Software Engineering

Chapter 3

Software Processes
Chapter Objectives

- To introduce software **process models**
- To describe a number of different process models and when they may be used
- To describe **outline process models** for:
  - requirements engineering
  - software development
  - testing and evolution
- To introduce **CASE** technology that supports software process activities
Software Processes - Software Life Cycle

Sets of activities for:

- specifying,
- designing,
- implementing and
- testing software systems
Software Development Effort Distribution

- Code/Unit Test: 40%
- Design: 21%
- Analysis: 20%
- Deployment: 9%
- Documentation: 10%
The software process: The 5 fundamental activities

- A structured set of fundamental activities required to develop a software system

1. **Specification (Analysis):**
   s/w requirements (functional, non-functional, environment/domain)
2. **Design**
   s/w design according to requirements specification
3. **Implementation:**
   Production of s/w meeting specification
4. **Validation:**
   Ensure that the s/w does what the clients want

**Post development activity**
5. **Evolution:**
   Maintenance - Meeting Dynamic environment changes
The software process (cont.)

- Software process model:
  - An abstract representation (simple/complex) of a process.
  - It presents a description of a process from some particular perspective

- Many organisation still rely on ad-hoc processes
  - no use of s/w processes methods
  - no use of best practice in s/w industry
Software process models

The 4 generic models (see details on next slides):

1. The waterfall model
2. Evolutionary development
3. Formal systems development
4. Reuse-based (Component-based) development
Software process models – The 4 generic models

1. The waterfall model
   - Separate and distinct phases of specification and development: Requirements, design, implementation, testing, …
   - No evolution process, only development
   - Widely used & practical
   - Recommended when requirements *are well known and stable at start*

2. Evolutionary development
   - Specification and development are interleaved
   - Develop rapidly & refine with client
   - Widely used & practical
   - Recommended when requirements *are not well known at start*
Software process models – The 4 generic models (cont.)

3. Formal systems development

- A mathematical system model is formally transformed to an implementation
- Produce a **formal mathematical** specs
- Transforms specs using mathematical methods to construct a program
- True implementation of specs
  » No testing for Defects is needed
  » Clean Room process (IBM)
- Not widely used
- Recommended for systems having security, reliability, safety .. requirements
4. Reuse-based (Component-based) development

- The system is \textit{assembled} from existing components
  - Components already developed within the organization
  - COTS “Commercial of the shelf” components
- Integrating rather than developing
- Allows rapid development
- Gaining more place
- Future trend
Waterfall model

- Requirements definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance
Waterfall model principle phases

1. Requirements analysis and definition *(client side)*
   - Functional, non-functional, domain (environment) requirements and constraints

2. System and software design
   - Overall sys architecture (block diagram)
   - S/w components/ interface relationships – DB design – I/O design

3. Implementation and unit testing
   - Coding, testing each unit subsystem

4. Integration and system testing

5. Operation and maintenance *(client side)*
Waterfall model problems

- Drawback: the difficulty of accommodating change after the process is underway
- Inflexible partitioning of the project into distinct stages
- Inflexible: to respond to dynamic business environment leading to requirements changes
- Appropriate when the requirements are well-understood and stable
Evolutionary development model
- Prototyping

- Develop an initial implementation prototype
- Client test drive … feedback
- Refine prototype
Evolutionary development model – Prototyping (cont.)

Concurrent activities

Outline description

Development

Validation

Specification

Initial version

Intermediate versions

Final version

Time
Evolutionary development model – Prototyping (cont.)

- **Problems**
  - Lack of process visibility at client management level (less regular reports/documentation ... the sys is changing continuously)
  - Systems are often poorly structured
  - Special skills (e.g. in languages/tools for rapid prototyping) may be required

- **Applicability**
  - For small or medium-size interactive systems
  - For parts of large systems (e.g. the user interface)
  - For short-lifetime systems
Formal systems development

- Based on the transformation of a mathematical specification through different representations to an executable program

- Uses special languages, e.g. Z-language

- Transformations are ‘correctness-preserving’
  - The program conforms to its specification

- Embodied in the ‘Cleanroom’ approach to software development
Formal systems development

- Requirements definition
- Formal specification
- Formal transformation
- Integration and system testing
Formal transformations

Formal transformations: T1, T2, T3, T4

T1: Formal specification → P1
T2: R1 → P2
T3: R2 → P3
T4: R3 → P4

Executable program

Proofs of transformation correctness: P1, P2, P3, P4
Formal systems development

- **Problems**
  - Need for specialised skills and training to apply the technique
  - Difficult to formally specify some aspects of the system such as the user interface

- **Applicability**
  - Critical systems especially those where a safety or security case must be made before the system is put into operation
Reuse-oriented “Component-Based” development

- Based on systematic reuse
- Systems are integrated from
  - Existing components (available in the library of the software house)
  - COTS (Commercial-off-the-shelf) systems (available or to be purchased)

**Advantages**

- Minimum s/w development
  - Less cost
  - Less risk
  - Less time
  - Reliable
  - Less testing

- Gaining more place in the market
Reuse-oriented development (cont.)

Problems:

- Relies on the availability of re-usable s/w components
- Compromised requirements to match re-use components
- Less control over system evolution (components new versions are not under control of developers)
Reuse-oriented development (cont.)

Process stages

- Component analysis
  - Search for components covering most required functionality

- Requirements modification
  - To match existing reuse components

- System design with reuse
  - Design framework for reuse components
  - Design non available reuse components

- Development and integration
  - Develop non available reuse components
  - Integration of reuse & developed components
Reuse-oriented development (cont.)
Process iteration

- Specs are developed **alongside** s/w

- There is no complete sys specs until the final increment is specified

- Two approaches for iteration
  - **Incremental** development
  - **Spiral** development
Software process models - Comparison

- Waterfall model
  - Requirements should be well defined *at start and committed to*

- Evolutionary model
  - Requirements & design decisions may be delayed: Poor structure - difficult to maintain

- Incremental development
  - Incremental *prioritised delivery of modules*
  - *Hybrid* of waterfall & Evolutionary
Incremental development (cont.)

- **Incremental delivery**: *each increment delivering part of the required functionality*

- User requirements are *prioritised* and the highest priority requirements are included in early increments
Incremental development (cont.)

- Increments should be small (< 20,000 LOC)
- The 4 generic process models may be used according to the current increment under consideration
Incremental development (cont.)

Define outline requirements → Assign requirements to increments → Design system architecture

Develop system increment → Validate increment → Integrate increment → Validate system

System incomplete: Start next increment → Final system
Incremental development advantages

- System functionality is available earlier
- Early involvement of client
- Early increments act as a prototype to help elicit requirements for later increments
- Lower risk of overall project failure
- High priority increment delivered first
- The highest priority system services tend to receive the most testing (first delivered)
Agile Development:
Extreme Programming “XP”
eXtreme Programming XP

- New approach to development based on the development and delivery of **very small increments of functionality**
- Relies on:
  - Constant code improvement
  - User involvement in the development team

- See link to Chapter on XP:
  - [RC4_Extreme Programming.ppt](http://example.com/RC4_Extreme_Programming.ppt)

See [www.extremeprogramming.org](http://www.extremeprogramming.org)
eXtreme Programming XP (cont.)

- Extreme Programming is the most famous of the agile methods
- Short inspect-and-adapt cycles and frequent, short feedback loops
  - working software is the primary measure of progress
  - Embrace changing requirements
- Based on the recognition that separating design, evaluation, and documentation activities in software development is a futile exercise
eXtreme Programming
XP 12 practices (cont.)

1. Planning Game
   - At the beginning of each release, the stakeholders negotiate the feature to realize.
   - The business people decide how important a feature is, and the developers decide how much that feature will cost to implement (user stories).

2. Small Releases
   - Small increments that result in a deliverable, working application

3. Testing
   - Black-box test cases are written before the corresponding code. This approach gives confidence to make modifications
eXtreme Programming
XP 12 practices (cont.)

4. Pair Programming
   • significant coding is done in pairs of programmers (working at the same workstation). An extra set of eyes has been shown to reduce the defect rate of software.

5. Refactoring
   • performing a number of small, and frequently applied, transformations, which improves structure of code without affecting its behavior

6. Continuous Integration
   • describes the immediate and ongoing integration of completed tasks into the system
eXtreme Programming
XP 12 practices (cont.)

7. Simple Design
   • consider the simplest thing that could possibly work and do that or the next best thing don’t predict what is coming next because it probably isn't

8. Collective Code Ownership
   • distribute skills and knowledge as much as necessary or possible (to avoid the “truck” factor)

9. On-site Customer
   • they specify requirements using user stories and are present to answer questions regarding them
   • they serve as an information source
10. **Coding Standards**

- use of uniform, consistent coding standards simplifies working on the source code
- offer recommendations to developers with respect to:
  - how to lay out the code, e.g. tabbing, braces, comments
  - where source code files should be located in the file system
  - which settings should be used for the compiler and linker
  - which naming conventions should be followed for files, classes, methods, attributes, parameters, and local variables.
Extreme programming
XP 12 practices (cont.)

11. Sustainable Pace
   • avoid overworking (keep to 40 hour weeks)

12. Metaphor
   • A metaphor is “a rhetorical figure, which expresses what was meant by using an imagination (mostly a picture), which stems from a completely different domain and which has no concrete relation to what was meant”
   • Metaphors allow developers and customers to communicate understanding through verbal pictures
Spiral development

- Best features of waterfall & prototyping models

  + Risk Analysis (missed in other models)

- Process is represented as a spiral rather than as a sequence of activities with backtracking

- Each loop in the spiral represents a phase in the process.
- No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required

- Risks are explicitly assessed and resolved throughout the process
Spiral Development
Spiral model of the software process

- Determine objectives, alternatives, and constraints
- Risk analysis
- Prototype 1
- Prototype 2
- Prototype 3
- Operational prototype
- Risk analysis
- Simulations, models, benchmarks
- Concept of Operation
- S/W requirements
- Product design
- Detailed design
- Code
- Unit test
- Integration test
- Acceptance test
- Service
- Develop, verify next-level product
- Plan next phase
- REVIEW
- Requirements plan
- Life-cycle plan
- Development plan
- Integration and test plan
- Requirement validation
- Design V&V
- Integration test
- Product design
- Detailed design
- Code
- Unit test
- Integration test
- Acceptance test
- Service
- Develop, verify next-level product
- Plan next phase
- REVIEW
- Requirements plan
- Life-cycle plan
- Development plan
- Integration and test plan
- Requirement validation
- Design V&V
- Integration test
- Acceptance test
- Service
- Develop, verify next-level product
- Plan next phase
Spiral model  4 sectors

1. **Objective setting**
   - Specific objectives for the phase are identified

2. **Risk analysis**
   - Risks are assessed and activities put in place to reduce the key risks

3. **Development and validation**
   - A development model for the system is chosen which can be any of the *generic 4 models*
   - Simulation, benchmarks: to further define requirements

4. **Reviewing/Planning**
   - Review with client
   - Plan next phase of the spiral if further loop is needed
Development Process & Risk Analysis/Risk Minimisation

- If Uncertainty of requirement risk:
  - Then, Prototyping model

- If User interface risk:
  - Then, Prototyping model

- If Security risks:
  - Then, Formal transformation model

- If Sub-system integration risk:
  - Then, Waterfall model
Software Specification Process

Requirements Engineering Process
The Requirements Engineering Process

- Feasibility study
- Requirements elicitation and analysis
- Requirements specification
- Requirements validation
- System models
- User and system requirements
- Requirements document
Software specification Process (Requirements Engineering Process)

- The process of establishing
  - What services are required (Functional requirements)
  - Constraints on the system’s operation and development ((Non-functional requirements)

- Requirements engineering process
  1. Feasibility study
     - Alternatives & Quick cost/benefit analysis
     - Feasibility: Technical, Financial, Human, Time schedule
     - Deliverables: Feasibility report
  2. Requirements elicitation and analysis: Facts finding
     - Interviews, JAD “Joint Application Development”, Questionnaires, Document inspection, Observation
     - Deliverables: System models (Diagrams)
The Requirements Engineering Process

3. Requirements specification
   - User level: abstract specification
   - System level: detailed specification
   - **Deliverables**: User and system requirements

4. Requirements validation for
   - Completeness
   - Consistency
   - Realism
   - **Deliverables**: Updated requirements

**Global Deliverables of the Requirements Eng Process**:
System Requirements Specification document
Software Design and Implementation Process

- **Software design**
  - System architecture design
  - Software structure that realises the specification
  - Interface, Data structures, Database, Algorithms, GUI

- **Implementation**
  - Translate design into an executable program
Design Process Activities

1. Architectural design
2. Abstract specification
3. System/subsystems Interface design
4. Component design
5. Data structure (Database) design
6. Algorithm design
Design Process Activities (cont.)

1. Architectural design
   - Subsystems/relationships, block diagram
   - **Deliverables**: System architecture

2. Abstract specification for each subsystem
   - **Deliverables**: S/W specs for each subsystem (services & constraints)

3. System/subsystems Interface design
   - With other subsystems of the sys
   - With external systems (Bank, GOSI, …)
   - **Deliverables**: Interface specs for each subsystem in relation to other subsystems or external systems
4. **Component design**
   - Services are allocated to components
   - Components interfaces are designed
     - Interfaces with other components of the system
     - Interfaces with external systems
     - GUI
     - Input
     - Output
   - **Deliverables**: Component specs
Design Process Activities (cont.)

5. **Data structure (Database) design**
   - Detailed design of data structure to be implemented (design or implementation activity)
   - **Deliverables:** Data structure specs

6. **Algorithm design**
   - Detailed design of algorithm for services to be implemented (design or implementation activity)
   - **Deliverables:** Algorithm specs
The software design process

- Requirements specification
- Architectural design
- Abstract specification
- Interface design
- Component design
- Data structure design
- Algorithm design

Design activities

System architecture
Software specification
Interface specification
Component specification
Data structure specification
Algorithm specification

Design deliverables (products)
Design methods

- Systematic approaches to developing a software design
- Structured methods: Set of notations & guidelines for s/w design
  - Graphical methods
  - CASE tools
- The design is usually documented as a set of graphical models
- Possible models
  - Process Model (Data-flow model)
  - Information/Data model (Entity-relation-attribute model)
  - Structural model: sys components and their interactions are documented
  - Object-Oriented model:
    » Inheritance model of the system
    » Interactions between objects
Programming and Debugging Process

- Programming: translating a design into a program
- Debugging: locating & correcting defects

- Programming is a personal activity - there is no generic programming process
- Programmers carry out some program testing to discover faults in the program and remove these faults in the debugging process
- Debugging process is part of both s/w development (as above by programmers) but s/w testing is done by testers
The debugging process

1. Defects/errors existence?
   - Yes: Other errors?
     - Yes: Other errors?
     - No: Debugged Program
   - No: Test by programmers

2. Test by programmers
   - Locate error
   - Design error repair
   - Repair error
   - Re-test program
Software Validation/Verification (Software V/V)

- Validation: Are we building the **right** product (satisfying client requirements)
- Verification: Are we building the product **right** (standards of the development process)

- Verification and validation is intended to show that a system conforms to its specification and meets the requirements of the system customer
- Involves checking and **review processes and system testing**
- Test Cases/Test Scenarios:
  - System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system
Testing = Verification + Validation

- Testing = Verification + Validation

- Verification: Document-based testing, Static Testing (no run)

- Validation: Dynamic Testing (Run code)
Defect Distribution in SW Life Cycle

Defect Distribution in Software Processes "SW Life Cycle"

- Requirements: 56%
- Design: 27%
- Code: 7%
- Others: 10%

Pie chart showing the distribution of defects across different stages of the software life cycle.
Verification: (standards of the development process)

- IEEE/ANSI: Determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

- Verification: document-based testing, static testing

- Evaluating, reviewing, inspecting, and doing desk checks of the work produced during development such as:
  - Requirements specifications: Do we have high quality requirements “clear, complete, measurable,...)
  - Design specifications
  - Code: Static analysis of code (code review) not dynamic execution of the code
Validation

- IEEE/ANSI: System evaluation during or at the end of the development process to determine whether it satisfies the specified requirements.

- Validation:
  - Run actual software on a computer.
  - Computer-based testing, Dynamic testing
V & V

- V&V are complementary
- Each provides filters to catch different kinds of problems
The System Testing Process

- Unit testing
- Module testing
- Sub-system testing
- System testing
- Acceptance testing

Programmer’s responsibility

Component testing
Integration testing
User testing

Service
Client Sign-off
OK
No

Programmer's responsibility
Testing stages

- **Unit testing**
  - Individual components are tested

- **Module testing**
  - Related collections of dependent components are tested

- **Sub-system testing**
  - Modules are integrated into sub-systems and tested. The focus here should be on **interface testing**

- **System testing/Integration Testing**
  - Testing functionality of the integrated system as a whole
  - Testing of **emergent properties**

- **Acceptance testing**
  - Testing with customer data (real, not simulated data) to check that the is acceptable
Alpha & Beta Testing

- **Alpha Testing**
  - For bespoke systems developed for a particular client
  - Inside the software house

- **Beta Testing**
  - For systems to be marketed as s/w products (COTS)
  - Is done by a number of potential customers & developers
  - Is done externally outside the s/w house development environment
Testing phases

Test plans (scenarios) are the link between development & testing
Software evolution (Maintenance)

- Generally less challenging than s/w development

- S/w must be designed to respond to Dynamic changes in Business Environment
  - As requirements change through changing business environment, the software that supports the business must also evolve and change

- Maintenance/changes:
  » Adaptive
  » Corrective
  » Perfective
System evolution (Maintenance)

- Define system requirements
- Assess existing systems
- Propose system changes
- Modify systems

Business Dynamic Environment

Maintenance of existing system
CASE:

Computer Aided Software Engineering

- **CASE tools**
  - Software tools to support software development and evolution processes

- **CASE TYPES**
  - Upper CASE: Supports early phases of s/w development (Analysis & Design)
  - Lower CASE: Supports implementation, code generation, testing (test cases generation, etc)

- **Activity automation**
  - Graphical editors for system model development
  - Data dictionary to manage design entities
  - Graphical UI builder for user interface construction
  - Debuggers to support program fault finding
  - Automated translators to generate new versions of a program
CASE technology

- Case technology
  - Led to significant improvements in the software process
  - Magnitude improvements were not as predicted
  - Software engineering requires creative thought - this is not readily automatable
  - Software engineering is a team activity and, for large projects, much time is spent in team interactions. CASE technology does not really support these
CASE classification

- Classification helps us understand the different types of CASE tools and their support for process activities
- Functional perspective
  - Tools are classified according to their specific function
- S/W process perspective / S/W Activity based
  - Tools are classified according to s/w process activities that are supported
- Integration perspective
  - Tools are classified according to their organisation into integrated units
## Functional tool classification

<table>
<thead>
<tr>
<th><strong>Tool type</strong></th>
<th><strong>Examples</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning tools</td>
<td>PERT tools, estimation tools, spreadsheets</td>
</tr>
<tr>
<td>Editing tools</td>
<td>Text editors, diagram editors, word processors</td>
</tr>
<tr>
<td>Change management tools</td>
<td>Requirements traceability tools, change control systems</td>
</tr>
<tr>
<td>Configuration management tools</td>
<td>Version management systems, system building tools</td>
</tr>
<tr>
<td>Prototyping tools</td>
<td>Very high-level languages, user interface generators</td>
</tr>
<tr>
<td>Method-support tools</td>
<td>Design editors, data dictionaries, code generators</td>
</tr>
<tr>
<td>Language-processing tools</td>
<td>Compilers, interpreters</td>
</tr>
<tr>
<td>Program analysis tools</td>
<td>Cross reference generators, static analysers, dynamic analysers</td>
</tr>
<tr>
<td>Testing tools</td>
<td>Test data generators, file comparators</td>
</tr>
<tr>
<td>Debugging tools</td>
<td>Interactive debugging systems</td>
</tr>
<tr>
<td>Documentation tools</td>
<td>Page layout programs, image editors</td>
</tr>
<tr>
<td>Re-engineering tools</td>
<td>Cross-reference systems, program restructuring systems</td>
</tr>
</tbody>
</table>
## Activity-based classification

<table>
<thead>
<tr>
<th>Tools</th>
<th>Specification</th>
<th>Design</th>
<th>Implementation</th>
<th>Verification and Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reengineering tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debugging tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program analysis tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language-processing tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method support tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prototyping tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration management tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change management tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Documentation tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Editing tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Planning tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CASE integration

- **Tools**
  - Support individual process tasks such as design consistency checking, text editing, etc.

- **Workbenches**
  - Support a process phase such as specification or design,
    Normally include a number of integrated tools

- **Environments**
  - Support all or a substantial part of an entire software process.
    Normally include several integrated workbenches
Tools, workbenches, environments

CASE technology

- Tools
  - Editors
  - Compilers

- Workbenches
  - File comparators
  - Integrated environments
  - Process-centred environments

- Environments
  - Analysis and design
  - Programming
  - Testing
    - Multi-method workbenches
    - Single-method workbenches
    - General-purpose workbenches
    - Language-specific workbenches