Development Process and Analysis
Software Crisis

- Declared in the late 60’s
- Expressed by delays and failures of major software projects (unreached goals, unpredictable costs and unmaintainable software)
- Was obviously driven by uncoordinated and unstructured programming ("hacking")
- Lead to a new research and engineering discipline: **Software Engineering**
Program Complexity

A Program

x 3

An Application System

x 3

A Program Product

x 3

An Application Systems Product

x 3

complexity x 9
A Program

- developed by a single programmer
- complete in itself
- *one* customer & *one* user: the author
- operational only on the author’s system for which it was developed

Example: Your address manager written in VisualBasic.
An Application System

- contains many programs for different tasks
- components coordinated in function
- programs disciplined in format
- well-defined interfaces between components
- one customer with many users

Example: Information system for all departments (stock, accounts, management, ...) of firm X
A Program Product

- developed by a developer team
- thoroughly tested and well documented
- *many* single customers (= users)
- specialized to one task
- available for different environments

Example: Microsoft PowerPoint
An Application Systems Product

- combines the attributes of
  - program (complete in itself, many functions)
  - application system
    - coordinated components
    - *many* users
  - program product
    - developed by a team
    - thoroughly tested
    - usable on different platforms
    - *many* customers

Example: SAP R/3
Characteristics of Making Software

- Software is an *immaterial product*.
- Software exhibits *no material aging process*.
- Software *does not suffer from spare parts*.
- Software is rather easily changed.
- Software has to be adapted to changes of requirements or environments.
- Software is the *result of product development*, not of a traditional production process.
Software Product Attributes

- Attributes of well-engineered software:
  - **Maintainability** (possibility of evolve)
  - **Dependability** (reliable systems)
  - **Efficiency** (smart resource handling)
  - **Usability** (appropriate user interface)
Goals and Tasks of Software Development

Main Goals
- Product related:
  - Usability
  - Productivity
  - Quality
- Process related:
  - Schedules and costs

Main Tasks
- Analysis
- Design
- Implementation
- Test
- Introduction
- Maintenance

Mission:
Delivering a product
- that is useful and used
- at the predicted costs
- in time
Requirements to Software Production

- Product has to **meet the specification.**
  - Develop in cycles.
  - Use and evaluate early prototypes.

- Product has to be **produced in time.**
  - Apply project management and project organization.
  - Plan the installation of the software system and the education of users.

- Product must **stay within the estimated costs.**
  - Use “of-the-shelf” components as far as possible (“buy” instead of “make”).
  - Apply standards.
Abstraction Levels

Requirements Analysis: Why?
- requirements
- domain knowledge
- goals

System Design: What?
- abstractions
- models
- structures
- architecture

Software Implementation: How?
- algorithms
- data structures
- generic services
- platform-specific services
The Waterfall Process

Requirements definition, analysis → System and software design → Implementation and unit testing → Integration and system testing → Operation and maintenance

The development phases are performed **sequentially**...

...but there are **backward loops** in case of changing requirements, error corrections, etc.

[Ian Sommerville; Software Engineering, Addison Wesley, 1982]
Evolutionary Development

Concurrent activities

Specifications
Development
Validation

Prototypes
Initial version
Intermediate versions
Final version

Outline description

[Ian Sommerville; Software Engineering, Addison Wesley, 1982]
Boehm’s Spiral Model

[Ian Sommerville; Software Engineering, Addison Wesley, 1982]
Unified Process

- **Inception**: establish business rationale and decide on scope of the project.

- **Elaboration**: collect detailed requirements, high-level analysis and design. Establish baseline architecture and plan for construction.

- **Construction**: iterative and incremental process. Each iteration builds production-quality software prototypes which implement subsets of requirements.

- **Transition** contains beta testing, performance tuning and user training.
First Step: Inception

- Inception can take many forms:
  - Sometimes just a chat at the coffee machine ...  
  - ... or a full-fledged feasibility study.
- During the inception phase you work out the business case for the project:
  - Calculate how much the project will cost.
  - Estimate how much profit it will bring in.
- Some initial analysis is required to get a sense of the project’s scope and size.
- Inception should be a few days of work to consider if it is worth doing a few months of work during elaboration.
- At the point of inception the project sponsor agrees to no more than a serious look at the project:

Do we go ahead with the project?
Second Step: Elaboration

- Starts after the “go-ahead” agreement.
- At this stage you have only vague requirements:
  “We are going to build the next generation customer support system for the Watts Galore Utility Company. We intend to use object-oriented technology to build a more flexible system that is more customer oriented - specifically, one that will support consolidated customer bills”.
- Elaboration is the point where you want a better understanding of the problem:
  - **What** is it you are actually going to build?
  - **How** are you going to build it?
  - What **technology** are you going to use?
- Elaboration includes to have a careful and thorough look at the possible risks in your project:

  What are the things that could derail you?
Elaboration: System Analysis

Rationale:
Finding and fixing a fault after software delivery is 100 times more expensive than finding and fixing it during systems analysis or early design phases.

- **Goal** of analysis is to develop a model of what the system will do.
- The analysis phase should include information required to understand what the software should do for a real world system.
- The client of a system should understand the analysis model.
- The analysis phase delivers a base from which further details are derived in the design phase.
- Analysis provides the requirements and the real-world environment in which the software system will exist.

Object-oriented analysis forces a seamless development process with no discontinuities because of continuous refinement and progressing from analysis through design to implementation.
Analysis: Actors, Steps, Deliverables

- Customer requirements
- Developer needs
- Manager priorities

Problem Analysis

Problem statement

Represented by
- Texts
- Spreadsheets
- Diagrams
...
Design: Actors, Steps, Deliverables

- Domain Knowledge
- User Interviews
- Real-world experience

Represented by
- Problem statement

Model Design

e.g. UML diagrams for
- Use Cases
- Classes
- Interactions
- Packages
- States
- Activities ...
Analysis: Capturing Requirements

**Identify** typical **use cases** of the system you are going to build.

- A **typical use case** for using a database:
  - “list all customers who have ordered a certain product”
  - “create a list with my top 10 customers”
  - “I want fax-letters to be sent automatically”

- A **developer** responds with specific **cost estimates**:
  - “The top 10 customer list can be developed in a week.”
  - “Creating the auto-fax function will take two months.”

- User and developer **negotiate** about the **priorities**.

- UML notation available for use cases (later today).
Elaboration: Planning

Schedule use cases to specific iterations and dates of delivery.

- Customers assign priorities to the use cases.
- Developers consider the architectural risk.
  - Concentrate to the use cases which are technologically most challenging.
- Developers need to be aware of the schedule risks.
- Estimate the length of each iteration
  - Be aware that you need to: analyze, design, code, write unit tests, integrate, and document.

The estimates should be done by the developers, not by the managers.
UP: Incremental Iterations

<table>
<thead>
<tr>
<th>Transition</th>
<th>Elaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>project</td>
<td>requirements</td>
</tr>
<tr>
<td>deliverables</td>
<td>binding contracts</td>
</tr>
</tbody>
</table>

| Analysis        | Coding               |
| Design          | Debugging            |
|                 | Construction         |
|                 | executable modules   |

<table>
<thead>
<tr>
<th>Elaboration</th>
<th>Construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>requirements</td>
<td>executable modules</td>
</tr>
<tr>
<td>binding contracts</td>
<td></td>
</tr>
</tbody>
</table>
UP: Iterations

Construction builds the system in a series of iterations. Each iteration is a project in itself.

During each iteration you go through a cycle of analyzing, designing, coding, debugging, integration and demonstration of the implemented use case by a prototype.
Modelling Process vs. Modelling Language

- **Modeling Process (e.g. “Unified Process”)**
  - Methodology used to describe the different stages during analysis and design of complex software
  - Recommended proceeding to get from one development phase to the next

- **Modeling Language (e.g. UML)**
  - Notation that visualizes requirements and results for each stage of a modeling process, e.g.
    - requirements of the customer
    - design decisions of the developer
    - properties and (expected) behaviour of the software
The Unified Modeling Language

- UML offers a model notation
  - Not a methodology for software development.
  - UML is used by the Unified Software Development Process (UP).
- UML aims to be intuitively understandable
  - You can use many parts when talking to non-software people.
- Many CASE tools available that support UML
UML Diagrams

- **Class Diagrams**
  - intra-class behavior
  - structuring
  - messages leading to state changes within objects

- **Interaction Diagrams**
  - interaction sequences between objects
  - structures

- **State Diagrams**
  - scenarios
  - workflows

- **Activity Diagrams**
  - workflows
  - structures
  - inter-class behavior

- **Use Case Diagrams**
  - scenarios
  - interaction sequences between objects

- **Package Diagrams**
  - structuring
  - intra-class behavior
Use Cases

- A **use case** is the specification of a sequence of interactions between an actor and the computer system to be developed.
- Use cases describe *user-visible* functionality.
- An **instance** of a use case is the execution of the described interactions.

A use case is defined by:

**Name:** Verb (or noun), name of the procedure  
**Priority:** Priority in development process  
**Description:** textual description with reference to the requirements specification or a glossary  
(possibly other things depending on specific situation)
Use Case Diagrams

- A **use case diagram** shows the relationship between actors and use cases in a system.
- A use case diagram for a system consists of:
  - the system boundary
  - actors
  - use cases
  - relationships
System Boundary

- The use case view describes the functionality of a system from an outside point of view, that means from the user’s point of view.
- Only the interactions between actors (users) and system are shown, what happens inside the system is hidden.
- This boundary is clarified by the system boundary.

Observation: Systems can also appear as actors

Entry system

Change price

Employee
Actors

- An **Actor** is an abstraction for an entity outside the system interacting with the system, e.g., a user.

  This enables:
  - several roles for one user
  - several users with the same role
  - systems as actors
Relationships

- **Relationships** between use cases and actors
- The **communication** between an actor and the system is represented by an edge between actor and use case.
- An actor can be associated with several use cases and vice versa.
Inclusion of Use Cases

- **Include** means the inclusion of particular action sequences in the base use case.
- The included use case can be used independent of the base use cases.

![Diagram](image-url)

- **Stereotype**
- **Base use case**
- **Included use case**
- **Inclusion**
Generalization of Use Cases

- **Generalization** correlates (taxonomically) more specialized use cases and more general ones. The special case inherits **every** property from the general and adds incrementally further properties or replaces them.

- Special use cases can substitute general ones.
Extension of Use Cases

- **Extensions** define variations and special cases with the meaning that the extending case completes the base use case, i.e., may be inserted in the behavior of the base case.
- Extensions are included, maybe optionally, at **extension points**.
- A use case may have many extension points.

![Diagram of Extension of Use Cases]

- Stereotype
- Account travel
  - extension points
  - Accounting
- «extend»
  - (Accounting)
- Extending case
  - Accounting per waiting list
- Extension
  - Base use case (extended case)
Generalization of Actors

- Actors may have a **taxonomic relationship**.
- In use case diagrams the taxonomy is usually not shown. Separate actor diagrams show the relationships between actors.

![Diagram showing generalization of actors](image)

- **generalized actor**: Customer
- **specialized actors**: Tourist, Business customer
References

- Online:
  - UML Notation Guide -
    http://etna.int-evry.fr/COURS/UML/notation/

- Books:
  - Martin Fowler. **UML Distilled**. Addison-Wesley
    (TUHH library: TIF-587)
  - Jacobson, Booch, Rumbaugh. **The Unified Software Development Process**. Addison-Wesley
    (TUHH library: TIF-624)