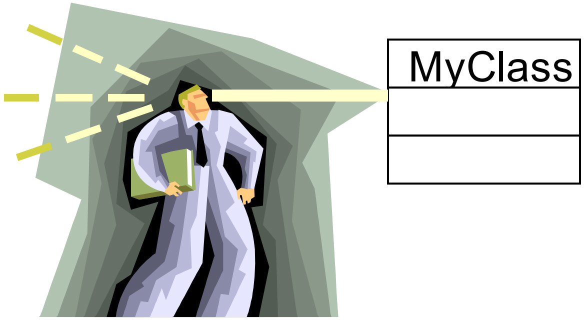
✧ A design class must have:

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

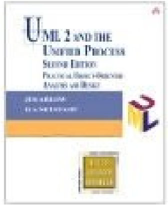
Well-formed design classes

- ✧ Design classes must have the following characteristics to be "well-formed":
  - Complete and sufficient
  - Primitive
  - High cohesion
  - Low coupling

How do the users of your classes see them?
Always look at *your* classes from *their* point of view!



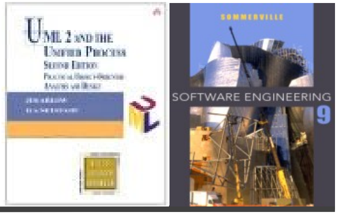
Completeness, sufficiency and primitiveness



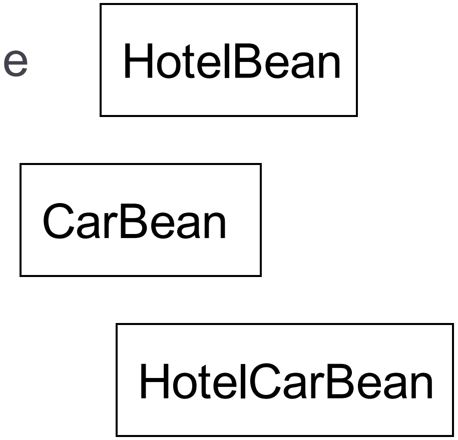
- Completeness:
  - Users of the class will make assumptions from the class name about the set of operations that it should make available
  - For example, a BankAccount class that provides a withdraw() operation will be expected to also provide a deposit() operation!
- Sufficiency:
  - A class should never surprise a user – it should contain exactly the expected set of features, no more and no less
- Primitiveness:
  - Operations should be designed to offer a single primitive, atomic service
  - A class should never offer more ways of doing the same thing:
    - This is confusing to users of the class, leads to maintenance burdens and can create consistency problems

The public members of a class define a "contract" between the class its clients

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
  - If a class needs to have many responsibilities, then some of these should be implemented by "helper" classes. The class then delegates to its helpers
- 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 form an association just to reuse a fragment of code in another class!
  - Use aggregation rather than inheritance (next slide)



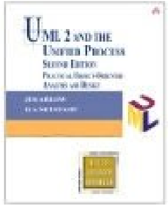
HotelBean

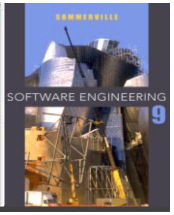
CarBean

HotelCarBean

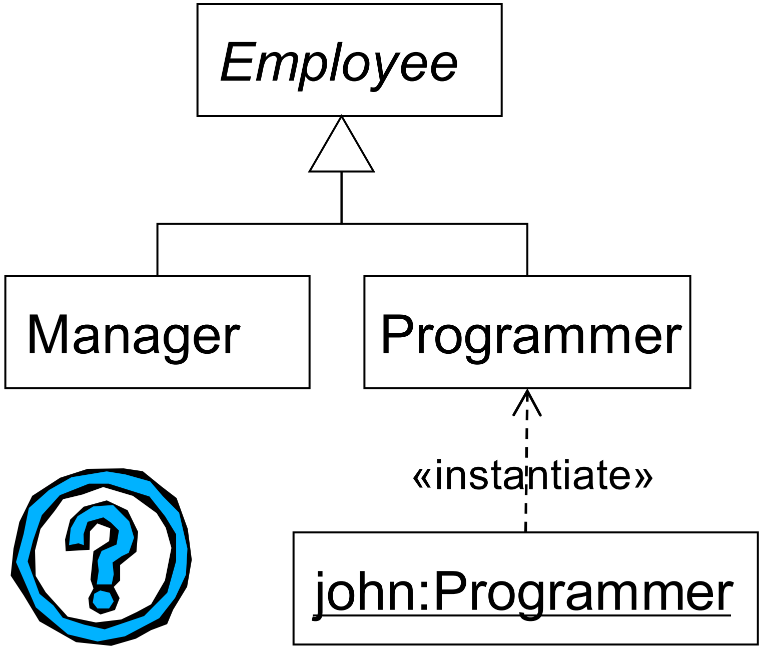
this example comes from a real system! What's wrong with it?

Aggregation vs. inheritance



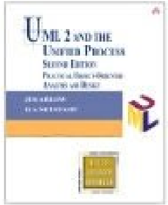


- Inheritance gives you fixed relationships between classes and objects
- You *can't* change the class of an object at runtime
- There is a fundamental semantic error here. Is an Employee *just* their job or does an Employee *have* a job?

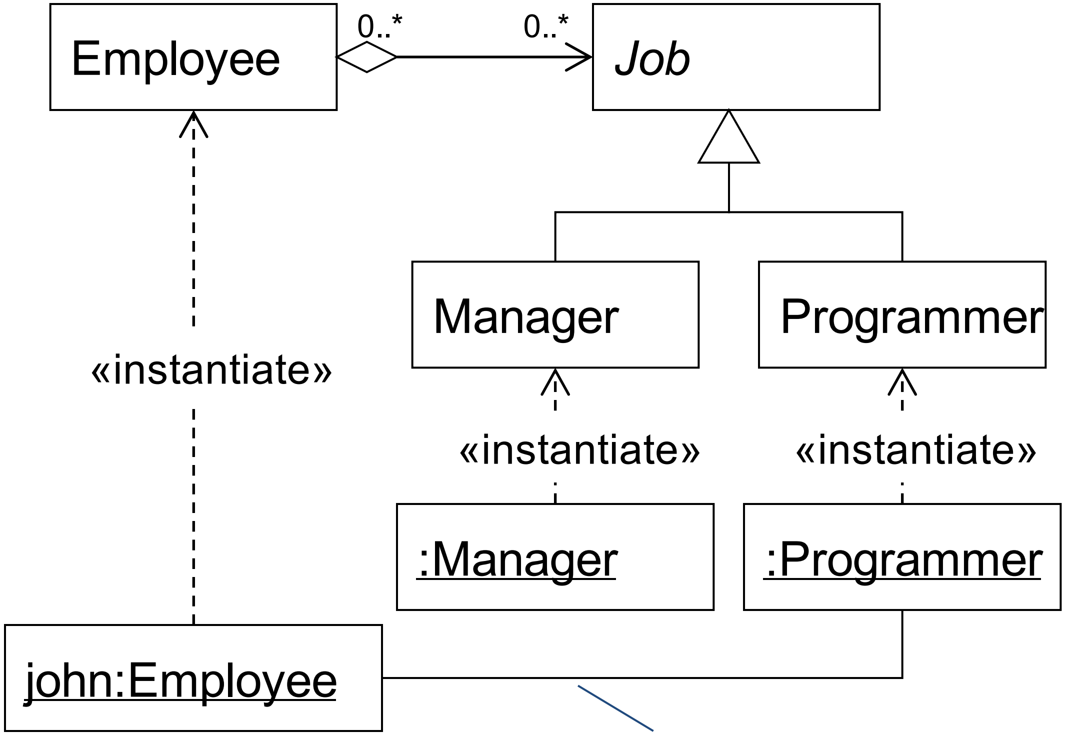


1. How can we promote john?
2. Can john have more than one job?

A better solution…



- Using aggregation we get the correct semantics:
  - An *Employee has a* Job
- With this more flexible model, Employees can have more than one Job



Employee

0..*

0..*

Job

«instantiate»

Manager

Programmer

«instantiate»

«instantiate»

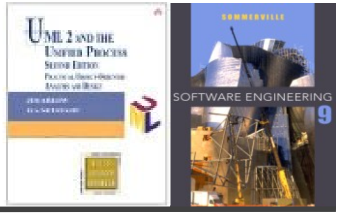
:Manager

:Programmer

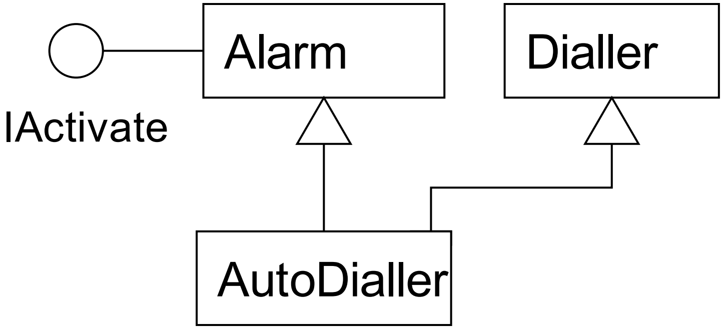
john:Employee

just change this link at runtime to promote john!

Multiple inheritance

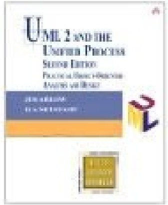


- Sometimes a class may have more than one superclass
- The "is kind of" and substitutability principles must apply for *all* of the classifications
- Multiple inheritance is sometimes the most elegant way of modelling something. However:
  - Not all languages support it (e.g. Java)
  - It can always be replaced by single inheritance and delegation



in this example the AutoDialler sounds an alarm and rings the police when triggered - it is logically both a *kind of* Alarm *and* a *kind of* Dialler

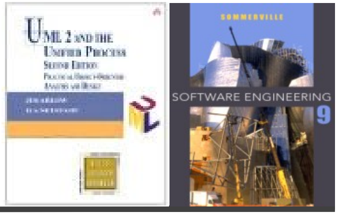
Inheritance vs. interface realization



- ✧ With inheritance we get two things:
  - Interface – the public operations of the base classes
  - Implementation – the attributes, relationships, protected and private operations of the base classes
- ✧ With interface realization we get exactly one thing:
  - Interface – a set of public operations, attributes and relationships that have no implementation

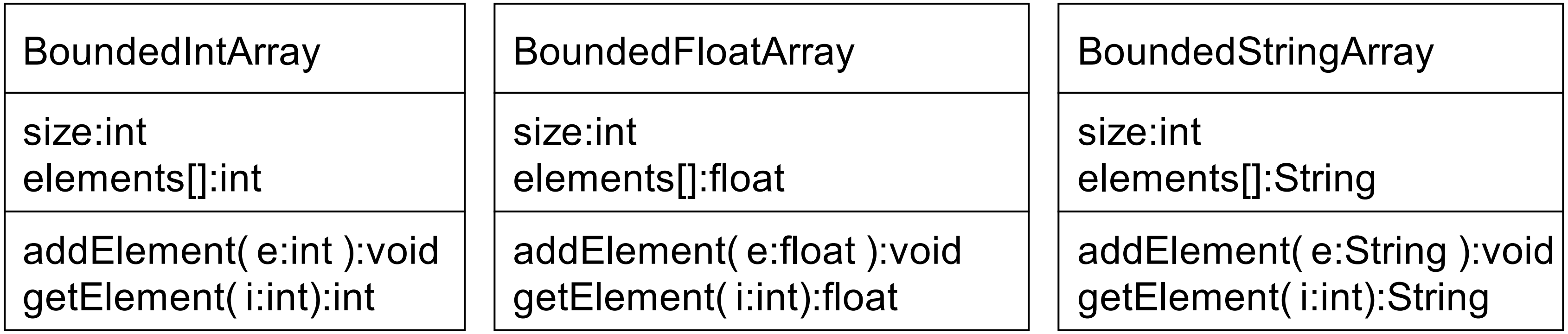
Use inheritance when we want to *inherit implementation*.
Use interface realization when we want to *define a contract*.

Templates



- ✧ Up to now, we have had to specify the types of all attributes, method returns and parameters. However, this can be a barrier to reuse
- ✧ Consider:

spot the difference!



BoundedIntArray

size:int
elements[]:int

addElement(e:int):void
getElement(i:int):int

BoundedFloatArray

size:int
elements[]:float

addElement(e:float):void
getElement(i:int):float

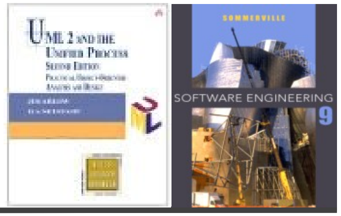
BoundedStringArray

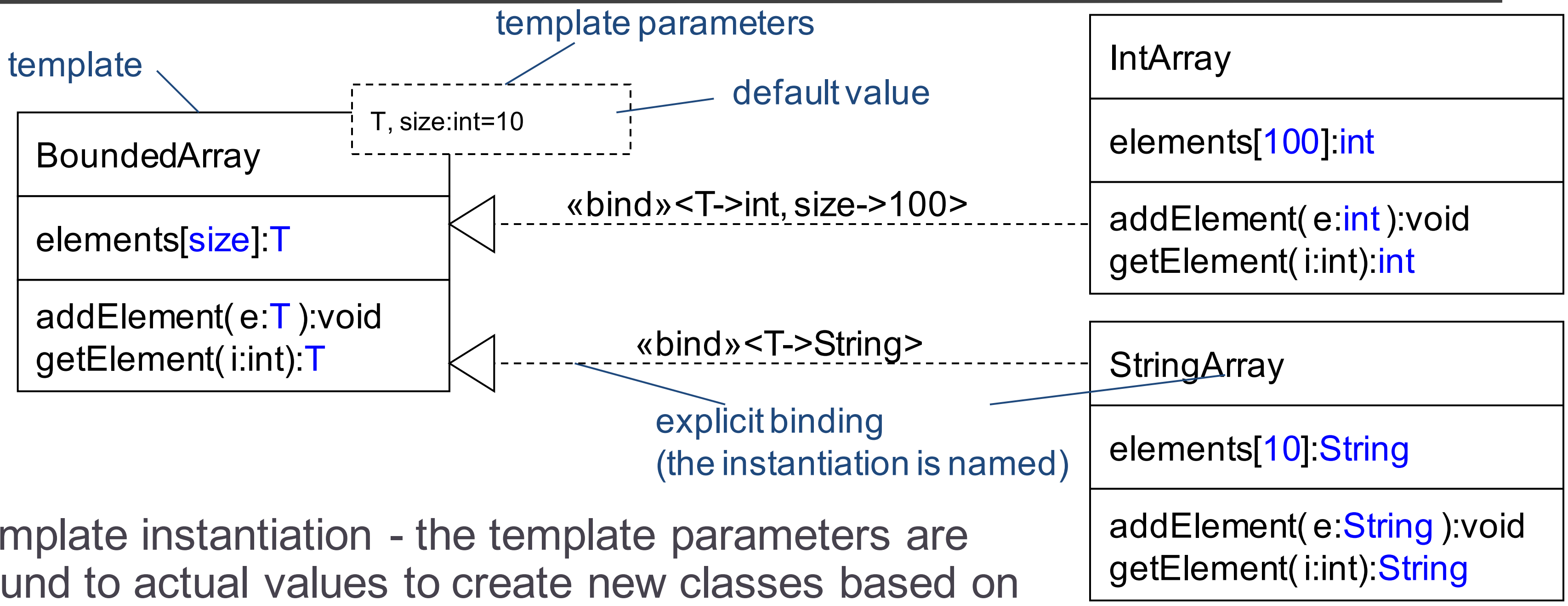
size:int
elements[]:String

addElement(e:String):void
getElement(i:int):String

etc.

Template syntax





template parameters

template

T, size:int=10

default value

BoundedArray

elements[size]:T

addElement(e:T):void
getElement(i:int):T

«bind»<T->int, size->100>

IntArray

elements[100]:int

addElement(e:int):void
getElement(i:int):int

«bind»<T->String>

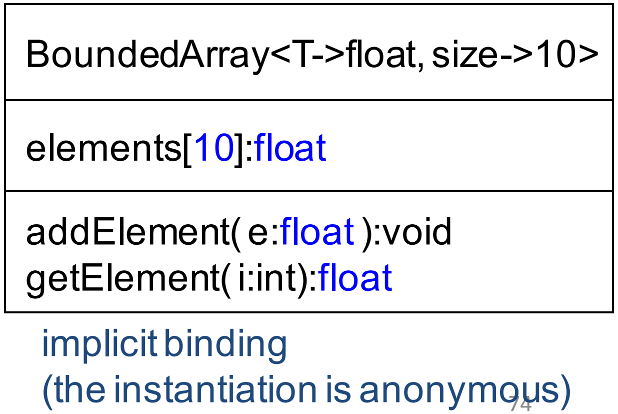
StringArray

elements[10]:String

addElement(e:String):void
getElement(i:int):String

explicit binding
(the instantiation is named)

- Template instantiation - the template parameters are bound to actual values to create new classes based on the template:
  - If the type of a parameter is not specified then the parameter defaults to being a classifier
  - Parameter names are local to the template – two templates *do not* have relationship to each other just because they use the same parameter names!
  - Explicit binding is preferred as it allows named instantiations



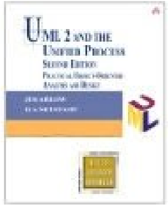
BoundedArray<T->float, size->10>

elements[10]:float

addElement(e:float):void
getElement(i:int):float

implicit binding
(the instantiation is anonymous)

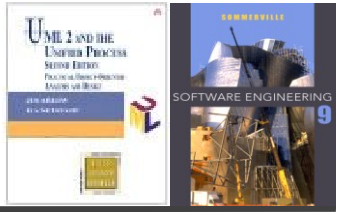
Templates & multiple inheritance

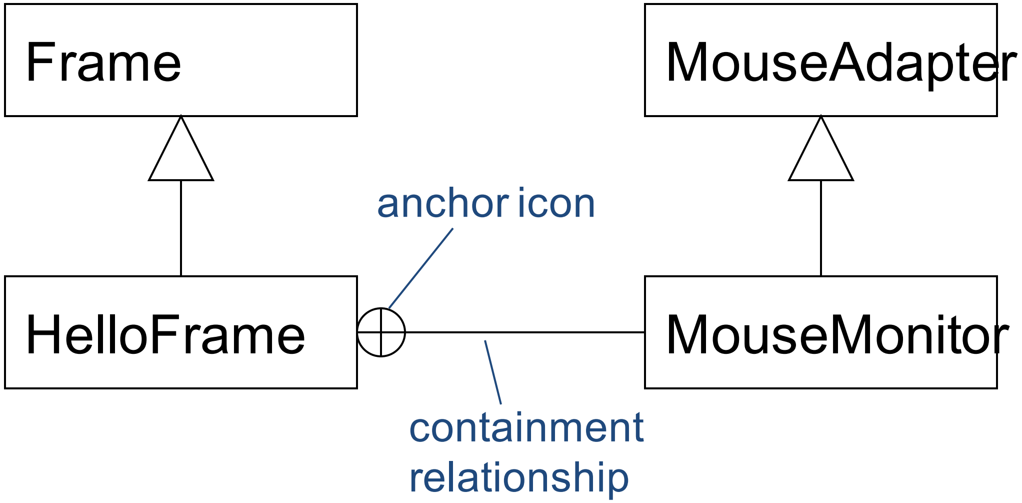


- Templates and multiple inheritance should only be used in design models where those features are available in the target language:

language	templates	multiple inheritance
C#	Yes	No
Java	Yes	No
C++	Yes	Yes
Smalltalk	No	No
Visual Basic	No	No
Python	No	Yes

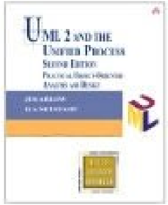
Nested classes





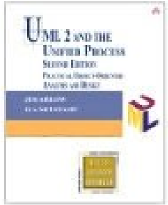
- A nested class is a class defined inside another class
  - It is encapsulated inside the namespace of its containing class
  - Nested classes tend to be design artifacts
- Nested classes are only accessible by:
  - their containing class
  - objects of that their containing class

Key points (design classes)



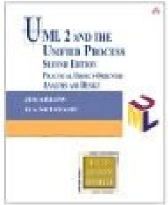
- ✧ Design classes come from:
  - A refinement of analysis classes (i.e. the business domain)
  - From the solution domain
- ✧ Design classes must be well-formed:
  - Complete and sufficient
  - Primitive operations
  - High cohesion
  - Low coupling
- ✧ Don’t overuse inheritance
  - Use inheritance for "is kind of"
  - Use aggregation for "is role played by"
  - Multiple inheritance should be used sparingly (mixins)
  - Use interfaces rather than inheritance to define contracts
- ✧ Use templates and nested classes only where the target language supports them

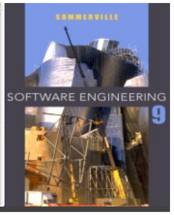
Design relationships

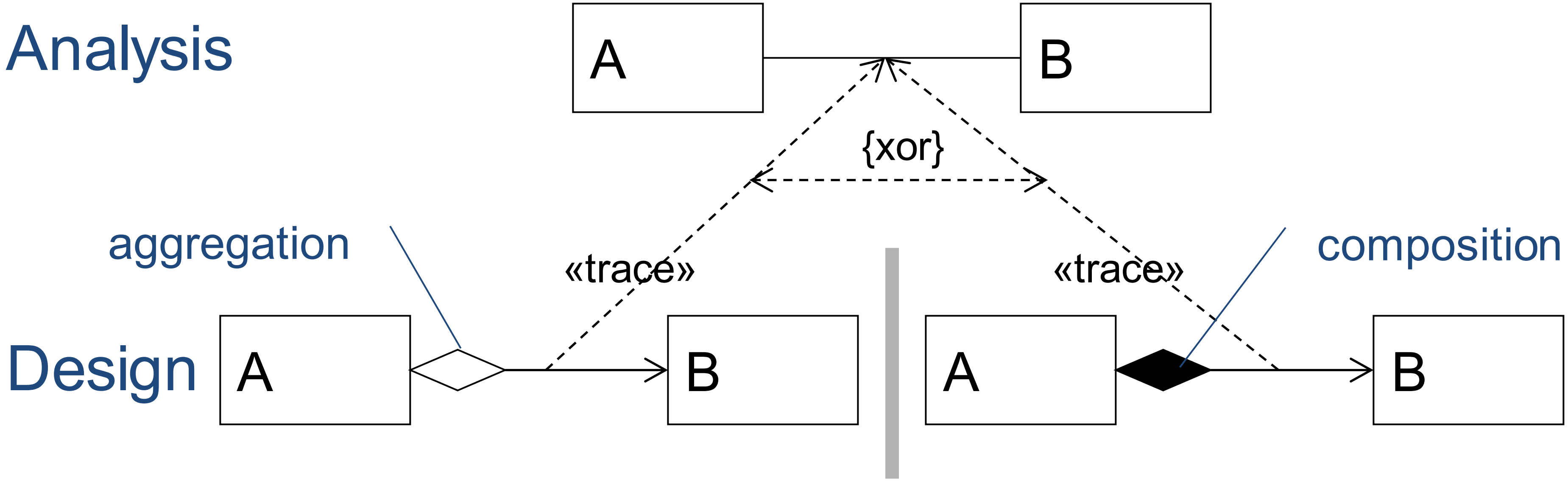


- Refining analysis associations to design associations involves several procedures:
  - refining associations to aggregation or composition relationships where appropriate
  - implementing one-to-many associations
  - implementing many-to-one associations
  - implementing many-to-many associations
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

Aggregation and composition

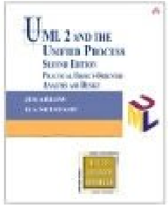


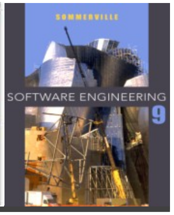




- ✧ In analysis, we often use unrefined associations. In design, these can become aggregation or composition relationships
- ✧ We must also add navigability, multiplicity and role names

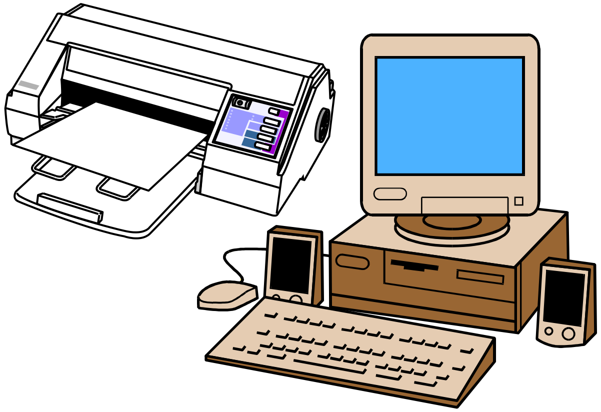
Aggregation and composition



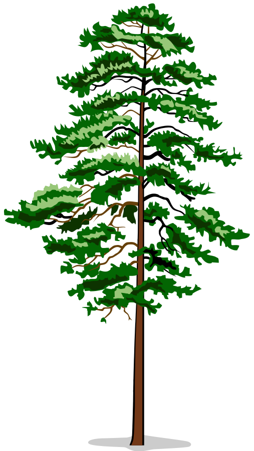


UML defines two types of association:

Aggregation



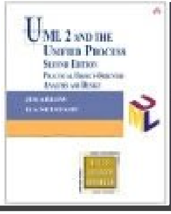
Composition

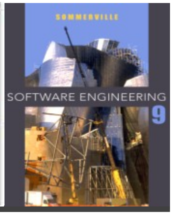


Some objects are weakly related like a computer and its peripherals

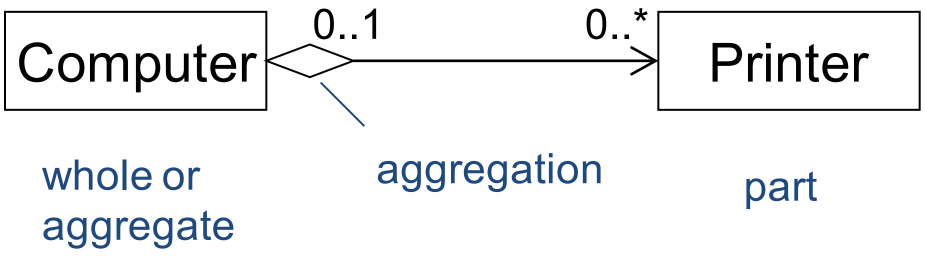
Some objects are strongly related like a tree and its leaves

Aggregation semantics





aggregation is a *whole–part* relationship



whole or aggregate

aggregation

part

A Computer may be attached to 0 or more Printers

At any one point in time a Printer is connected to 0 or 1 Computer

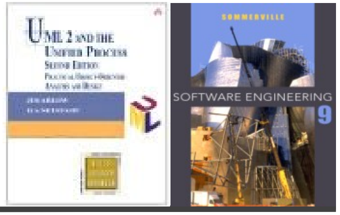
Over time, many Computers may use a given Printer

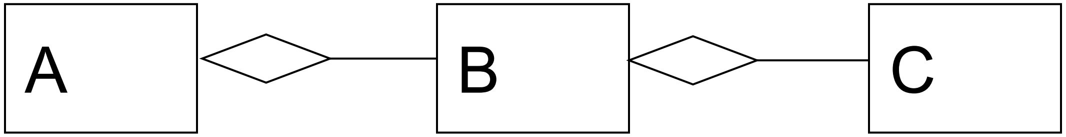
The Printer exists even if there are no Computers

The Printer is independent of the Computer

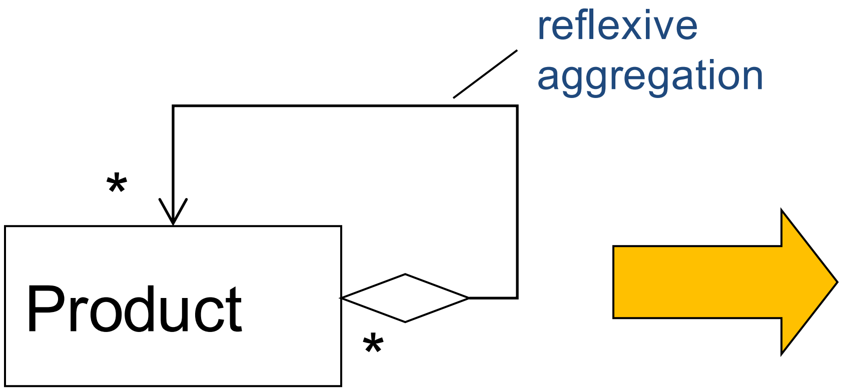
- The aggregate can sometimes exist independently of the parts, sometimes not
- The parts can exist independently of the aggregate
- The aggregate is in some way incomplete if some of the parts are missing
- It is possible to have shared ownership of the parts by several aggregates

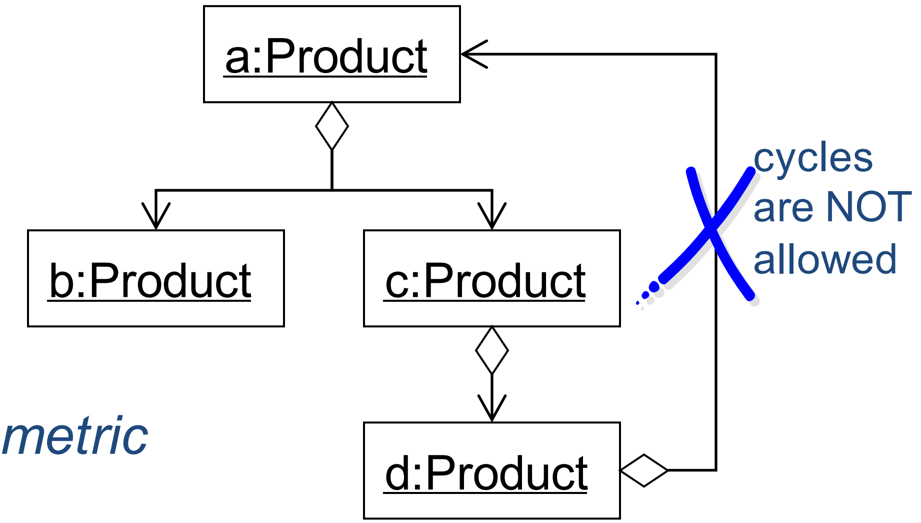
Transitive and asymmetric





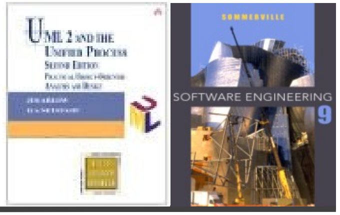
Aggregation (and composition) are *transitive*
If C is a part of B and B is a part of A, then C is a part of A

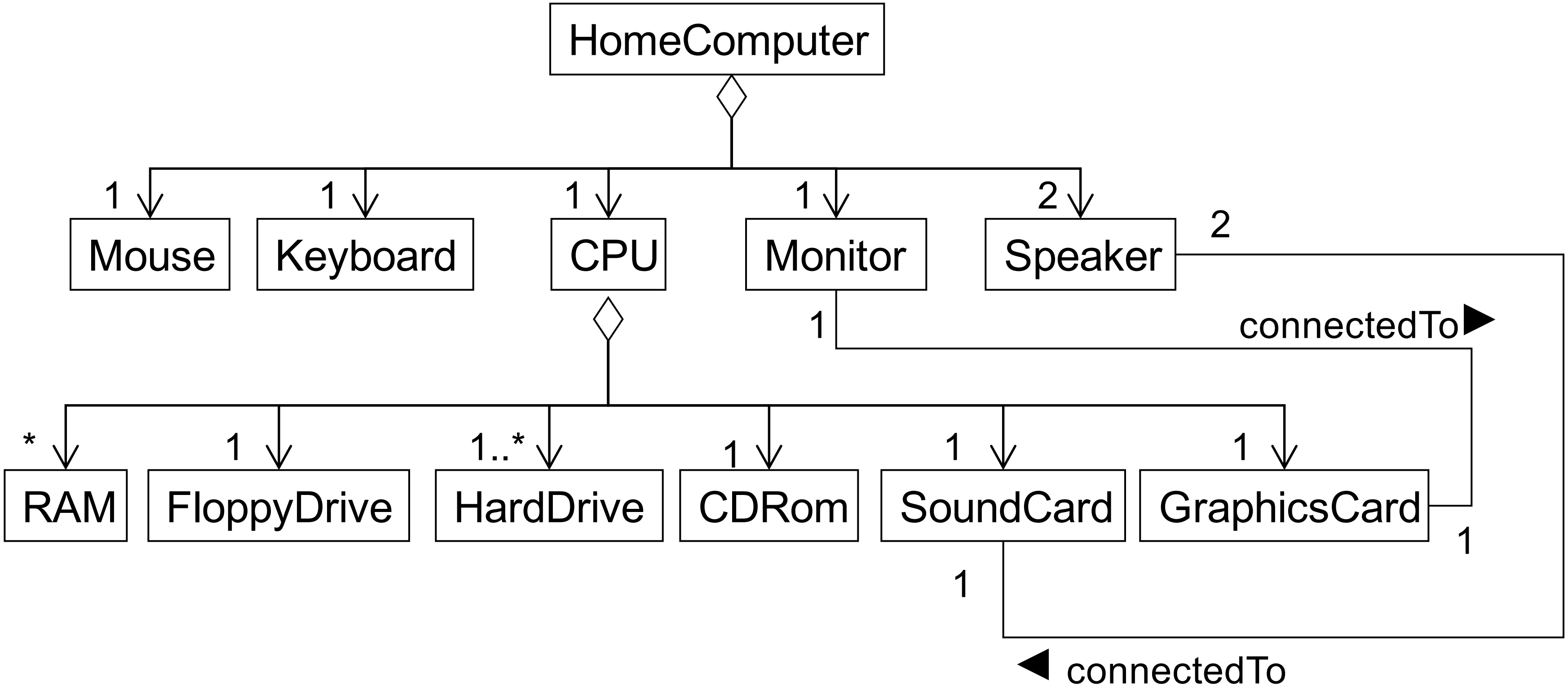




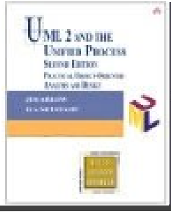
Aggregation (and composition) are *asymmetric*
An object can *never* be part of itself!

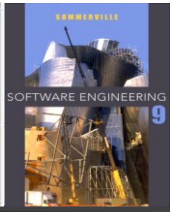
Aggregation hierarchy



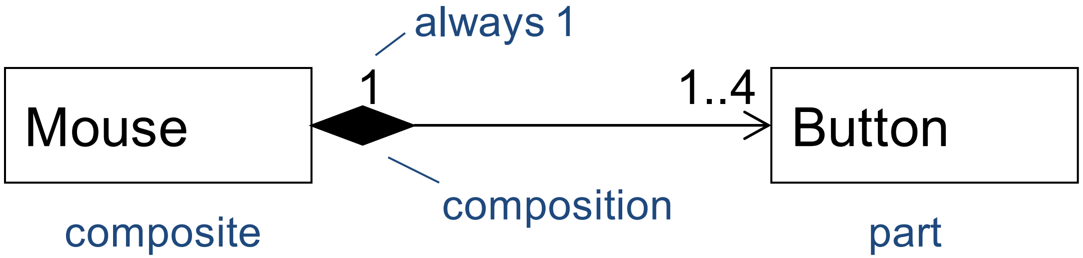


Composition semantics





composition is a strong form of aggregation



always 1

1

1..4

Mouse

Button

composition

composite

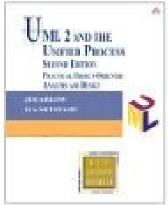
part

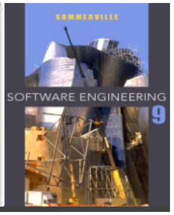
The buttons have no independent existence. If we destroy the mouse, we destroy the buttons. They are an integral part of the mouse

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
- ✧ Composition is transitive and asymmetric

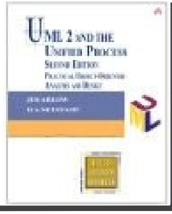
Composition and attributes

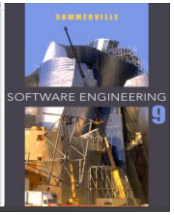


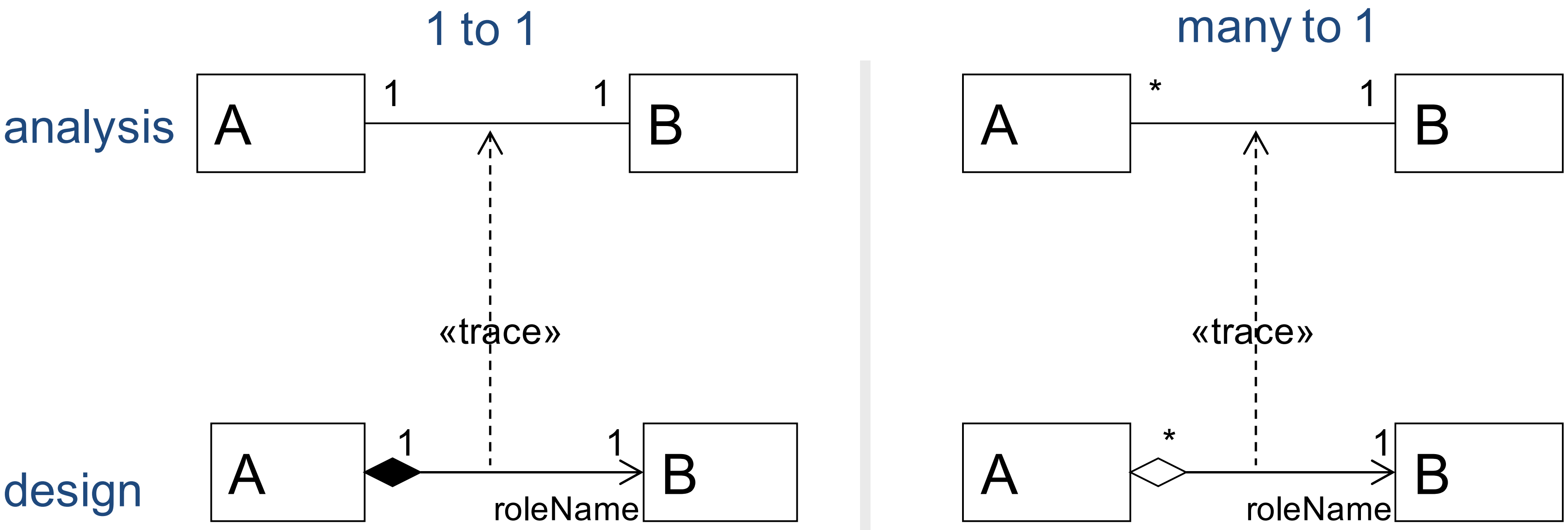


- ✧ Attributes are in effect composition relationships between a class and the classes of its attributes
- ✧ Attributes should be reserved for primitive data types (int, String, Date etc.) and **not** references to other classes

1 to 1 and many to 1 associations



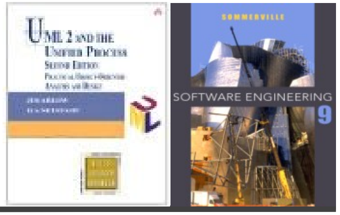




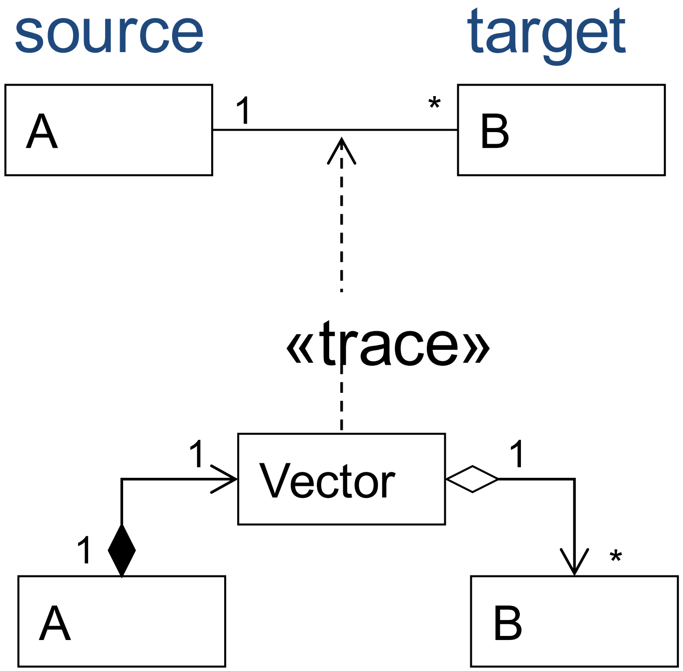
- One-to-one associations in analysis *usually* imply single ownership and *usually* refine to compositions

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

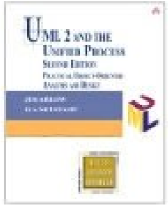
1 to many associations



- To refine 1-to-many associations we introduce a *collection (class)*
- Collection classes instances store a collection of object references to objects of the target
- A collection class always has methods for:
  - Adding an object to the collection
  - Removing an object from the collection
  - Retrieving an object reference in the collection
  - Traversing the collection
- Collection classes are typically supplied in libraries that come as part of the implementation language
- In Java we find collections in the java.util library



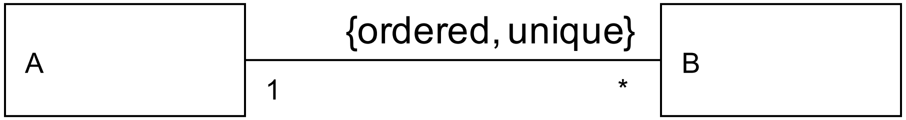
Collection semantics



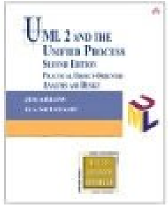
- You can specify collection semantics by using association end properties:

property	semantics
{ordered}	Elements in the collection are maintained in a strict order
{unordered}	There is no ordering of the elements in the collection
{unique}	Elements in the collection are all unique an object appears in the collection once
{nonunique}	Duplicate elements are allowed in the collection

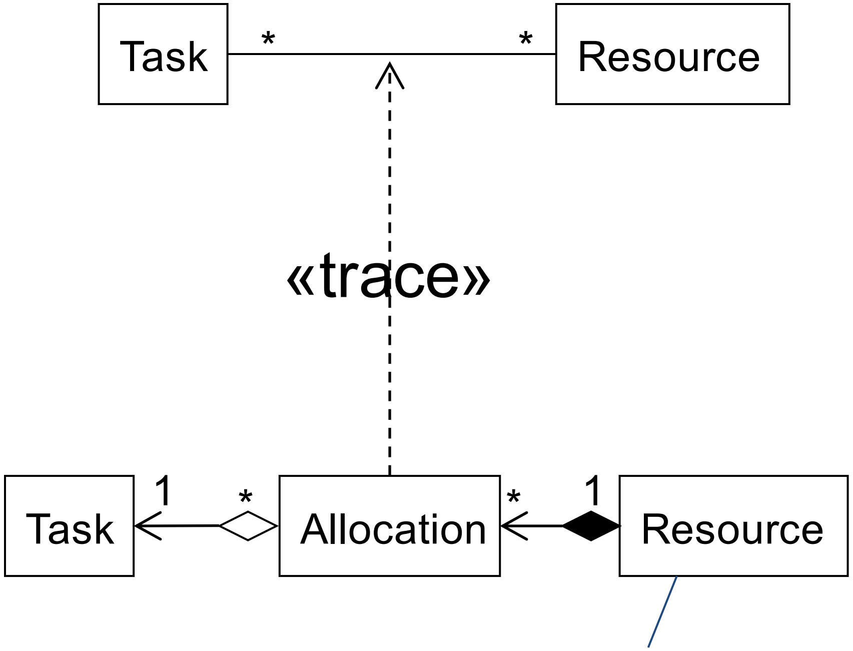
property pair	OCL collection
{unordered, nonunique}	Bag
{unordered, unique}	Set (default)
{ordered, unique}	OrderedSet
{ordered, nonunique}	Sequence



Many to many associations

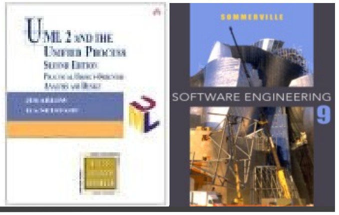


- There is no commonly used OO language that directly supports many-to-many associations
- We must reify such associations into design classes
- Again, we must decide which side of the association should have primacy and use composition, aggregation and navigability accordingly

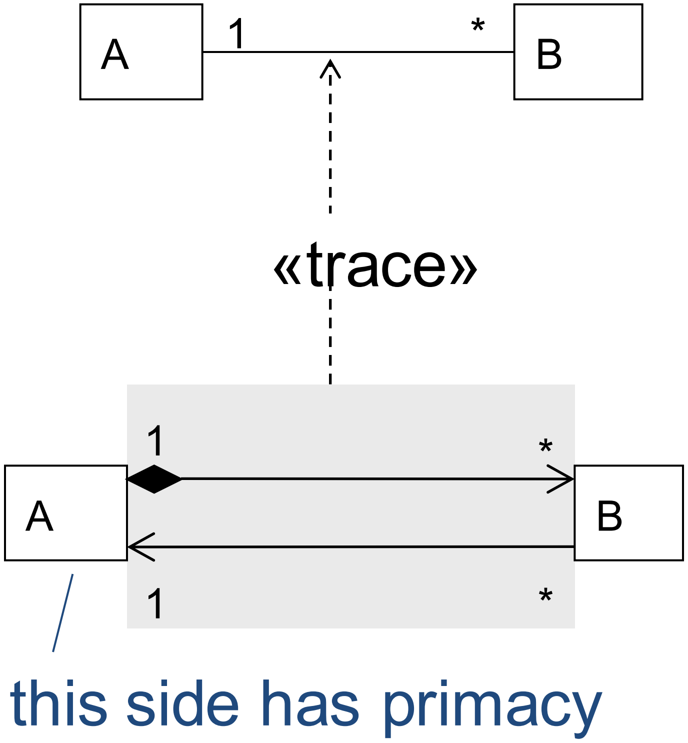


this side has primacy

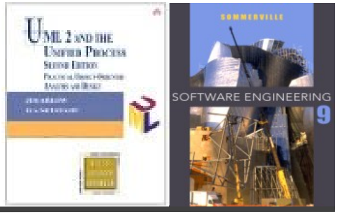
Bi-directional associations



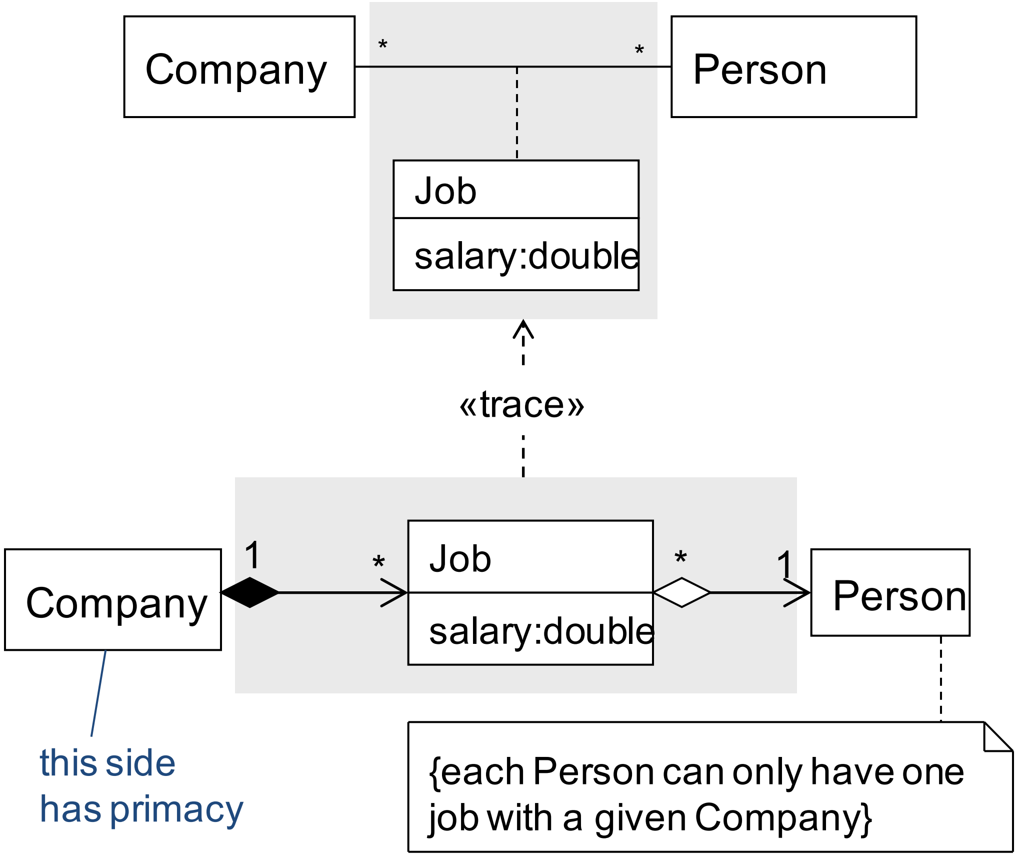
- ✧ There is no commonly used OO language that directly supports bi-directional associations
- ✧ We must resolve each bi-directional associations into two unidirectional associations
- ✧ Again, we must decide which side of the association should have primacy and use composition, aggregation and navigability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



Association classes

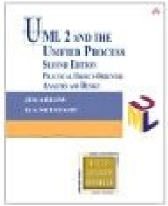


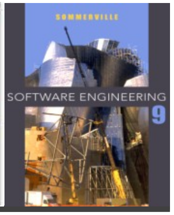
- There is no commonly used OO language that directly supports association classes
- Refine all association classes into a design class
- Decide which side of the association has primacy and use composition, aggregation and navigability accordingly



{each Person can only have one job with a given Company}

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
    - The whole is incomplete in some way without the parts
  - Composition
    - A strong form of aggregation
    - Parts are entirely dependent on the whole
    - Parts may not be shared
    - The whole is incomplete without the parts
- One-to-many, many-to-many, bi-directional associations and association classes are refined in design