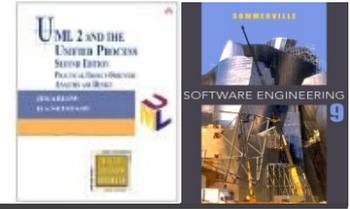


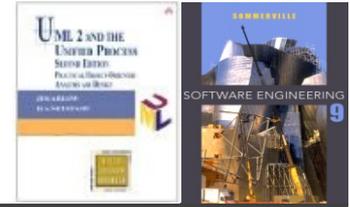
Course Summary and Outline of Advanced Software Engineering Techniques

Lecture 13

Topics covered



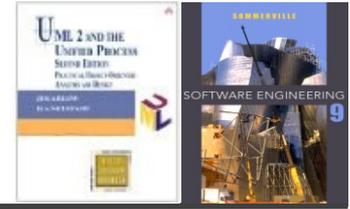
- ✧ Covered techniques of software engineering
- ✧ Outline of advanced techniques
- ✧ Covered UML diagrams
- ✧ Advanced UML modeling
- ✧ What comes next?



Covered Techniques of Software Engineering

Lecture 13/Part 1

Software process models



✧ Software engineering

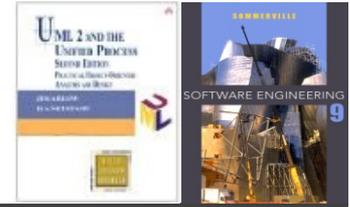
✧ Software process activities

- Software specification.
- Software analysis and design.
- Software implementation.
- Software validation.
- Software evolution.

✧ Software process models

- The waterfall model.
- Incremental development.
- Reuse-oriented software engineering

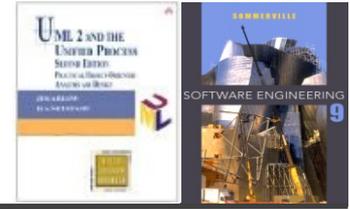
Requirements engineering



- ✧ Requirements and their types
 - User vs. system requirements
 - Functional vs. non-functional requirements
- ✧ Requirements specification
- ✧ Requirements engineering process
 - Requirements elicitation and analysis
 - Requirements validation
 - Requirements management

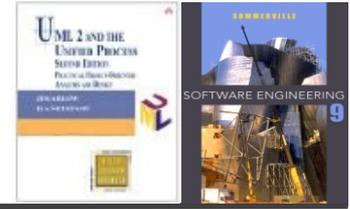
Focused on functional requirements mainly

Non-functional Requirements Engineering



- ✧ Non-functional requirements classification
- ✧ Non-functional requirements implementation
- ✧ Product requirements
 - Availability, Reliability, Safety, Security
 - Performance, Modifiability, Testability, Usability
- ✧ Organisational requirements
 - Development requirements
 - Operational requirements
 - Environmental requirements
- ✧ External requirements
 - Legislative requirements

Analysis and Design



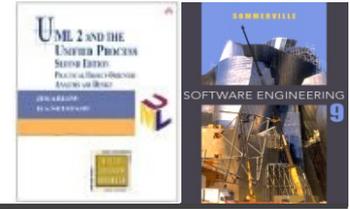
✧ Software analysis and design

- System context
- Architectural design
- Analysis and design models

✧ Structured vs. object-oriented methods

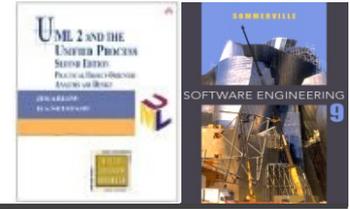
- Principles
- Notations
- Methods

Object-Oriented Analysis



- ✧ Role of the UML in OO analysis
- ✧ Objects and classes
- ✧ Finding analysis classes
- ✧ Relationships between objects and classes
- ✧ Inheritance and polymorphism

Structured Analysis



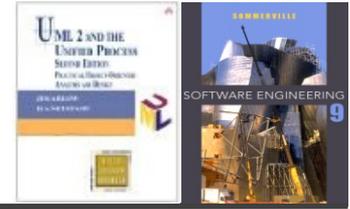
✧ Yourdon Modern Structured Analysis (YMSA)

- Context diagram (CD)
- Data flow diagram (DFD)

✧ Data modelling

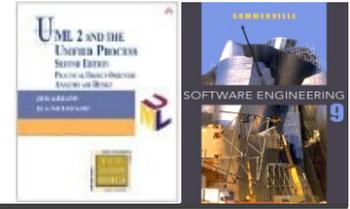
- Entity relationship diagram (ERD)
- Normalization and database design

System Design



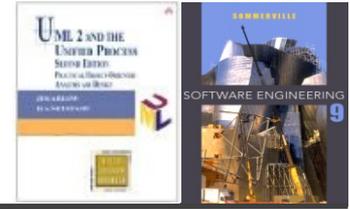
- ✧ Design patterns
- ✧ Design for dependability
 - Dependable processes
 - Redundancy and diversity
 - Dependable systems architectures
- ✧ Design for security
 - Design guidelines for security
 - System survivability
- ✧ Design for performance, modifiability, testability and usability

Architectural design



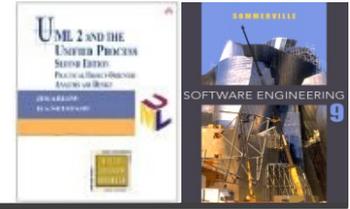
- ✧ Architectural design decisions
- ✧ Architectural patterns
 - Model-view-controller
 - Layered architecture
 - Repository architecture
 - Client-server architecture
 - Pipe-and-filter architecture
- ✧ Application architectures

Implementation



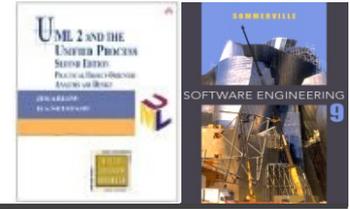
- ✧ Implementation issues
- ✧ Reuse
- ✧ Configuration management
- ✧ Host-target development
- ✧ Open-source development
- ✧ Programming guidelines

User Interface Design



- ✧ History and motivation
- ✧ Human limits
- ✧ Designing user interface
 - ✧ Fundamental UI design principles
 - ✧ Prominent positions on screen
- ✧ Evaluating user interface
- ✧ Examples

Testing, Verification and Validation



✧ Validation and verification

✧ Static analysis

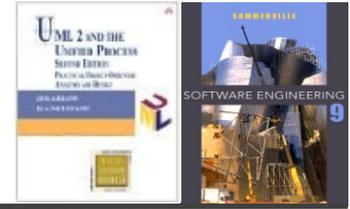
- Verification and formal methods
- Model checking
- Automated static analysis

✧ Testing and its stages

- Development testing
- Release testing
- User testing

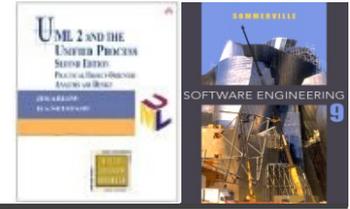
✧ Testing of non-functional properties

Operation, Maintenance and System Evolution



- ✧ Evolution processes
 - Change processes for software systems
- ✧ Program evolution dynamics
 - Understanding software evolution
- ✧ Software maintenance
 - Making changes to operational software systems
- ✧ Legacy system management
 - Making decisions about software change

Software Development Management



✧ Project management

✧ Project planning

- Scheduling
- Software pricing

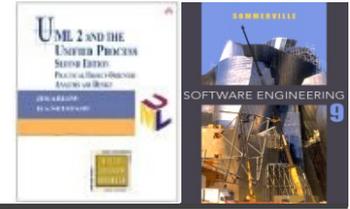
✧ Risk management

- Risk identification
- Risk analysis
- Risk planning
- Risk monitoring

✧ People management

- Motivation
- Teamwork

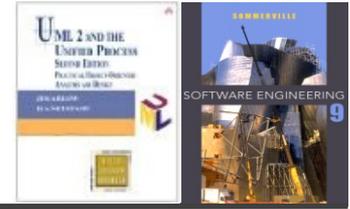




Outline of Advanced Techniques

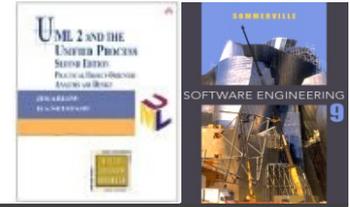
Lecture 13/Part 2

Software reuse



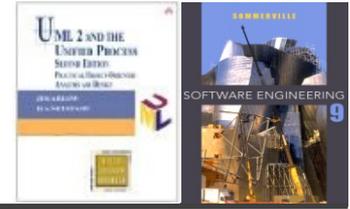
- ✧ In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.
- ✧ Software engineering has been more focused on original development but it is now recognised that to achieve better software, more quickly and at lower cost, we need a design process that is based on systematic software reuse.
- ✧ There has been a major switch to reuse-based development over the past 10 years.

Component-based software engineering



- ✧ Component-based software engineering (CBSE) is an approach to software development that relies on the reuse of entities called ‘software components’.
- ✧ It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific.
- ✧ CBSE essentials:
 - **Independent components** specified by their interfaces.
 - **Component standards** to facilitate component integration.
 - **Middleware** that provides support for component interoperability.
 - **A development process** that is geared to reuse.

Distributed systems



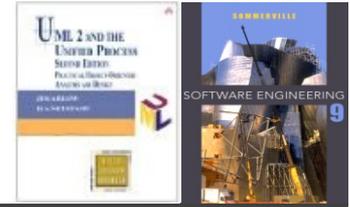
✧ Virtually all large computer-based systems are now distributed systems.

“... a collection of independent computers that appears to the user as a single coherent system.”

✧ Distributed systems issues

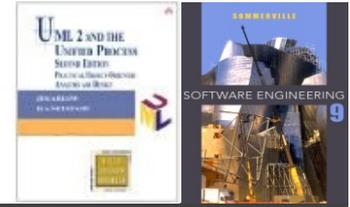
- Distributed systems are **more complex** than systems that run on a single processor.
- Complexity arises because different parts of the system are **independently managed** as is the network.
- There is **no single authority** in charge of the system so top-down control is impossible.

Service-oriented architectures



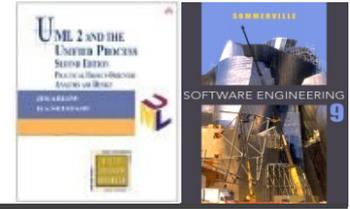
- ✧ A means of developing distributed systems where the components are stand-alone services
- ✧ Services may execute on different computers from different service providers
- ✧ Standard protocols have been developed to support service communication and information exchange
- ✧ Benefits of SOA:
 - Services can be provided **locally** or **outsourced** to ext. providers
 - Services are **language-independent**
 - Investment in **legacy systems** can be preserved
 - Inter-organisational computing is facilitated through **simplified information exchange**

Embedded systems

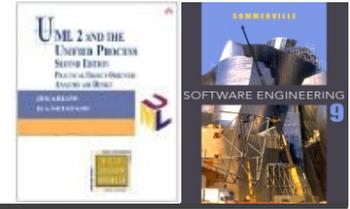


- ✧ Computers are used to control a wide range of systems from **simple domestic machines**, through **games controllers**, to entire **manufacturing plants**.
- ✧ Their software must react to events generated by the hardware and, often, issue control signals in response to these events.
- ✧ The software in these systems is **embedded in system hardware**, often in **read-only memory**, and usually responds, in **real time**, to events from the system's environment.
- ✧ Issues of **safety** and **reliability** may dominate the system design.

Aspect-oriented software development



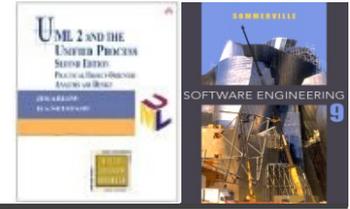
- ✧ An approach to software development based around a relatively new type of abstraction - **an aspect**.
- ✧ Used in conjunction with other approaches - normally object-oriented software engineering.
- ✧ Aspects encapsulate functionality that **cross-cuts** and co-exists with other functionality.
- ✧ Aspects include a definition of **where they should be included** in a program as well as **code implementing** the cross-cutting concern.



Covered UML Diagrams

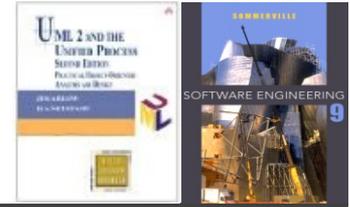
Lecture 13/Part 3

UML in Software Development



- ✧ System modeling
- ✧ External perspective models
 - **Use case diagram**
- ✧ Structural perspective models
 - **Class diagram**, Object diagram, Component diagram, Package diagram, Deployment diagram, Composite structure diagram
- ✧ Interaction perspective models
 - **Sequence diagram**, Communication diagram, Interaction overview diagram, Timing diagram
- ✧ Behavioral perspective models
 - **Activity diagram**, State diagram

UML Use Case Diagram



✧ Use Case modelling

- System boundary – subject
- Use cases
- Actors

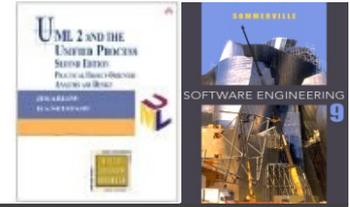
✧ Textual Use Case specification

- Branching with IF
- Repetition with FOR and WHILE
- Alternative flows

✧ Advanced Use Case modelling

- Actor generalisation
- Use case generalisation
- «include»
- «extend»

UML Activity Diagram

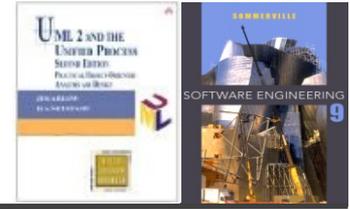


✧ Activity diagrams can model flows of activities using:

- Activities and connectors
- Activity partitions
- Action nodes
 - Call action node
 - Send signal/accept event action node
 - Accept time event action node
- Control nodes
 - Decision and merge
 - Fork and join
- Object nodes
 - Input and output parameters
 - Pins

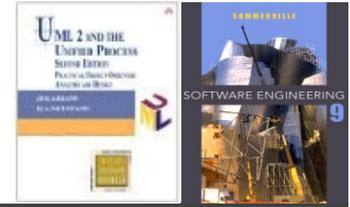
✧ **Interaction overview diagrams** as their advanced feature

UML Class Diagram



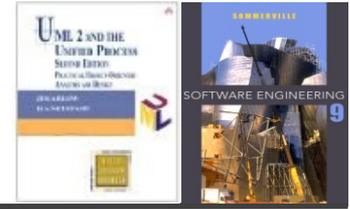
- ✧ Analytical vs. Design class model
- ✧ Objects and classes
- ✧ Relationships between objects and classes
 - Links
 - Associations
 - Aggregation and composition
 - Dependencies
- ✧ Inheritance and polymorphism

UML Interaction Diagrams



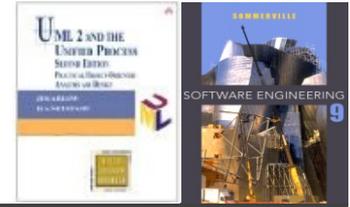
- ✧ There are four types of interaction diagram:
- **Sequence diagrams** – emphasize time-ordered sequence of message sends
 - **Communication diagrams** – emphasize the structural relationships between lifelines
 - **Interaction overview diagrams** – show how complex behavior is realized by a set of simpler interactions
 - **Timing diagrams** – emphasize the real-time aspects of an interaction

UML Packages



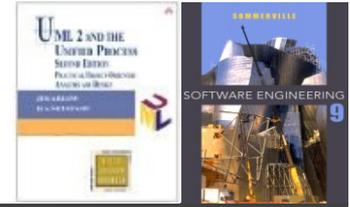
- ✧ Packages are the UML way of grouping modeling elements
- ✧ There are dependency and generalisation relationships between packages
- ✧ The package structure of the analysis model defines the logical system architecture

UML Component Diagram



- ✧ Interfaces specify a named set of public features:
 - They define a contract that classes and subsystems may realise
 - Programming to interfaces rather than to classes reduces dependencies between the classes and subsystems in our model
 - Programming to interfaces increases flexibility and extensibility
- ✧ Design subsystems and interfaces allow us to:
 - Componentize our system
 - Define an architecture

UML Deployment Diagram



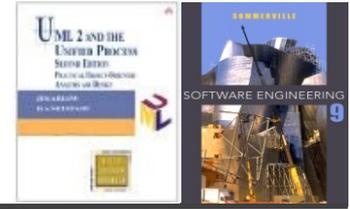
✧ The descriptor form deployment diagram

- Allows you to show how functionality represented by artefacts is distributed across nodes
- Nodes represent types of physical hardware or execution environments

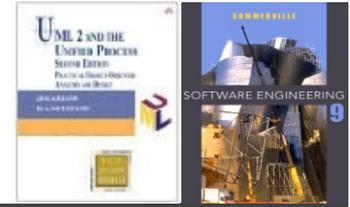
✧ The instance form deployment diagram

- Allows you to show how functionality represented by artefact instances is distributed across node instances
- Node instances represent actual physical hardware or execution environments

UML State Diagram



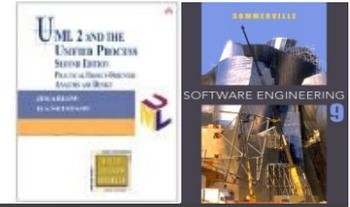
- ✧ Behavioral state machines
- ✧ Protocol state machines
- ✧ States
 - Actions, exit and entry actions, activities
- ✧ Transitions
 - Guard conditions, actions
- ✧ Events
 - Call, signal, change and time
- ✧ Composite states
 - Simple and orthogonal composite states



Advanced UML Modeling

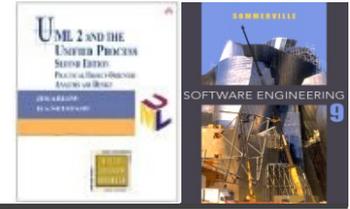
Lecture 13/Part 4

Advanced Activity diagrams



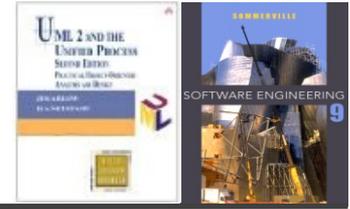
- ✧ Connectors
- ✧ Interruptible activity regions
- ✧ Exception handling
- ✧ Expansion nodes
- ✧ Signals and events
- ✧ Streaming
- ✧ Advanced object flow features
- ✧ Multicast and multireceive
- ✧ Parameters

Advanced Interaction diagrams



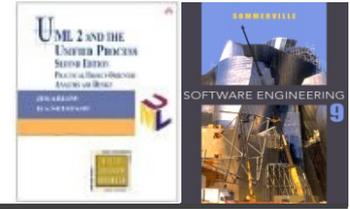
- ✧ Timing diagram
- ✧ Interaction overview diagram

Advanced State diagrams



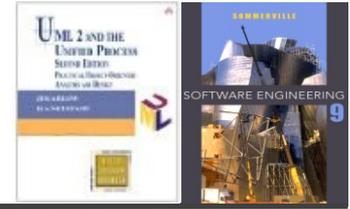
- ✧ Composite states
- ✧ Submachine states
- ✧ Submachine communication
- ✧ History

Object constraint language (OCL)

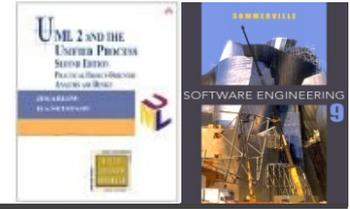


- ✧ The Object Constraint Language (OCL) is a declarative language for describing rules that apply to UML models.
 - The OCL is a precise text language that provides constraint and object query expressions.
- ✧ OCL statements are constructed in four parts:
 - a **context** that defines the limited situation in which the statement is valid
 - a **property** that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute)
 - an **operation** (e.g., arithmetic, set-oriented) that manipulates or qualifies a property, and
 - **keywords** (e.g., if, then, else, and, or, not, implies) that are used to specify conditional expressions.

UML Profiles



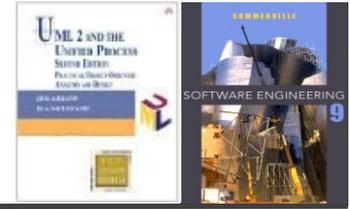
- ✧ A UML profile provides a **generic extension mechanism for customizing UML models** for particular domains and platforms.
 - Extension mechanisms allow refining standard semantics in strictly additive manner, so that they can't contradict standard semantics.
- ✧ Profiles are defined using **stereotypes, tag definitions, and constraints** that are applied to specific model elements, such as Classes, Attributes, and Activities.
- ✧ A Profile is a **collection of such extensions** that collectively customize UML for a particular domain (e.g., aerospace, healthcare, financial) or platform (J2EE, .NET).



What Comes Next?

Lecture 13/Part 5

Course finalization



✧ Seminar projects

- Assessment, “Úspěšné absolvování cvičení“ IS notebook

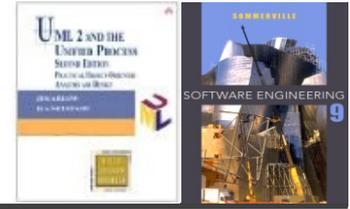
✧ Exam

- Number of exam dates
- Reservation/cancelation policies
- Length of the exam
- Form of the exam – test part and UML modelling part
- Results and their viewing

✧ Opinion poll

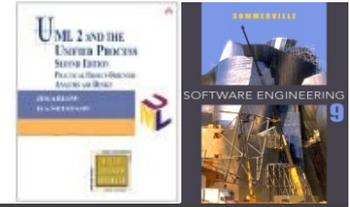
- This is the first year of this course
- Do not forget to give us your feedback! 😊

Follow-up and related courses



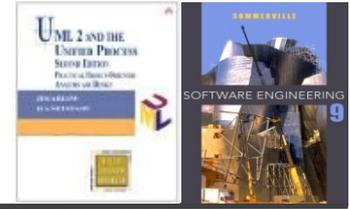
- ✧ PA103 Objektové metody návrhu informačních systémů
- ✧ PA102 Technologie informačních systémů I, II
- ✧ PV167 Projekt z objektového návrhu inf. systémů
- ✧ PA104 Vedení týmového projektu
- ✧ PV207 Business Process Management
- ✧ PV165 Procesní řízení
- ✧ PV045 Management informačního systému
- ✧ PA189 Agile Management in IT
- ✧ PV028 Aplikační informační systémy

Follow-up and related courses



- ✧ PV043 Informační systémy podniků
- ✧ PV230 Podnikové portály
- ✧ PV019 Geografické informační systémy I, II
- ✧ PV058 Informační systémy ve veřejné a státní správě
- ✧ PV213 Enterprise Information Systems in Practice
- ✧ PV098 Řízení implementace IS
- ✧ PB168 Základy databázových a informačních systémů
- ✧ PB114 Datové modelování I
- ✧ SSME Courses

Thanks



Thank you for your attention
and good luck with the exam!