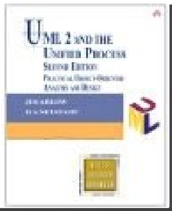
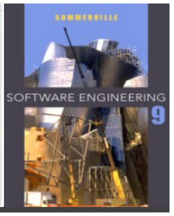
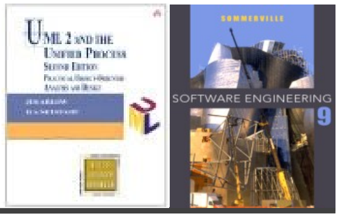



High-Level Design

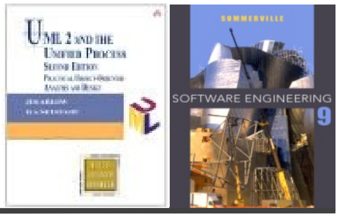
Lecture 7

Purpose of high-level design



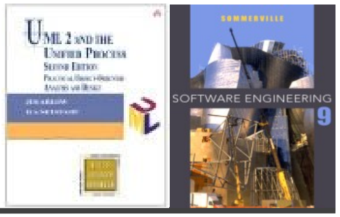
- Refine how the system's functions are to be implemented and how non-functional requirements are to be ensured
- Decide on strategic design issues such as concurrency, redundancy, persistence, distribution etc. to end with a design satisfying both functional and non-functional requirements
- Create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

Design best practices



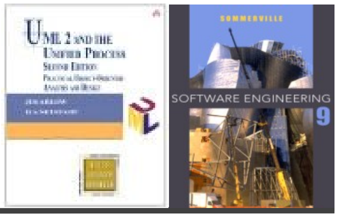
- ✧ A system design consists of a collection of decisions that help to control different attributes of software quality.
  - The design aims to ensure achievement of system functionality, but whenever there are different ways to achieve the functionality, the impact of each design decision on software 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 dependability
- ✧ Design for security
- ✧ Design for performance, modifiability and usability

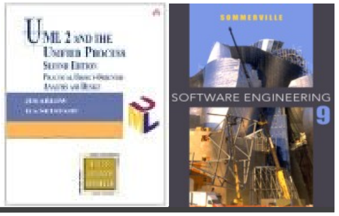
- ✧ UML State diagram



Design for Dependability

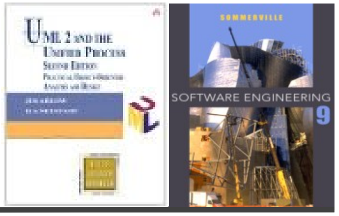
Lecture 7/Part 1

Software dependability



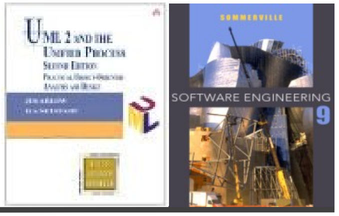
- ✧ In general, software customers expect all software to be dependable. However, for non-critical applications, they may be willing to accept some system failures.
- ✧ Some applications (critical systems) have very high dependability requirements and special software engineering techniques may be used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

Dependability achievement



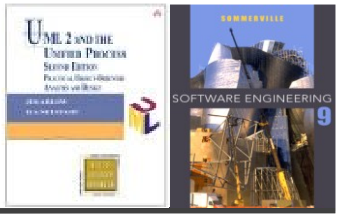
- Fault avoidance
  - The development process is organised so that faults in the system are detected and repaired before delivery to the customer.
  - Verification and validation techniques are used to discover and remove faults in a system before it is deployed.
- Fault detection
  - Run-time techniques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

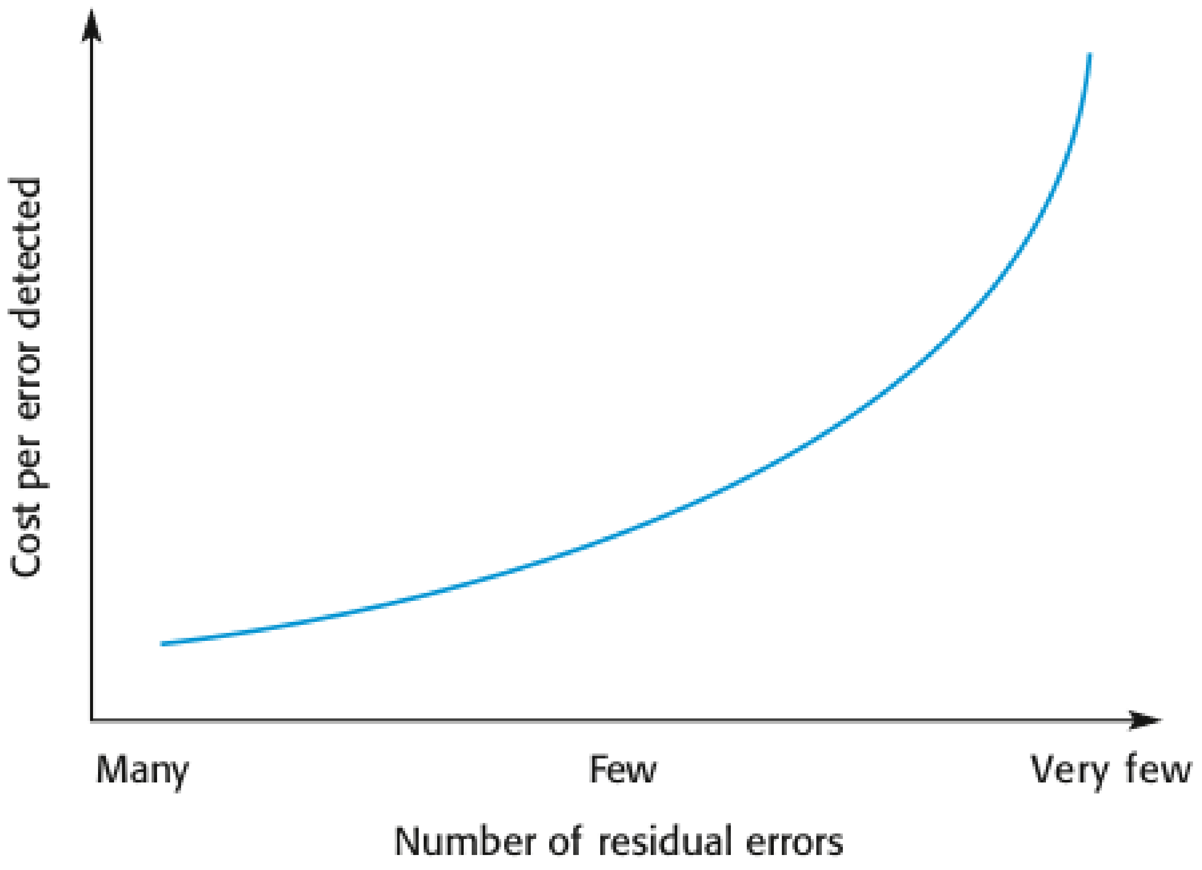
Dependable processes for fault avoidance



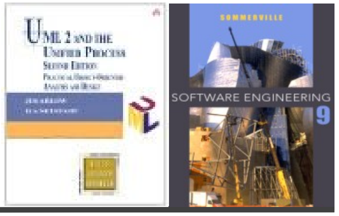
- ✧ To ensure a minimal number of software faults, it is important to have a well-defined, repeatable software process.
- ✧ The process should not depend entirely on individual skills; rather can be enacted by different people.
- ✧ Regulators use information about the process to check if good software 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

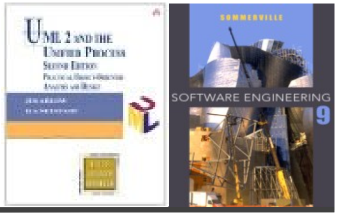




Dynamic fault detection tactics



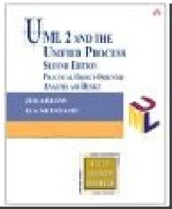
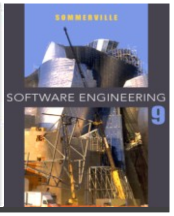
- **Ping/echo.** One component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny.
- **Heartbeat (dead man timer).** In this case one component emits a heartbeat message periodically and another component listens for it. If the heartbeat fails,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



Fault tolerance

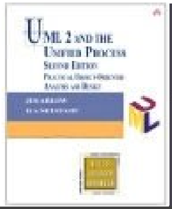
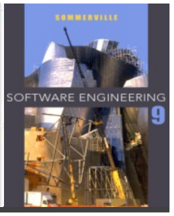
- **In critical situations**, software systems must be fault tolerant.
  - Fault tolerance is required where there are high availability requirements or where system failure costs are very high.
- **Fault tolerance** means that the system can continue in operation in spite of software failure.
  - Even 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

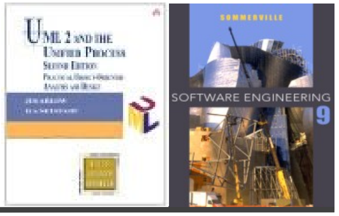
- Redundancy
  - Keep more than 1 version of a critical component available so that if one fails then a backup is available.
  - E.g. switch to backup servers automatically if failure occurs.
- Diversity
  - Provide the same functionality in different ways so that they will not fail in the same way.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

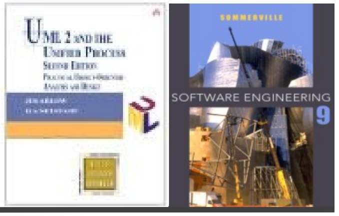



- **Voting.** Processes running on redundant processors each take equivalent input and compute a simple output value that is sent to a voter to choose non-deviant result.
- **Active redundancy** (hot restart). All redundant components respond to events in parallel. Consequently, they are all in the same 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)



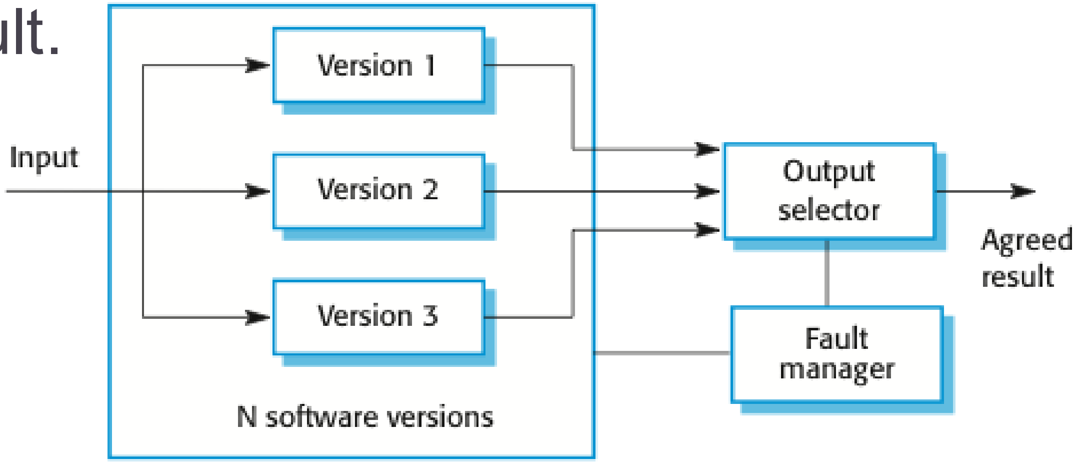
- **Spare.** A standby spare computing platform is configured to replace many different failed components. It must be rebooted to the appropriate software configuration and have its state initialized when a failure occurs.
- **Shadow operation.** A previously failed component may be run in "shadow mode" 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.



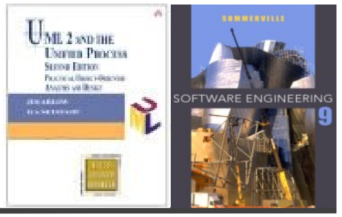
N-version programming pattern

- ✧ 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.

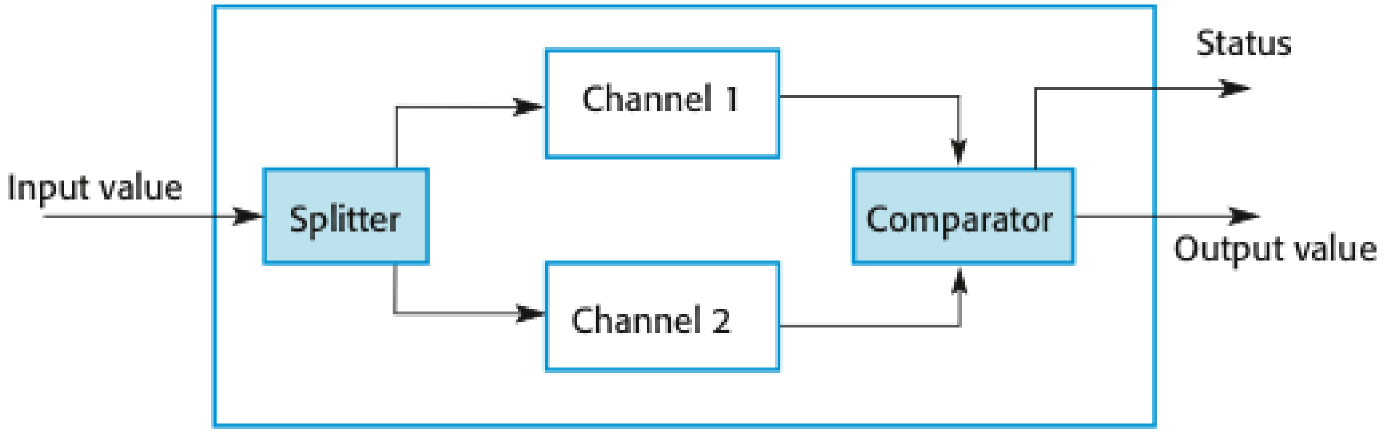
Which of the tactics are involved here?



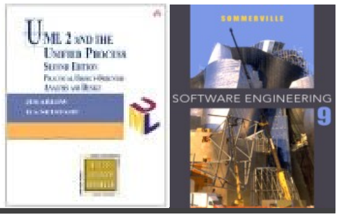
Self-monitoring architectures



- ✧ 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



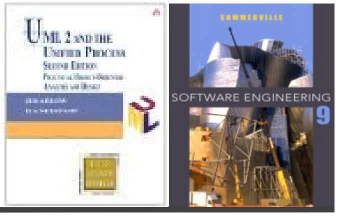
✧ A specialized system that is associated with some other control system, which can take emergency action if a failure occurs.

- System to stop a train if it passes a red light
- System to shut down a reactor if temperature/pressure are too high

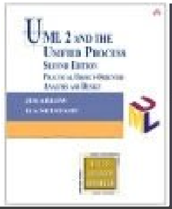
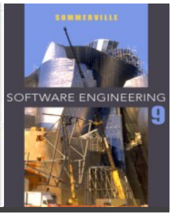
✧ Protection systems are redundant because they include monitoring and control capabilities that replicate those in the control software.

✧ Protection systems should be diverse and use different technology from the control software.

Key points



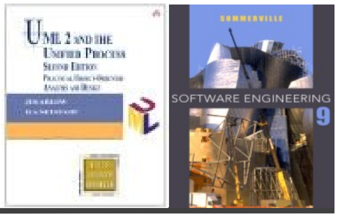
- Dependability in a program can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment, and by including fault tolerance facilities.
- The use of redundancy and diversity in hardware, software processes and software systems is essential for the 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

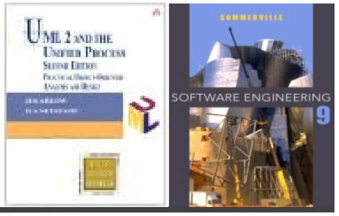
Lecture 7/Part 2

Design for security



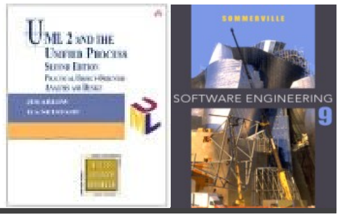
- ✧ Two fundamental issues have to be considered when designing an architecture for security.
  - Protection
    - How should the system be organised so that critical assets can be protected against external attack?
  - Distribution
    - How should system assets be distributed so that the effects of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

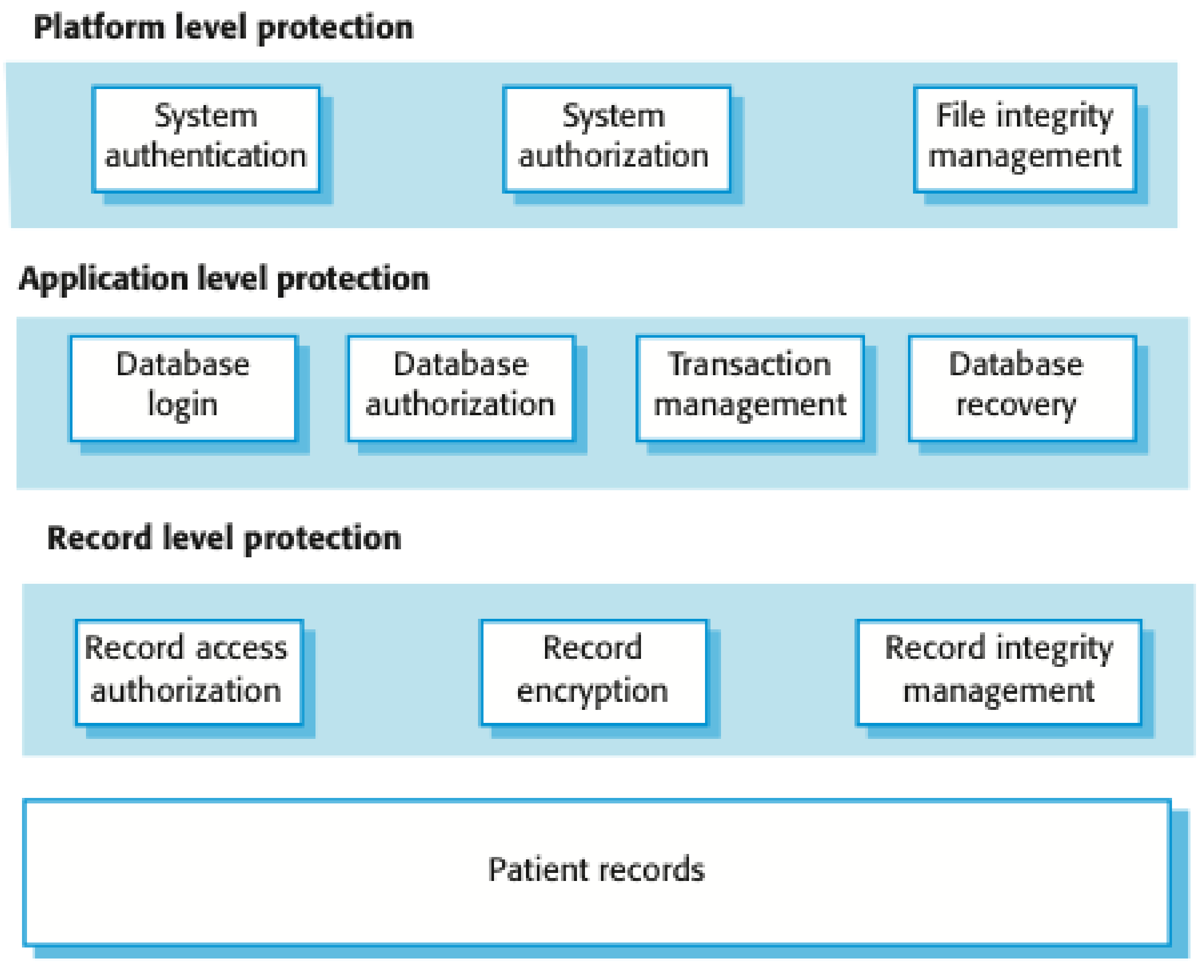
Protection



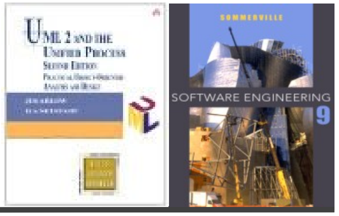
- Platform-level protection
  - Top-level controls on the platform on which a system runs.
- Application-level protection
  - Specific protection mechanisms built into the application itself e.g. additional password protection.
- Record-level protection
  - Protection that is invoked when access to specific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

A layered protection architecture

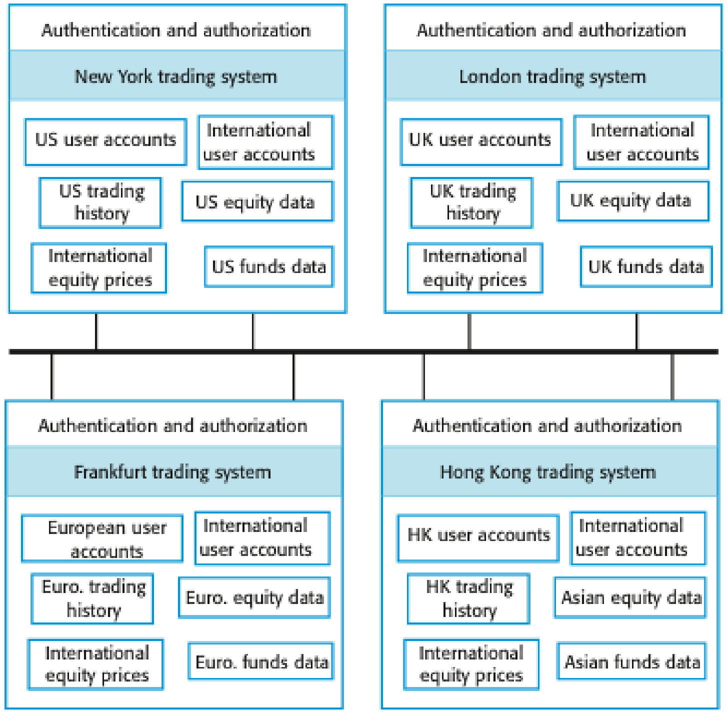




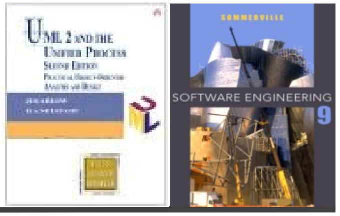
Distribution



- ✧ 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

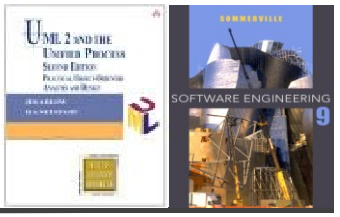


Security tactics



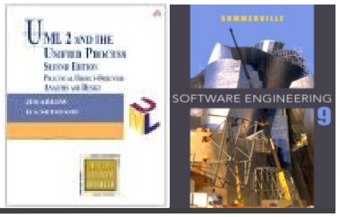
- ✧ Security tactics encapsulate good practice in secure systems design
- ✧ Security tactics serve two purposes:
  - They raise awareness of security issues in a software engineering team. Security is considered when design decisions are made.
  - They can be used as the basis of a review checklist that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

Tactics for secure systems engineering



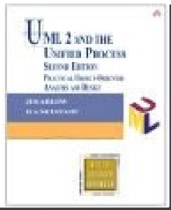
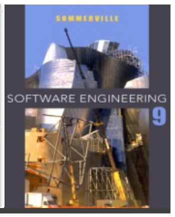
Security tactics
Base security decisions on an explicit security policy
Avoid a single point of failure
Fail securely
Balance security and usability
Log user actions
Use redundancy and diversity to reduce risk
Compartmentalize your assets
Design for recoverability
Design for deployment
Validate all inputs

Design guidelines 1-3



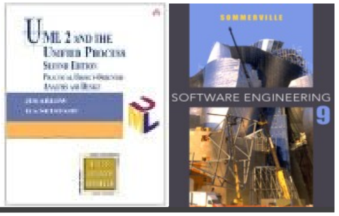
- ✧ Base decisions on an explicit security policy
  - Define a security policy for the organization that sets out the fundamental security requirements that should apply to all organizational systems.
- ✧ Avoid a single point of failure
  - Ensure that a security failure can only result when there is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

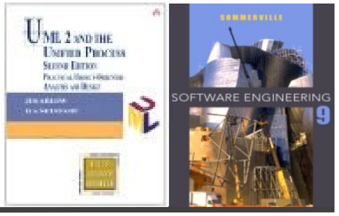
- ✧ Balance security and usability
  - Try to avoid security procedures that make the system difficult to use. Sometimes you have to accept weaker security to make the system more usable.
- ✧ Log user actions
  - Maintain a log of user actions that can be analyzed to discover who did what. If users know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

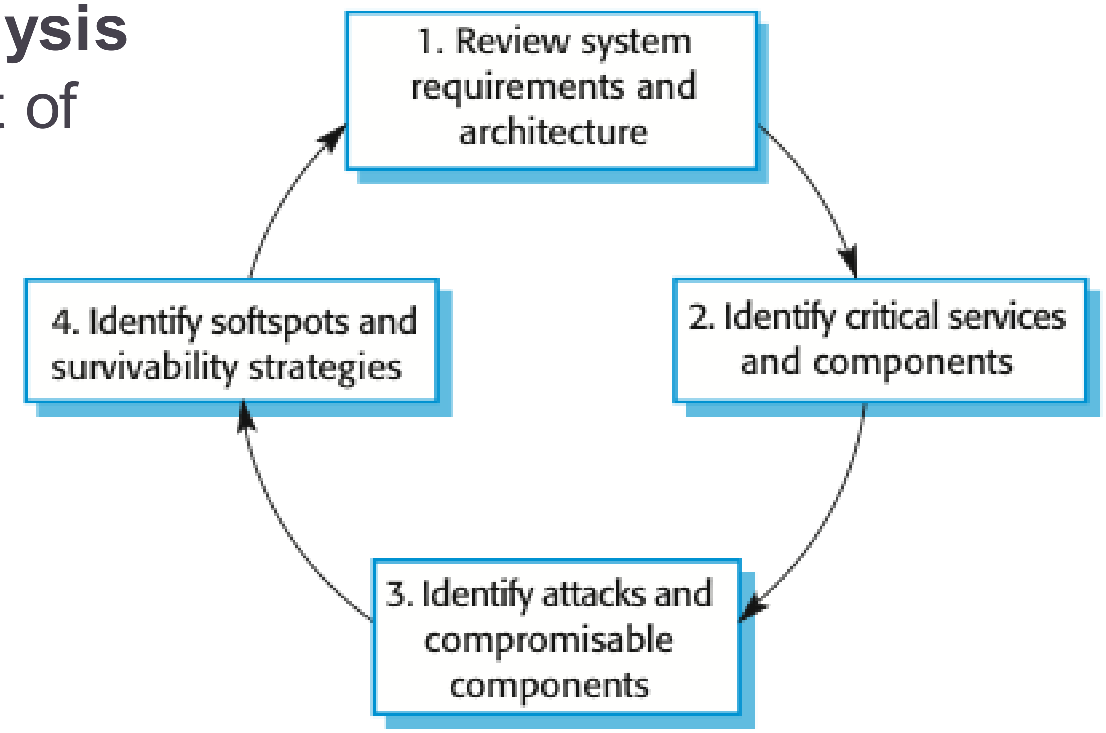


- ✧ Compartmentalize your assets
  - Organize the system so that assets are in separate areas and users only have access to the information that they need rather than all system information.
- ✧ Design for recoverability
  - Design the system to simplify recoverability after a successful attack.
- ✧ 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.

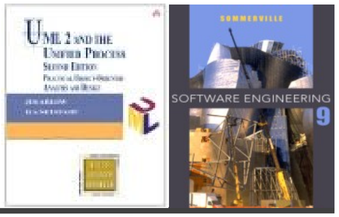
System survivability



- Survivability 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

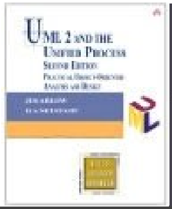
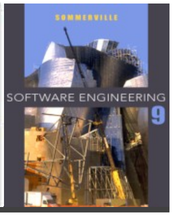

1. Review system requirements and architecture
2. Identify critical services and components
3. Identify attacks and compromisable components
4. Identify softspots and survivability strategies

Survivability strategies

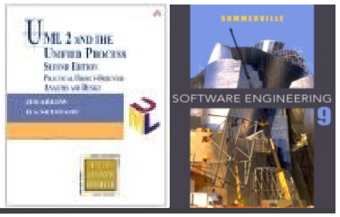


- ✧ Resistance
  - Avoiding problems by building capabilities into the system to resist attacks
- ✧ Recognition
  - Detecting problems by building capabilities into the system to detect attacks and failures and assess the resultant damage
- ✧ Recovery
  - Tolerating problems by building capabilities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

Key points

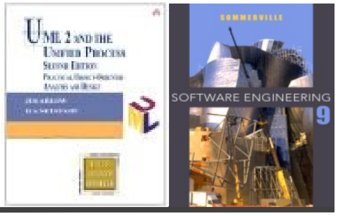
- ✧ Design for security involves architectural design, following good design practice and minimising the introduction of system vulnerabilities
- ✧ General security guidelines sensitize designers to security issues and serve as review checklists
- ✧ System survivability reflects the ability of a system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



Design for Performance, Modifiability and Usability

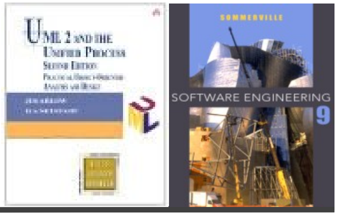
Lecture 7/Part 3

Performance tactics – Resource management



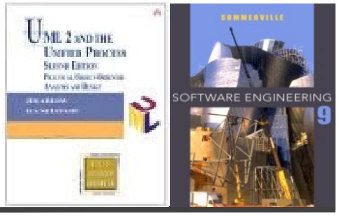
- ✧ **Introduce concurrency.** If requests can be processed in parallel, the blocked time can be reduced.
- ✧ **Maintain multiple copies** of either data or computations. The purpose of replicas is to reduce the contention that would occur if all computations took place on a central server.
- ✧ **Increase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

Performance tactics – Resource arbitration



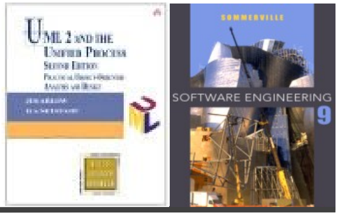
- ✧ The selection of **optimal scheduling strategy** for each resource influences optimal resource usage, minimizes the number of resources used, minimizes latency, maximizes throughput, prevents starvation, and so forth.
- ✧ A scheduling policy conceptually has two parts: a **priority assignment** and **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



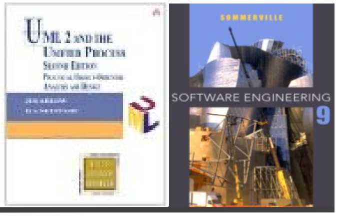
- ✧ **Maintain semantic coherence.** The goal is to ensure that all the responsibilities in a module work together without excessive reliance on other modules.
- ✧ **Generalize the module.** Making a module more general allows it to compute a broader range of functions on input.
- ✧ **Limit possible 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



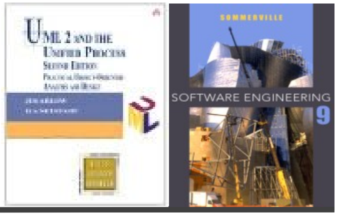
- **A ripple effect** from a modification is the necessity of making changes to modules not directly affected by it.
  - For instance, if module A is changed to accomplish a particular modification, then module B is changed only because of the change to module A. B has to be modified because it depends, 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



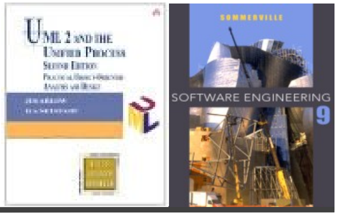
- **Maintain existing interfaces.** If B depends on the name and signature of an interface of A, maintaining this interface and its syntax allows B to remain unchanged.
- **Restrict communication paths.** Restrict the modules with which a given module shares data via data production and consumption.
- **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.

Modifiability tactics – Defer binding time



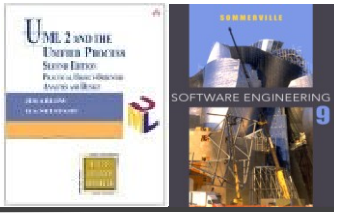
- **Runtime registration** supports plug-and-play operation at the cost of additional overhead to manage the registration. Publish/subscribe registration, for example, can be implemented at either runtime or load time.
- **Configuration files** are intended to set parameters at startup.
- **Polymorphism**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

Usability tactics – Design-time tactics



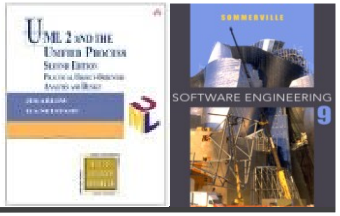
- **Separate the user interface from the rest of the application.** Localizing expected changes is the rationale for semantic coherence.
- Since the user interface is expected to change frequently both during the development and after deployment, maintaining the user interface code separately will localize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

Usability tactics – Runtime tactics



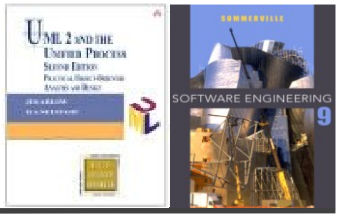
- **Maintain a model of the task.** The task model is used to determine context so the system can have some idea of what the user is attempting and provide various kinds of assistance.
  - For example, knowing that sentences usually start with capital letters would allow an application to correct a lower-case 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.

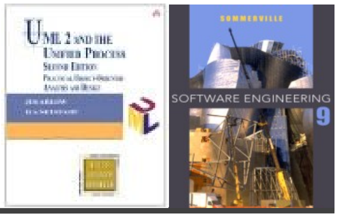
Quality conflicts



✧ 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.

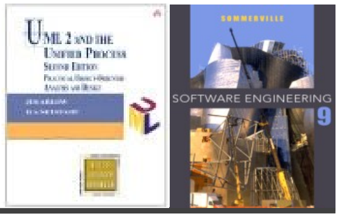
✧ The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.



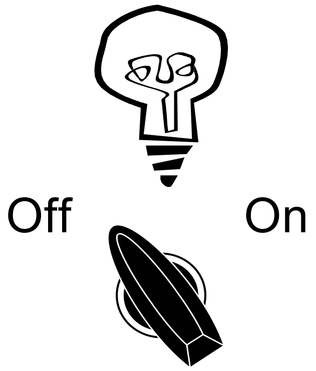
UML State Diagram

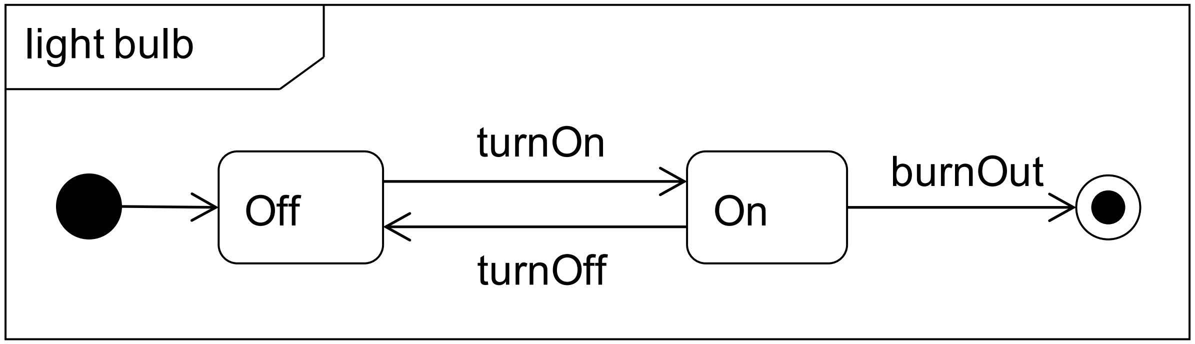
Lecture 7/Part 4

State machines

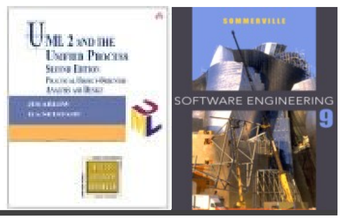


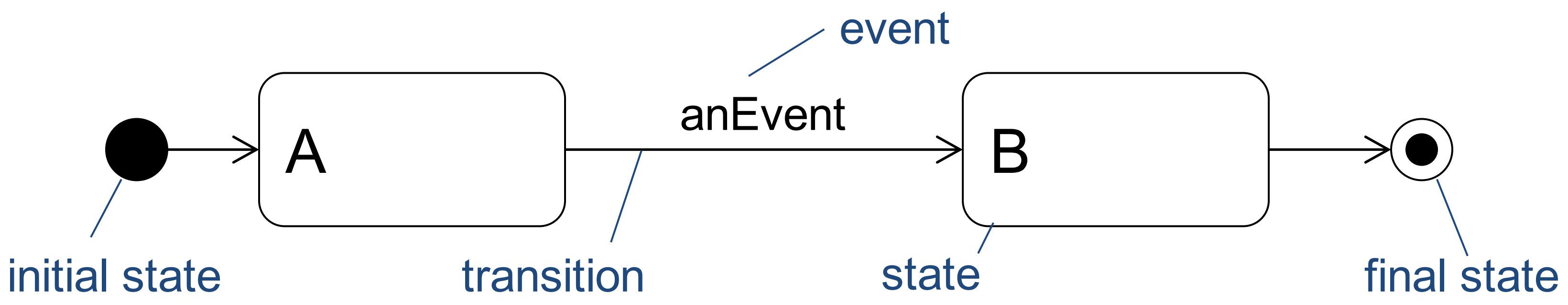
- ✧ Models life stages of a **single** model element – e.g. object, use case, module
- ✧ Every state machine exists in the context of a particular model element that:
  - Has a clear life history modelled as a progression of *states, transitions* and *events*
  - Responds to events dispatched from outside of the element
- ✧ There are two types of state machines:
  - **Behavioural state machines** - define the behaviour of a model element
  - **Protocol state machines** - model the protocol of a classifier
    - E.g. call conditions and call ordering of an interface that itself has no behaviour





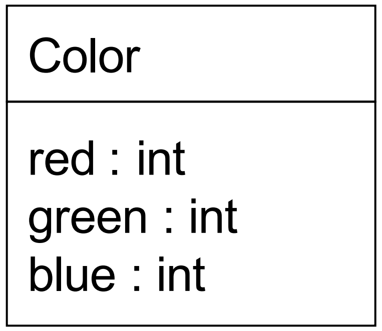
Basic state machine syntax



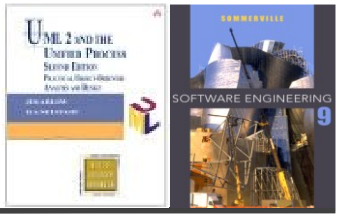


- ✧ State = a situation or condition during the life of an object
  - Determined at any point in time by the **values of its attributes**, the relationships to other objects, or the **activities** it is performing.
- ✧ Every state machine should have one initial state which indicates the first state of the sequence
- ✧ Unless the states cycle endlessly, state machines should have a final state which terminates its lifecycle

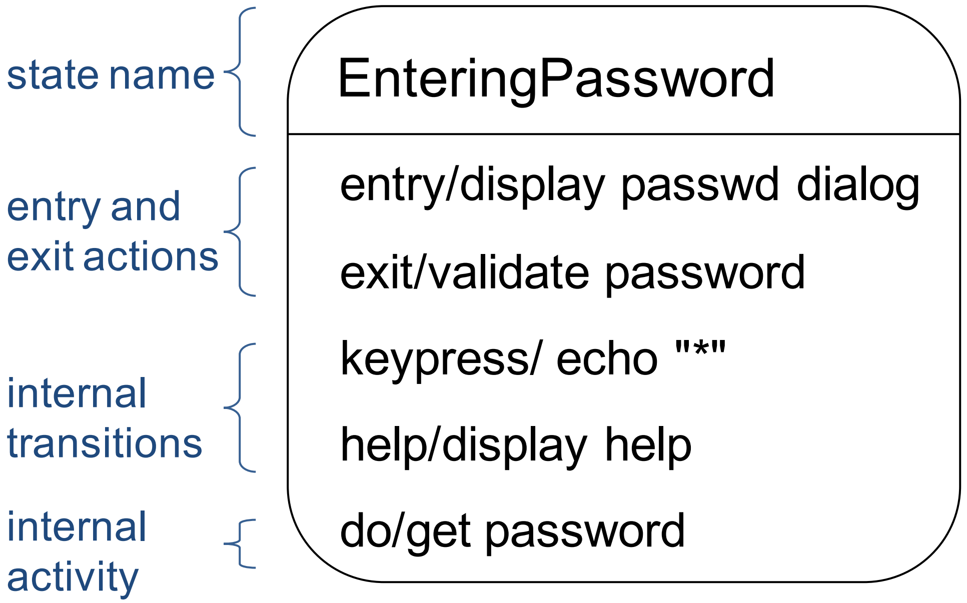
How many states?



State syntax

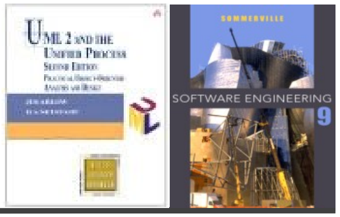


- ✧ Actions are **instantaneous** and **uninterruptible**
  - Entry actions occur immediately on state entry
  - Exit actions occur immediately on state leaving
- ✧ Internal transitions occur **within** the state. They do not fire transition to a new state
- ✧ Activities take a finite amount of time and are interruptible



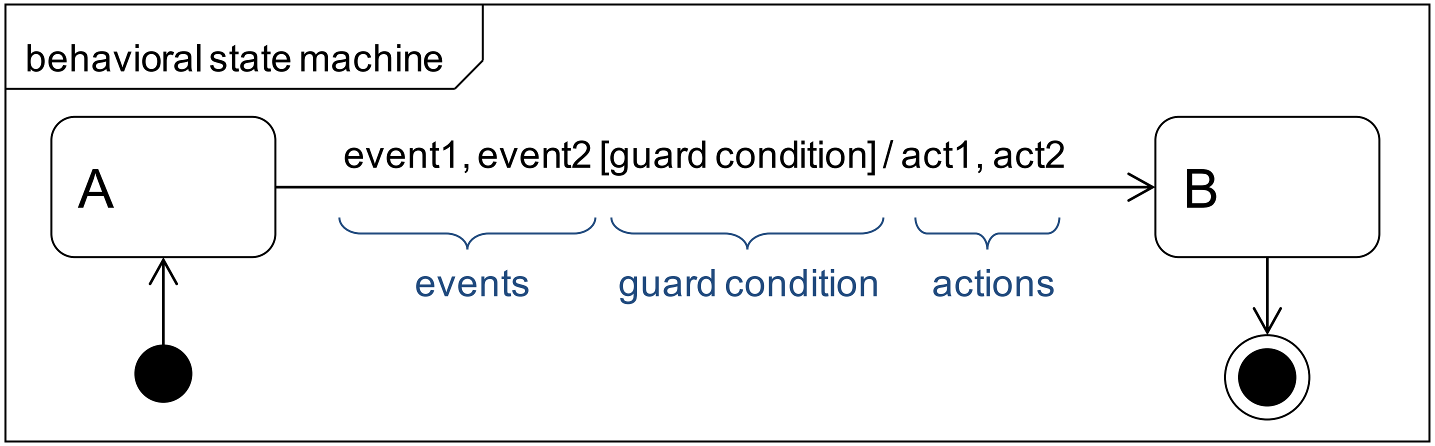
Action syntax: eventTrigger / action
Activity syntax: do / activity

Transitions



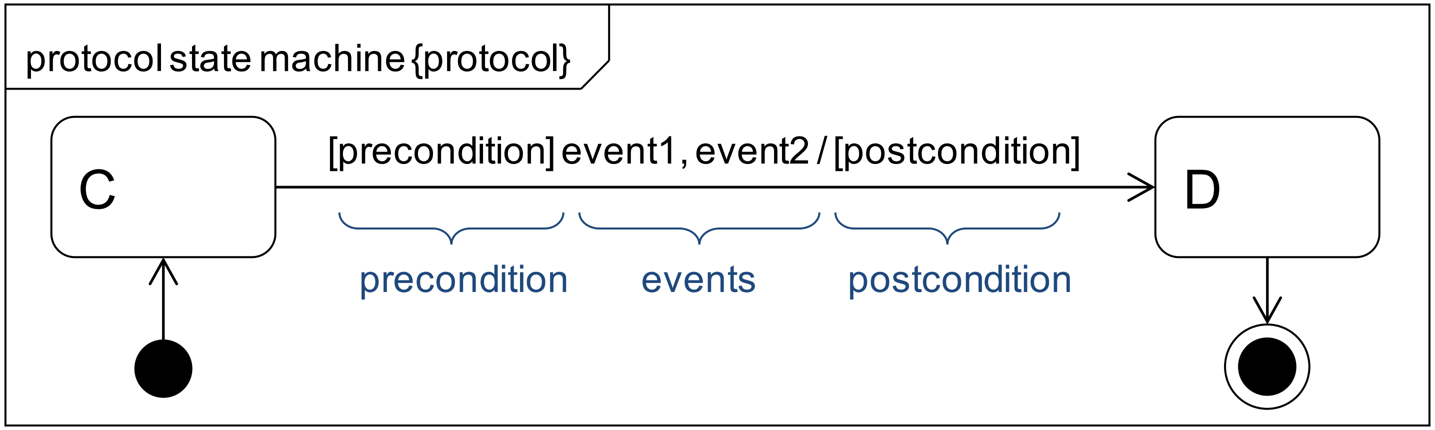
Behavioral state machine

Specifies object's reactions to events.



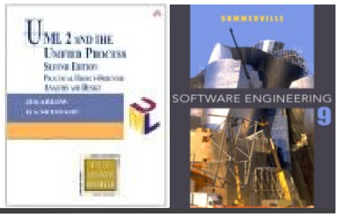
Protocol state machine

Specifies legal sequences of events.

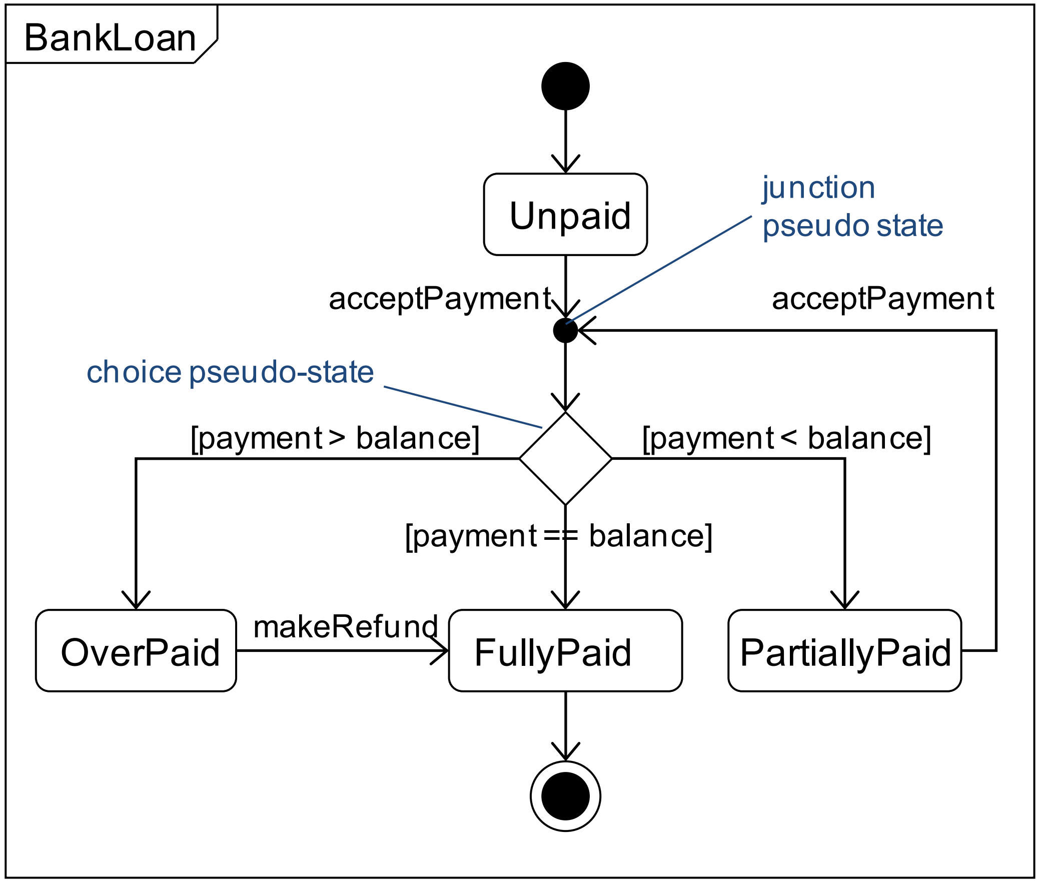


protocol state machine {protocol}
C
[precondition] event1, event2 / [postcondition]
D
precondition
events
postcondition

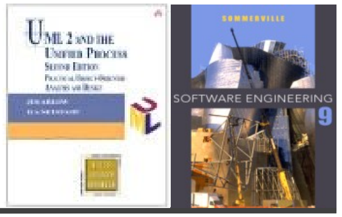
Choice and junction pseudo states



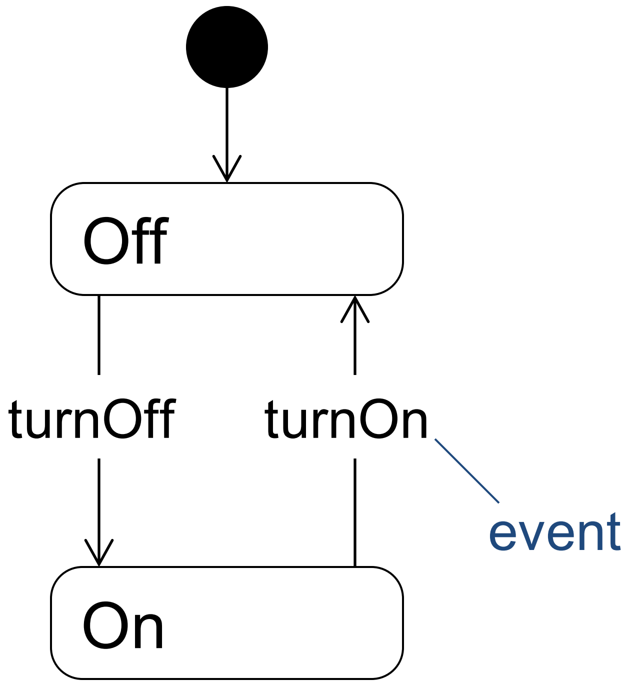
- **Choice pseudo state** directs its single incoming transition to one of its outgoing transitions
  - Each outgoing transition must have a mutually exclusive guard condition
  - Equivalent to two outgoing transitions from one state
- **Junction pseudo state** connects multiple incoming transitions into one (or more) transitions.
  - When there are more outgoing transitions, they must have guard conditions



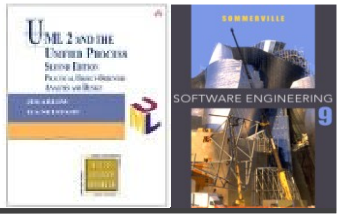
Events



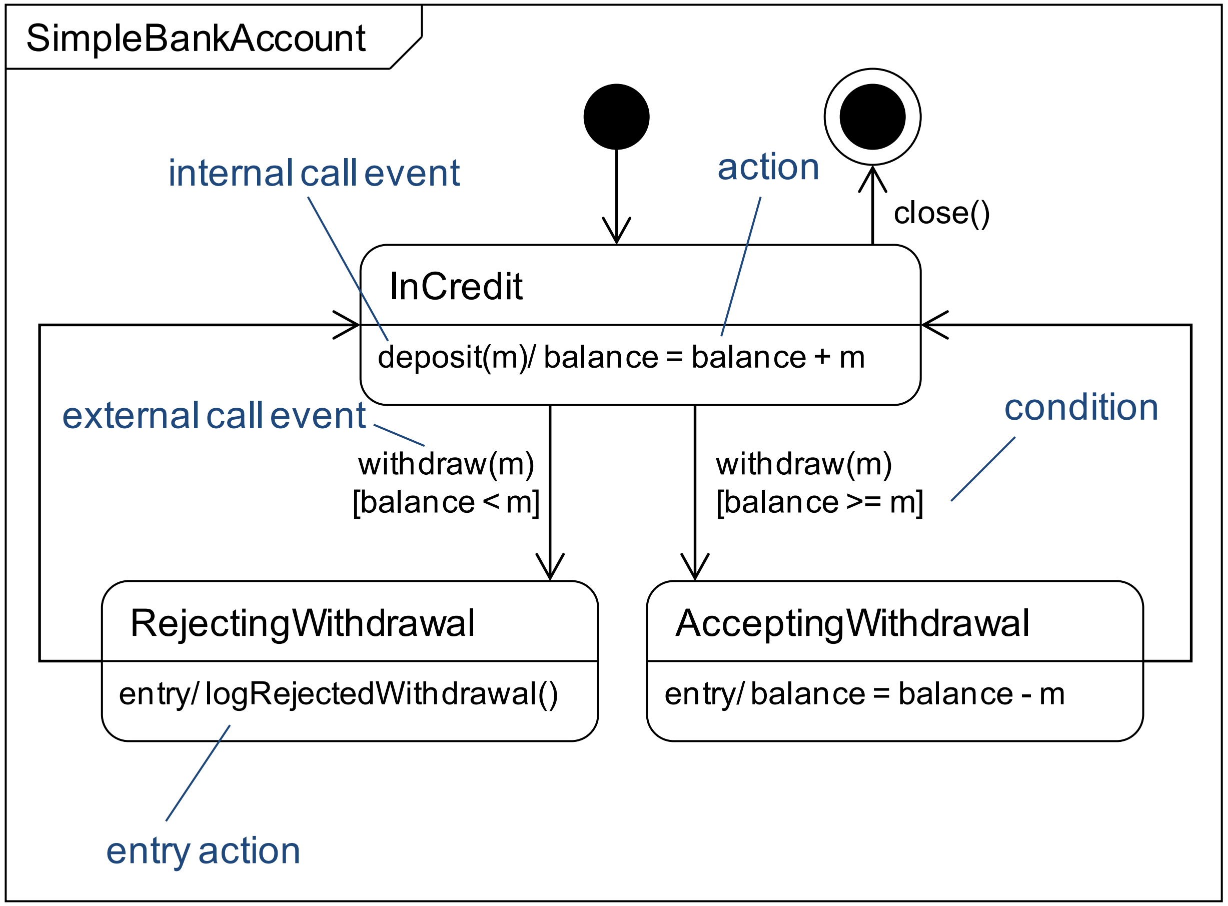
- "The specification of a noteworthy occurrence that has location in time and space"
- Events trigger transitions in state machines
- Events can be shown externally, on transitions, or internally within states (internal transitions)
- There are four types of event:
  - Call event
  - Signal event
  - Change event
  - Time event



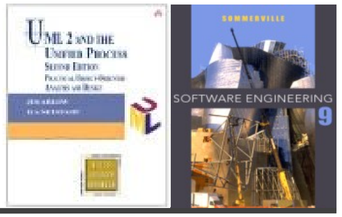
Call event



- A call for an operation execution
- The event should have the same signature as an operation of the context class
- A sequence of actions may be specified for a call event - they may use attributes and operations of the context class
- The return value must match the return type of the operation



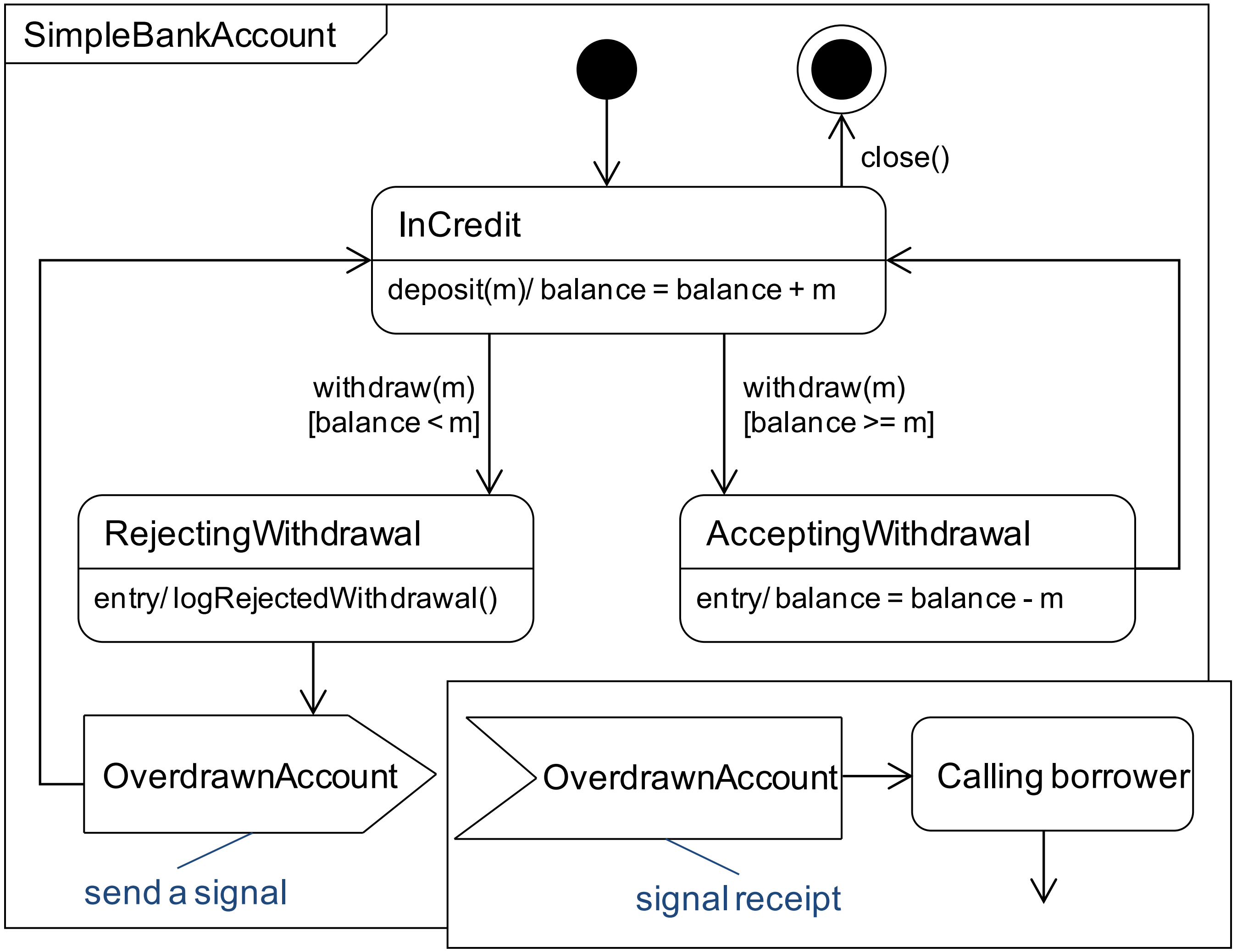
SimpleBankAccount
internal call event
action
close()
InCredit
deposit(m)/ balance = balance + m
external call event
condition
withdraw(m) [balance < m]
withdraw(m) [balance >= m]
RejectingWithdrawal
entry/ logRejectedWithdrawal()
AcceptingWithdrawal
entry/ balance = balance - m
entry action



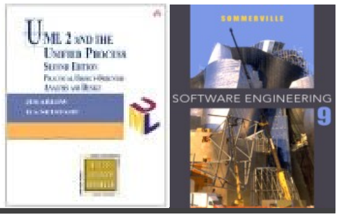
Signal events

- A signal is a package of information that is sent asynchronously between objects

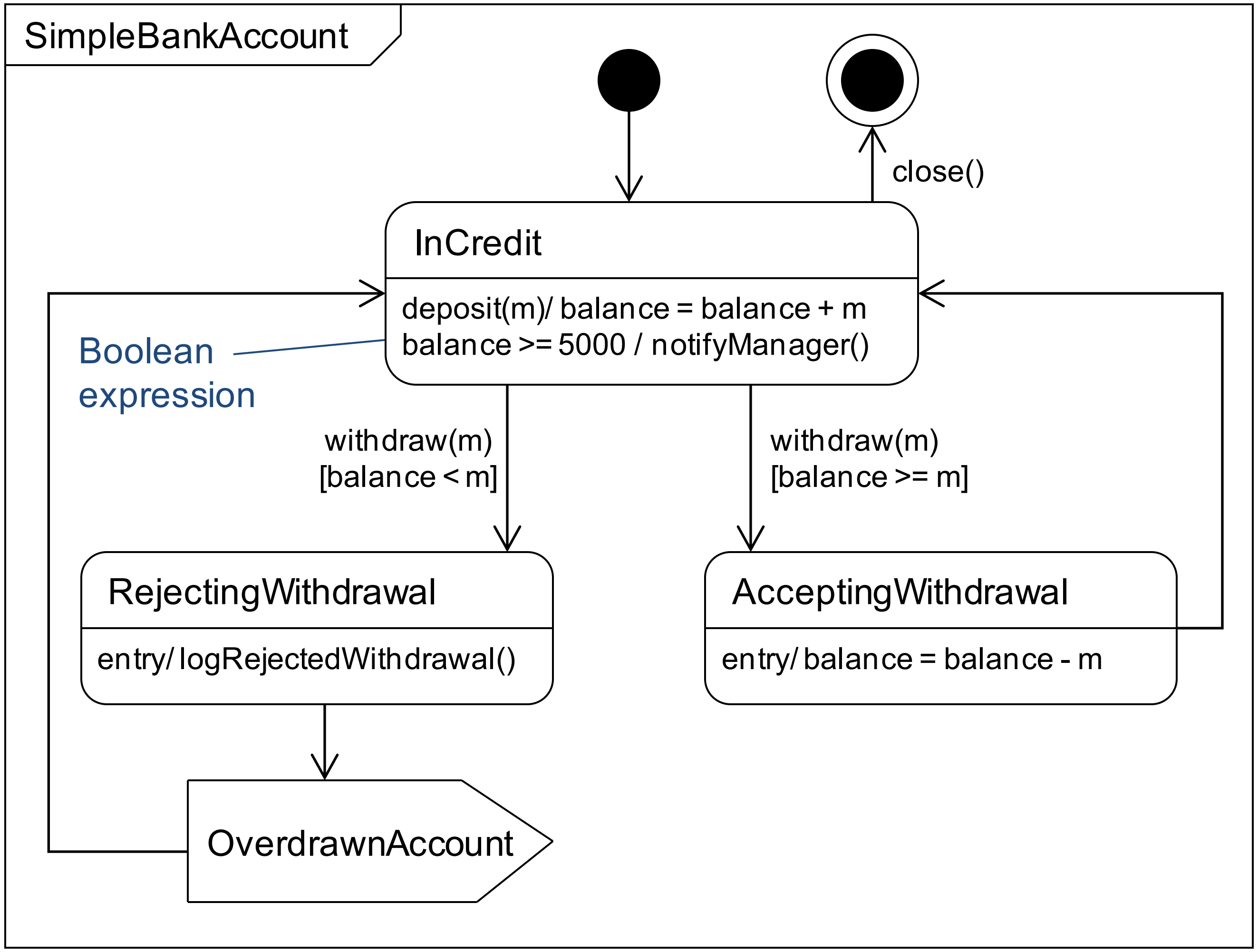
«signal» OverdrawnAccount
date : Date accountNumber : long amountOverdrawn : long



Change events

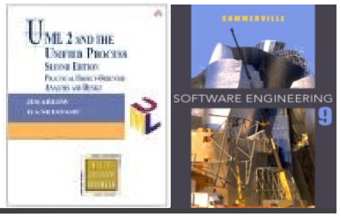


- The action is performed when the Boolean expression transitions from false to true
  - The event is *edge triggered* on a false to true transition
  - The values in the Boolean expression must be constants, globals or attributes of the context class
- A change event implies continually testing the condition whilst in the state

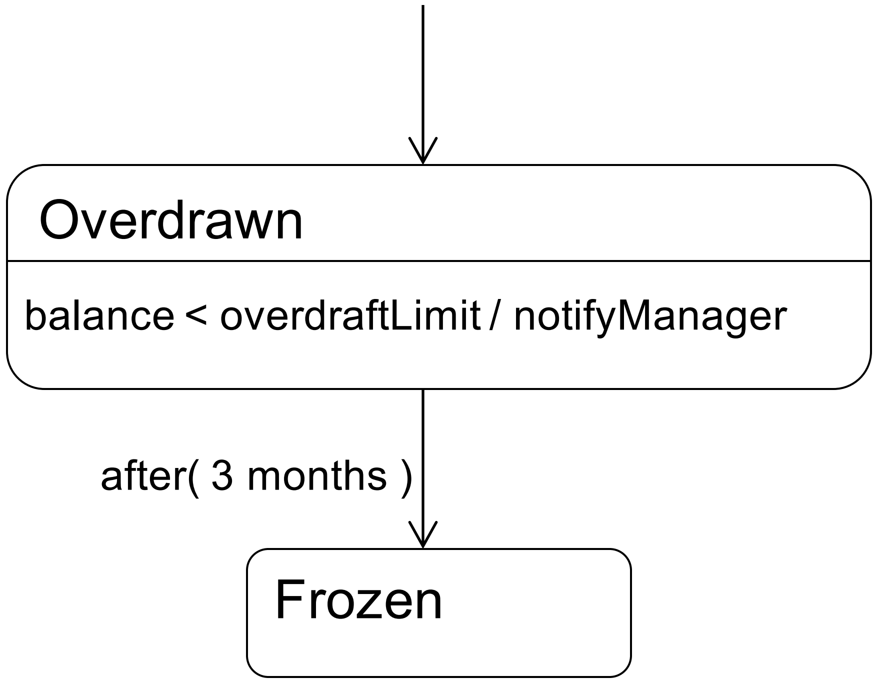


SimpleBankAccount
close()
InCredit
deposit(m)/ balance = balance + m
balance >= 5000 / notifyManager()
Boolean expression
withdraw(m) [balance < m]
withdraw(m) [balance >= m]
RejectingWithdrawal
entry/ logRejectedWithdrawal()
AcceptingWithdrawal
entry/ balance = balance - m
OverdrawnAccount

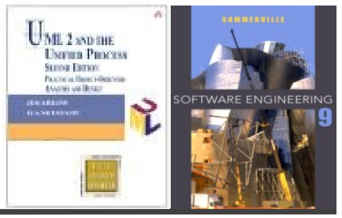
Time events



- Time events occur when a time expression becomes true
- There are two keywords, after and when
- Elapsed time:
  - after(3 months)
- Absolute time:
  - when(date =20/3/2000)

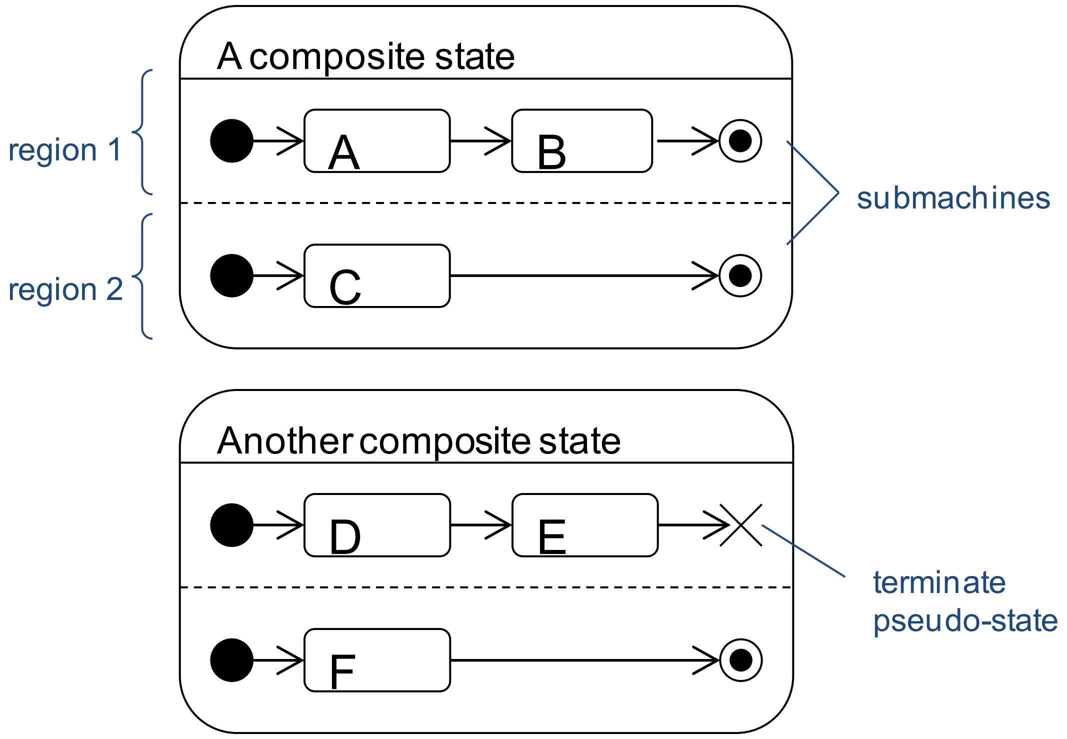


Context: CreditAccount class



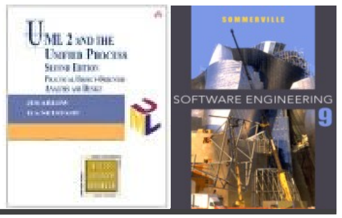
Composite states

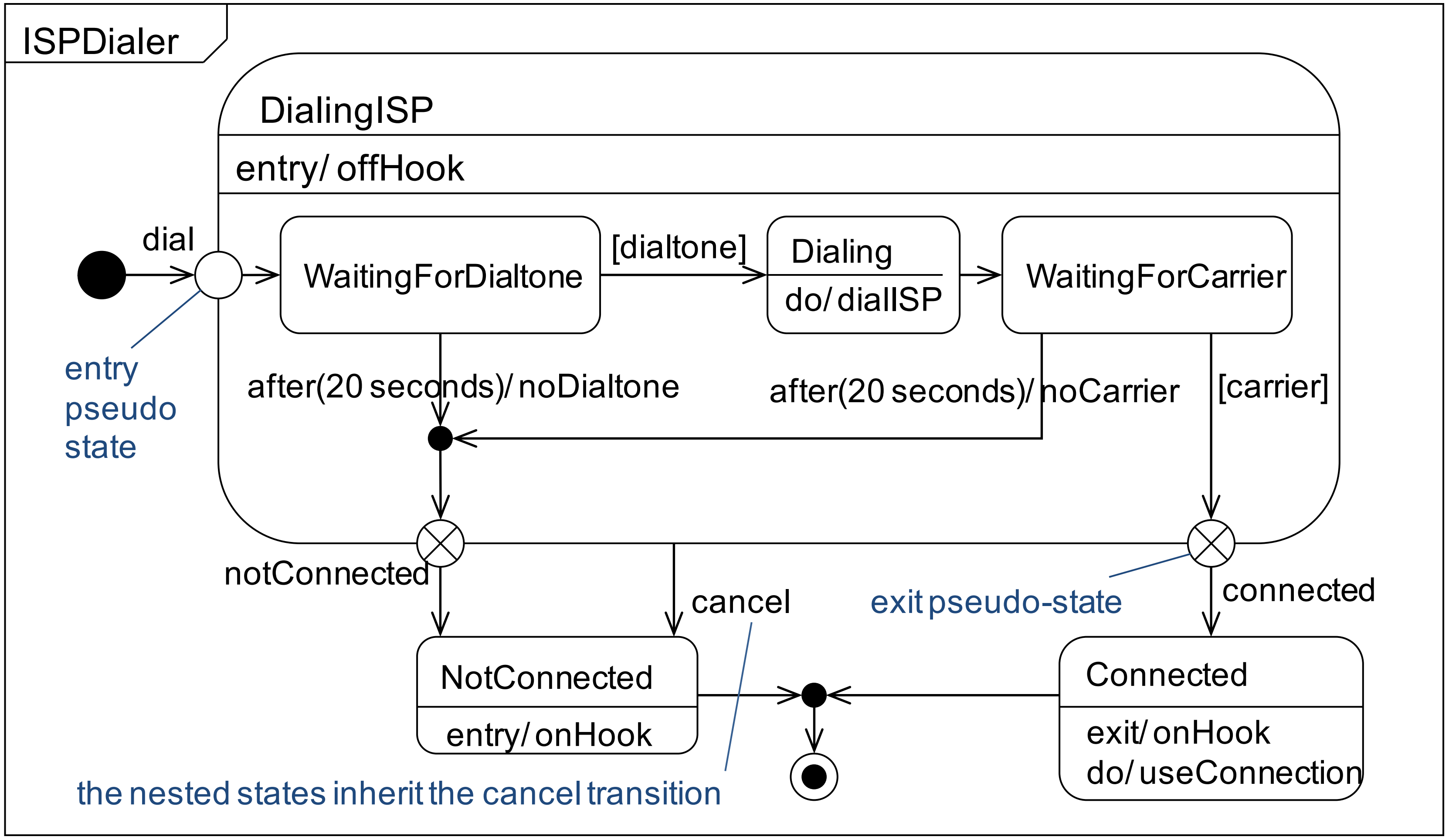
- Have one or more regions that each contain a nested submachine
  - Simple composite state
    - exactly one region
  - Orthogonal composite state
    - two or more regions
- The final state terminates its enclosing region – all other regions continue to execute
- The terminate pseudo-state terminates the whole state machine



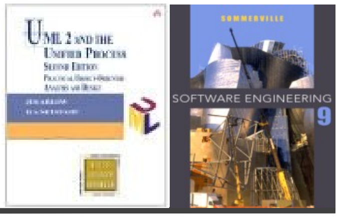
© Clear View Training 2010 v2.6

Simple composite states



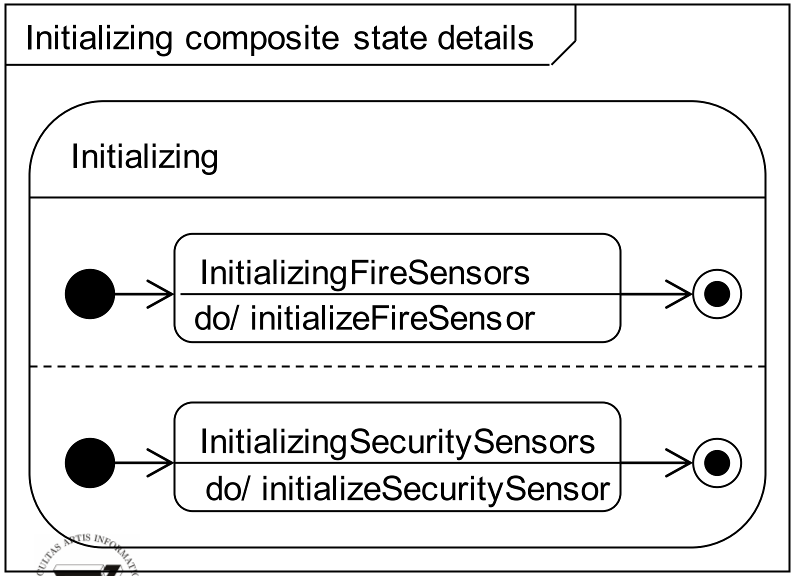


Orthogonal composite states

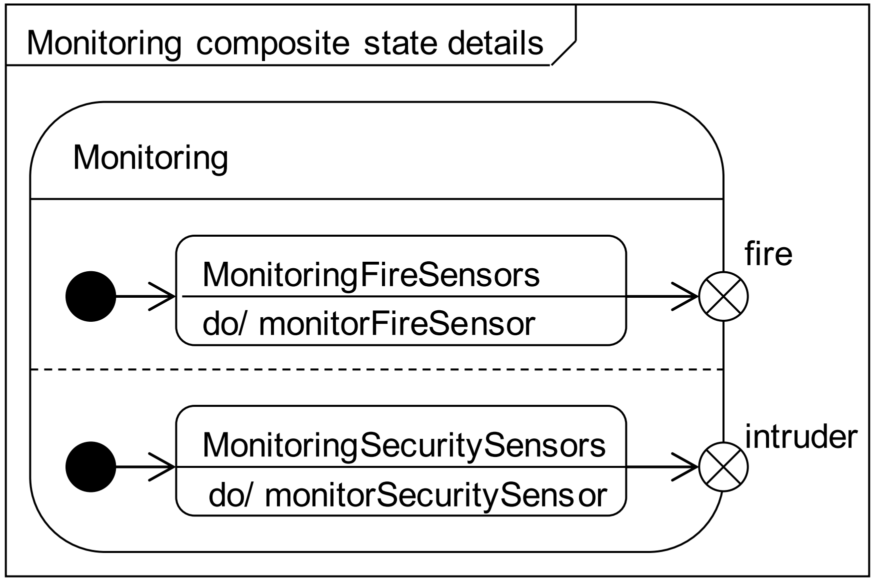


- ✧ Has two or more regions
- ✧ When we enter the superstate, both submachines start executing concurrently - this is an implicit fork

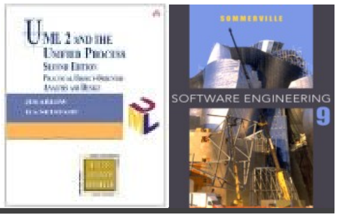
Synchronized exit - exit the superstate when *both* regions have terminated


Initializing composite state details
Initializing
InitializingFireSensors
do/ initializeFireSensor
InitializingSecuritySensors
do/ initializeSecuritySensor

Unsynchronized exit - exit the superstate when *either* region terminates. The other region continues


Monitoring composite state details
Monitoring
MonitoringFireSensors
do/ monitorFireSensor
fire
MonitoringSecuritySensors
do/ monitorSecuritySensor
intruder

Key points



- Behavioral and protocol state machines
- States
  - Initial and final
  - Exit and entry actions, activities
- Transitions
  - Guard conditions, actions
- Events
  - Call, signal, change and time
- Composite states
  - Simple and orthogonal composite states