

Software system engineering (2IW60)

Prof.dr.ir. Jan Friso Grootte
Prof.dr. Mark van den Brand



Practical issues

- Lectures:
 - Prof.dr.ir. Jan Friso Grootte (j.f.grootte@tue.nl), HG6.75
 - Prof.dr. Mark van den Brand (m.g.i.v.d.brand@tue.nl), HG5.59
- More information on my part (block A) of SSE:
 - <http://www.win.tue.nl/~mvdbrand/courses/sse/0809/>
 - slides
 - papers
 - Software Engineering books (not required):
 - Software Engineering by H. van Vliet, 3rd ed., Wiley
 - Object-Oriented Software Engineering by T.C. Lethbridge and R. Laganière, 2nd ed., McGraw-Hill
 - Software Engineering by R.S. Pressman, 6th ed., McGraw-Hill

Practical issues

- Examination:
 - Written exam on topics of block A and B:
 - Material to study:
 - slides
 - papers
 - Paper reviewing in block A
 - Practical exercise in block B

Software Engineering topics

- Why software engineering?
- Requirements engineering
- Software design
- Software architecture
- Testing

Why Software Engineering?

- The nature of software ...
 - Software is everywhere
 - Dependable
 - Robust
 - Software is intangible
 - Hard to understand development effort
 - Software is easy to reproduce
 - Cost is in its *development*
 - in other engineering products, manufacturing is the costly stage
 - The industry is labor-intensive
 - Hard to automate

Why Software Engineering?

- The nature of software ...
 - Untrained people can hack something together
 - Quality problems are hard to notice
 - Software is easy to modify
 - People make changes without fully understanding it
 - Software does not 'wear out'
 - It *deteriorates* by having its design changed:
 - erroneously, or
 - in ways that were not anticipated, thus making it complex

Why Software Engineering?

- The nature of software ...
 - Conclusions
 - Much software has poor design and is getting worse
 - Demand for software is high and rising
 - We are in an ever lasting 'software crisis'
 - We have to learn to 'engineer' software

Why Software Engineering?

- Types of software ...
 - Custom
 - For a specific customer
 - Generic
 - Sold on open market
 - Often called
 - COTS (Commercial Off The Shelf)
 - Shrink-wrapped
 - Embedded
 - Built into hardware
 - Hard to change

What is Software Engineering?

- The process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints
- Other definition:
 - IEEE: (1) the application of a systematic, disciplined, quantifiable approach to the development, operation, maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

What is Software Engineering?

- Solving customers' problems
 - This is the *goal* of software engineering
 - Sometimes the solution is to *buy, not build*
 - Adding unnecessary features does not help solve the problem
 - **Software engineers must *communicate effectively* to identify and understand the problem**

What is Software Engineering?

- Systematic development and evolution
 - An engineering process involves applying *well understood techniques* in an organized and *disciplined* way
 - Many well-accepted practices have been formally standardized
 - e.g. by the IEEE or ISO
 - Most development work is *evolution*

What is Software Engineering?

- Large, high quality software systems
 - Software engineering techniques are needed because large systems *cannot be completely understood* by one person
 - Identification of missing quality aspects before building
 - Teamwork and co-ordination are required
 - Key challenge: Dividing up the work and ensuring that the parts of the system work properly together
 - The end-product must be of high quality

What is Software Engineering?

- Cost, time and other constraints
 - Finite resources
 - The benefit must outweigh the cost
 - Others are competing to do the job cheaper and faster
 - **Inaccurate estimates of cost and time have caused many project failures**
- Quality attributes:
 - Usability, efficiency, reliability, maintainability, reusability
 - The different qualities can conflict
 - Increasing efficiency can reduce maintainability
 - Increasing usability can reduce efficiency

Activities Common to Software Projects

- Requirements and specification
 - Includes
 - Domain analysis
 - Defining the problem
 - Requirements gathering
 - Obtaining input from as many sources as possible
 - Requirements analysis
 - Organizing the information
 - Requirements specification
 - Writing detailed instructions about how the software should behave

Activities Common to Software Projects

- Design
 - Deciding how the requirements should be implemented, using the available technology
 - Includes:
 - *Systems engineering*: Deciding what should be in hardware and what in software
 - *Software architecture*: Dividing the system into subsystems and deciding how the subsystems will interact
 - *Detailed design* of the internals of a subsystem
 - *User interface design* (probably less important)
 - *Design of databases* (probably less important)

Activities Common to Software Projects

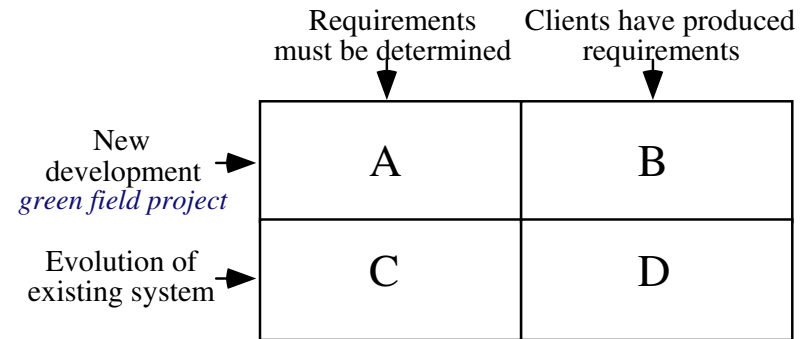
- Modeling
 - Creating representations of the domain or the software
 - Use case modeling
 - Structural modeling
 - Dynamic and behavioral modeling
- Programming
- Quality assurance
 - Reviews and inspections
 - Testing
- Deployment
- Managing the process

Requirements Engineering

- **Domain analysis**
 - The process by which a software engineer learns about the domain to better understand the problem:
 - The *domain* is the general field of business or technology in which the clients will use the software
 - **A domain expert is a person who has a deep knowledge of the domain**
 - Benefits of performing domain analysis:
 - Faster development
 - Better system
 - Anticipation of extensions

Requirements Engineering

Starting point for software projects



Requirements Engineering

- **Problem and scope:**
 - A problem can be expressed as:
 - A *difficulty* the users or customers are facing,
 - Or as an *opportunity* that will result in some benefit such as improved productivity or sales.
 - The solution to the problem normally will entail developing software
 - A good problem statement is short and succinct
- **Narrow the scope** by defining a more precise problem
 - Exclude some of these things if too broad
 - Determine high-level goals if too narrow

What is a Requirement ?

- It is a statement describing either
 - 1) an aspect of what the proposed system must do,
 - or 2) a constraint on the system's development.
 - In either case it must contribute in some way towards adequately solving the customer's problem;
 - the set of requirements as a whole represents a negotiated agreement among the stakeholders.
- A collection of requirements is a *requirements document*.

Types of Requirements

- **Functional requirements**
 - Describe *what* the system should do
- **Quality requirements**
 - *Constraints* on the design to meet specified levels of quality
- **Platform requirements**
 - *Constraints* on the environment and technology of the system
- **Process requirements**
 - *Constraints* on the project plan and development methods

Functional Requirements

- What *inputs* the system should accept
- What *outputs* the system should produce
- What data the system should *store* that other systems might use
- What *computations* the system should perform
- The *timing and synchronization* of the above

- Use cases very suited to describe functional requirements

Quality Requirements

- All must be verifiable
- **Examples: constraints on**
 - Response time
 - Throughput
 - Resource usage
 - Reliability
 - Availability
 - Recovery from failure
 - Allowances for maintainability and enhancement
 - Allowances for reusability

Use cases: describing how the user will use the system

- A *use case* is a typical sequence of actions that a user performs in order to complete a given task
- The objective of *use case analysis* is to model the system from the point of view of
 - ... how users interact with this system
 - ... when trying to achieve their objectives.It is one of the key activities in requirements analysis
- A *use case model* consists of
 - a set of use cases
 - an optional description or diagram indicating how they are related

Use cases

- A use case should
 - Cover the *full sequence of steps* from the beginning of a task until the end.
 - Describe the *user's interaction* with the system ...
 - Not the computations the system performs.
 - Be written so as to be as *independent* as possible from any particular user interface design.
 - Only include actions in which the actor interacts with the computer.
 - Not actions a user does manually

The modeling processes: Choose use cases on which to focus

- Often one use case (or a very small number) can be identified as *central* to the system
 - The entire system can be built around this particular use case
- There are other reasons for focusing on particular use cases:
 - Some use cases will represent a high *risk* because for some reason their implementation is problematic
 - Some use cases will have high political or commercial value

The benefits of basing software development on use cases

- They can
 - Help to define the *scope* of the system
 - Be used to *plan* the development process
 - Be used to both develop and validate the requirements
 - Form the basis for the definition of test cases
 - Be used to structure user manuals

Use cases

- Use cases have shortcomings:
 - The use cases themselves must be validated
 - Using the requirements validation methods.
 - Some aspects of software are not covered by use case analysis.
 - Innovative solutions may not be considered.

Some Techniques for Gathering and Analysing Requirements

- **Observation**
 - Read documents and discuss requirements with users
 - Shadowing important potential users as they do their work
 - Session videotaping
- **Interviewing**
 - Conduct a series of interviews
 - Ask about specific details
 - Ask about the stakeholder's vision for the future
 - Ask if they have alternative ideas
 - Ask for other sources of information
 - Ask them to draw diagrams
- **Brainstorming**

Gathering and Analysing Requirements

- **Prototyping**
 - The simplest kind: *paper prototype*.
 - a set of pictures of the system that are shown to users in sequence to explain what would happen
 - The most common: a mock-up of the system's UI
 - Written in a rapid prototyping language
 - Does *not* normally perform any computations, access any databases or interact with any other systems
 - May prototype a particular aspect of the system

Gathering and Analysing Requirements

- **Use case analysis**
 - Determine the classes of users that will use the facilities of this system (actors)
 - Determine the tasks that each actor will need to do with the system

Level of detail required in a requirements document

- How much detail should be provided depends on:
 - The size of the system
 - The need to interface to other systems
 - The readership
 - The stage in requirements gathering
 - The level of experience with the domain and the technology
 - The cost that would be incurred if the requirements were faulty

Reviewing Requirements

- Each individual requirement should
 - **Have** benefits that outweigh the costs of **development**
 - **Be** important **for the solution of the current problem**
 - **Be expressed using a** clear and consistent notation
 - **Be** unambiguous
 - **Be** logically consistent
 - **Lead to a system of** sufficient quality
 - **Be** realistic **with available resources**
 - **Be** verifiable
 - **Be uniquely** identifiable
 - Does not over-constrain the design **of the system**

Requirements Review Checklist

1. Does each user requirement have a **unique identifier**?
2. Is each user requirement **atomic** and **simply formulated**? (Single sentence. Composite requirements can best be split.)
3. Are user requirements organized into **coherent groups**? (If necessary, hierarchical; not more than about ten per group.)
4. Is each user requirement **verifiable** (in a provisional acceptance test)? (Where possible, quantify.)
5. Is each user requirement **prioritized**?
6. Are all **unstable** user requirements marked as such? (TBC=`To Be Confirmed`)

Requirements Review Checklist (cont.)

7. Are all user requirements **necessary**?
8. Are the user requirements **complete**? Can everything not explicitly constrained indeed be viewed as developer freedom? Is a product that satisfies every requirement indeed acceptable? (No requirements missing.)
9. Are the user requirements **consistent**? (Non-conflicting.)
10. Are the user requirements sufficiently **precise** and **unambiguous**? (Which interfaces are involved, who has the initiative, who supplies what data; avoid passive voice.)
11. Are the user requirements **understandable** to those who will need to work with them later?
12. Are the user requirements **realizable** within budget?

Requirements Review Checklist (cont.)

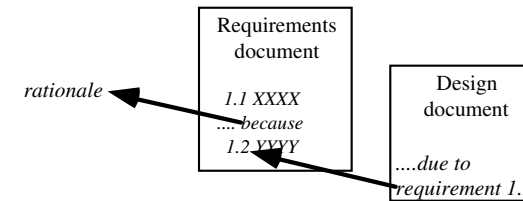
13. Do the user requirements express actual **customer needs** (in the language of the problem domain), **rather than solutions** (in developer jargon)?
14. Are the user requirements not **overly restrictive**? (Allow enough developer freedom.)
15. Is **overlap** among user requirements pointed out explicitly? (Redundancy; cross-reference.)
16. Are **dependencies** between user requirements pointed out explicitly? Are these consistent with the priorities?
17. Are the characteristics of **users** and of **typical usage** described in sufficient detail? (No user categories missing.)

Requirements Review Checklist (cont.)

- 18. Is the **operational environment** described in sufficient detail, including e.g. relevant data flows in the "outside" world that do not interact directly with the system? (No external interfaces missing. No capabilities missing. Diagram included.)

Requirements documents

- **The document should be:**
 - **sufficiently complete**
 - **well organized**
 - **clear**
 - **agreed to by all the stakeholders**
- **Traceability:**



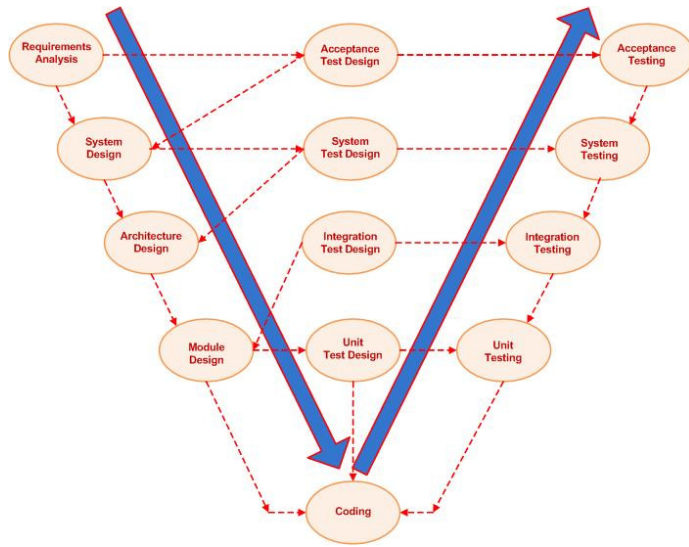
Managing Changing Requirements

- **Requirements change because:**
 - **Business process changes**
 - **Technology changes**
 - **The problem becomes better understood**

Managing Changing Requirements

- **Requirements analysis never stops**
 - **Continue to interact with the clients and users**
 - **The benefits of changes must outweigh the costs.**
 - **Certain small changes (e.g. look and feel of the UI) are usually quick and easy to make at relatively little cost.**
 - **Larger-scale changes have to be carefully assessed**
 - **Forcing unexpected changes into a partially built system will probably result in a poor design and late delivery**
- **Some changes are enhancements in disguise**
 - **Avoid making the system *bigger*, only make it *better***

V model for software development



The Process of Design

- **Definition:**
 - *Design* is a problem-solving process whose objective is to find and describe a way:
 - To implement the system's *functional requirements*...
 - While respecting the constraints imposed by the *quality, platform and process requirements*...
 - including the budget
 - And while adhering to general principles of *good quality*

Design as a series of decisions

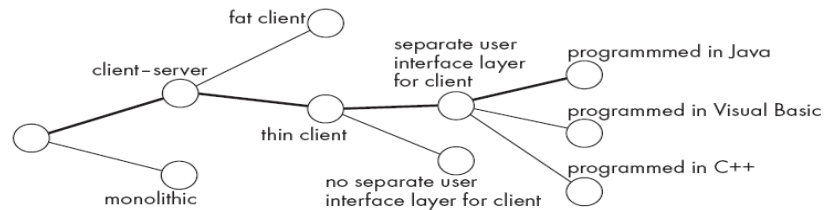
- A designer is faced with a series of *design issues*
 - These are sub-problems of the overall design problem
 - Each issue normally has several alternative solutions:
 - *design options*
 - The designer makes a *design decision* to resolve each issue
 - This process involves choosing the best option from the alternatives

Making decisions

- To make each design decision, the software engineer **uses:**
 - Knowledge of
 - the requirements
 - the design as created so far
 - the technology available
 - software design principles and 'best practices'
 - what has worked well in the past

Design space

- The space of possible designs that could be achieved by choosing different sets of alternatives is often called the *design space*
- For example:



From requirements to implementation

- **System:**
 - A logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software or both.
 - A system can have a specification which is then implemented by a collection of components.
 - A system continues to exist, even if its components are changed or replaced.
 - The goal of requirements analysis is to determine the responsibilities of a system.
- **Subsystem:**
 - A system that is part of a larger system, and which has a definite interface

From requirements to implementation

- **Component:**
 - Any piece of software or hardware that has a clear role
 - A component can be isolated, allowing you to replace it with a different component that has equivalent functionality
 - Many components are designed to be reusable
 - Conversely, others perform special-purpose functions

From requirements to implementation

- **Module:**
 - A component that is defined at the programming language level
 - For example: classes and packages are modules in Java
- **Function:**
 - Unit at programming level with specific behaviour
 - For example: methods in Java, functions in C

Top-down and bottom-up design

- **Top-down design**
 - First design the very high level structure of the system.
 - Then gradually work down to detailed decisions about low-level constructs.
 - Finally arrive at detailed decisions such as:
 - the format of particular data items;
 - the individual algorithms that will be used.

Top-down and bottom-up design

- **Bottom-up design**
 - Make decisions about reusable low-level utilities.
 - Then decide how these will be put together to create high-level constructs.
- **A mix of top-down and bottom-up approaches are normally used:**
 - Top-down design is almost always needed to give the system a good structure.
 - Bottom-up design is normally useful so that reusable components can be created.

Different aspects of design

- **Architecture design:**
 - The division into subsystems and components,
 - How these will be connected.
 - How they will interact.
 - Their interfaces.
- **Class design:**
 - The various features of classes.
- **User interface design**
- **Algorithm design:**
 - The design of computational mechanisms.
- **Protocol design:**
 - The design of communications protocol.

Principles Leading to Good Design

- **Overall goals of good design:**
 - Increasing profit by reducing cost and increasing revenue
 - Ensuring that we actually conform with the requirements
 - Accelerating development
 - Increasing qualities such as
 - Usability
 - Efficiency
 - Reliability
 - Maintainability
 - Reusability

Design Principle 1

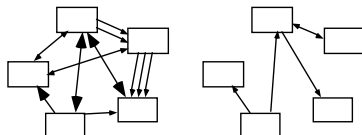
- **Divide and conquer**
 - Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things
 - Separate people can work on each part.
 - An individual software engineer can specialize.
 - Each individual component is smaller, and therefore easier to understand.
 - Parts can be replaced or changed without having to replace or extensively change other parts.

Design Principle 2

- **Increase cohesion where possible:**
 - A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things
 - This makes the system as a whole easier to understand and change
 - Type of cohesion:
 - Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

Design Principle 3

- **Reduce coupling where possible:**
 - *Coupling* occurs when there are *interdependencies* between one module and another
 - When interdependencies exist, changes in one place will require changes somewhere else.
 - A network of interdependencies makes it hard to see at a glance how some component works.
 - Type of coupling:
 - Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External



Design Principle 4

- **Keep the level of abstraction as high as possible**
 - Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity
 - A good abstraction is said to provide *information hiding*
 - Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details

Design Principle 5

- Increase reusability where possible
- Design the various aspects of your system so that they can be used again in other contexts
 - Generalize your design as much as possible
 - Follow the preceding three design principles
 - Design your system to contain hooks
 - Simplify your design as much as possible

Design Principle 6

- Reuse existing designs and code where possible
- Design with reuse is complementary to design for reusability
 - Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
 - *Cloning* should not be seen as a form of reuse

Design Principle 7

- Design for flexibility
- Actively anticipate changes that a design may have to undergo in the future, and prepare for them
 - Reduce coupling and increase cohesion
 - Create abstractions
 - Do not hard-code anything
 - Leave all options open
 - Do not restrict the options of people who have to modify the system later
 - Use reusable code and make code reusable

Design Principle 8

- Anticipate obsolescence
- Plan for changes in the technology or environment so the software will continue to run or can be easily changed
 - Avoid using early releases of technology
 - Avoid using software libraries that are specific to particular environments
 - Avoid using undocumented features or little-used features of software libraries
 - Avoid using software or special hardware from companies that are less likely to provide long-term support
 - Use standard languages and technologies that are supported by multiple vendors

Design Principle 9: Design for Portability

- Design for Portability
 - Have the software run on as many platforms as possible
 - Avoid the use of facilities that are specific to one particular environment
 - E.g. a library only available in Microsoft Windows

Design Principle 10

- Design for Testability
 - Take steps to make testing easier
 - Design a program to automatically test the software
 - Ensure that all the functionality of the code can be driven by an external program, bypassing a graphical user interface
 - In Java, you can create a main() method in each class in order to exercise the other methods

Design Principle 11

- Design defensively
 - Never trust how others will try to use a component you are designing
 - Handle all cases where other code might attempt to use your component inappropriately
 - Check that all of the inputs to your component are valid: the *preconditions*
 - Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking

Design by contract

- A technique that allows you to design defensively in an efficient and systematic way
 - Key idea
 - each method has an explicit *contract* with its callers
 - The contract has a set of assertions that state:
 - What *preconditions* the called method requires to be true when it starts executing
 - What *postconditions* the called method agrees to ensure are true when it finishes executing
 - What *invariants* the called method agrees will not change as it executes

Principles Leading to Good Design

- Overall *goals* of good design:
 - Increasing profit by reducing cost and increasing revenue
 - Ensuring that we actually conform with the requirements
 - Accelerating development
 - Increasing qualities such as
 - Usability
 - Efficiency
 - Reliability
 - Maintainability
 - Reusability

Techniques for making design decisions

- Using priorities and objectives to decide among alternatives
 1. List and describe the alternatives for the design decision.
 2. List the advantages and disadvantages of each alternative with respect to your objectives and priorities.
 3. Determine whether any of the alternatives prevents you from meeting one or more of the objectives.
 4. Choose the alternative that helps you to best meet your objectives.
 5. Adjust priorities for subsequent decision making.

Using cost-benefit analysis to choose among alternatives

- To estimate the *costs*, add up:
 - The incremental cost of doing the *software engineering* work, including ongoing maintenance
 - The incremental costs of any *development technology* required
 - The incremental costs that *end-users and product support personnel* will experience
- To estimate the *benefits*, add up:
 - The incremental software engineering time saved
 - The incremental benefits measured in terms of either increased sales or else financial benefit to users

Writing a Good Design Document

- Design documents as an aid to making better designs
 - They force you to be explicit and consider the important issues before starting implementation.
 - They allow a group of people to review the design and therefore to improve it.
 - Design documents as a means of communication.
 - To those who will be *implementing* the design.
 - To those who will need, in the future, to *modify* the design.
 - To those who need to create systems or subsystems that *interface* with the system being designed.

When writing the document

- Avoid documenting information that would be *readily obvious* to a skilled programmer or designer.
- Avoid writing details in a design document that would be better placed as *comments* in the code.
- Avoid writing details that can be *extracted automatically* from the code, such as the list of public methods.

Software Architecture

- **Software architecture** is process of designing the global organization of a software system, including:
 - Dividing software into subsystems.
 - Deciding how these will interact.
 - Determining their interfaces.
 - The architecture is the core of the design, so all software engineers need to understand it.
 - The architecture will often constrain the overall efficiency, reusability and maintainability of the system.

Definition of Software Architecture

- **Definition:** The software architecture of a program or computing system is the structure or structures of the system, which compromise software elements, the externally visible properties of those elements, and relationships among them.
- **Quality criteria of software architectures:**
 - Availability
 - Modifiability
 - Performance
 - Security
 - Testability
 - Usability

The importance of software architecture

- **Why you need to develop an architectural model:**
 - To enable everyone to better understand the system
 - To allow people to work on individual pieces of the system in isolation
 - To prepare for extension of the system
 - To facilitate reuse and reusability

Contents of a good architectural model

- A system's architecture will often be expressed in terms of several different *views*
 - The logical breakdown into subsystems
 - The interfaces among the subsystems
 - The dynamics of the interaction among components at run time
 - The data that will be shared among the subsystems
 - The components that will exist at run time, and the machines or devices on which they will be located

Design *stable* architecture

- To ensure the maintainability and reliability of a system, an architectural model must be designed to be *stable*.
 - Being stable means that the new features can be easily added with only small changes to the architecture

Developing an architectural model

- Start by sketching an outline of the architecture
 - Based on the principal requirements and use cases
 - Determine the main components that will be needed
 - Choose among the various architectural patterns
 - Multi-Layer
 - Client-server
 - Pipe-and-filter
 - *Suggestion*: have several different teams independently develop a first draft of the architecture and merge together the best ideas

Developing an architectural model

- Refine the architecture
 - Identify the main ways in which the components will interact and the interfaces between them
 - Decide how each piece of data and functionality will be distributed among the various components
 - Determine if you can re-use an existing framework, or if you can build a framework
- Consider each use case and adjust the architecture to make it realizable
- Mature the architecture

AUTOSAR

- AUTomotive Open System ARchitecture is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers.
- Objective is to create and establish open standards for automotive E/E (Electrics/Electronics) architectures that will provide a basic infrastructure to assist with developing vehicular software, user interfaces and management for all application domains.

AUTOSAR

- paves the way for innovative electronic systems that further improve performance, safety and environmental friendliness
- is a key enabling technology to manage the growing electrics/electronics complexity.
- aims to be prepared for the upcoming technologies and to improve cost-efficiency without making any compromise with respect to quality
- facilitates the exchange and update of software and hardware over the service life of the vehicle

AUTOSAR

- follows the layered architecture concept. The total software solution is divided into 3 layers, known as
 - Application software,
 - Runtime Environment (RTE) and
 - Basic Software (BSW).
- All the functionalities are kept under the application software module, the Run Time Environment acts as a virtual function bus for an Electronic control unit and messages are transferred using the basic software.

Software testing

- Basic definitions
 - A *failure* is an unacceptable behaviour exhibited by a system
 - The frequency of failures measures the *reliability*
 - An important design objective is to achieve a very low failure rate and hence high reliability.
 - A failure can result from a violation of an *explicit* or *implicit* requirement

Software testing

- **Basic definitions**
 - A *defect* is a flaw in any aspect of the system that contributes, or may potentially contribute, to the occurrence of one or more failures
 - could be in the requirements, the design and the code
 - It might take several defects to cause a particular failure
 - An *error* is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect

Effective and Efficient Testing

- To test *effectively*, you must use a strategy that uncovers as many defects as possible.
- To test *efficiently*, you must find the largest possible number of defects using the fewest possible tests
 - Testing is like detective work:
 - The tester must try to understand how programmers and designers think, so as to better find defects.
 - The tester must not leave anything uncovered, and must be suspicious of everything.
 - It does not pay to take an excessive amount of time; tester has to be *efficient*.

Glass-box testing

- Also called ‘white-box’ or ‘structural’ testing
- Testers have access to the system design
 - They can
 - Examine the design documents
 - View the code
 - Observe at run time the steps taken by algorithms and their internal data
 - Individual programmers often informally employ glass-box testing to verify their own code

Flow graph for glass-box testing

- To help the programmer to systematically test the code
 - Each branch in the code (such as if and while statements) creates a node in the graph
 - The testing strategy has to reach a targeted coverage of statements and branches; the objective can be to:
 - cover all possible paths (often infeasible)
 - cover all possible edges (most efficient)
 - cover all possible nodes (simpler)

Black-box testing

- Testers provide the system with inputs and observe the outputs
 - They can see none of:
 - the source code
 - the internal data
 - any of the design documentation describing the system's internals

Detecting specific categories of defects

- A tester must try to uncover any defects the other software engineers might have introduced.
- This means designing tests that explicitly try to catch a range of specific types of defects that commonly occur.

Defects in Ordinary Algorithms

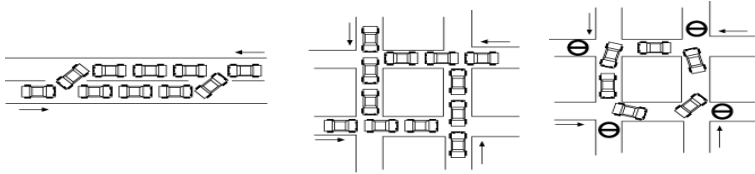
- Incorrect logical conditions
- Performing a calculation in the wrong part of a control construct
- Not terminating a loop or recursion
- Not setting up the correct preconditions for an algorithm
- Not handling null conditions
- Not handling singleton or non-singleton conditions
- Off-by-one errors
- Operator precedence errors
- Use of inappropriate standard algorithms

Defects in Numerical Algorithms

- Not using enough bits or digits
- Not using enough places after the decimal point or significant figures
- Ordering operations poorly so errors build up
- Assuming a floating point value will be exactly equal to some other value

Defects in Timing and Co-ordination

- **Deadlock and livelock**
 - **Defects:**
 - A deadlock is a situation where two or more threads are stopped, waiting for each other to do something.
 - The system is hung
 - Livelock is similar, but now the system can do some computations, but can never get out of some states.



Defects in Timing and Co-ordination

- **Deadlock and livelock**
 - **Testing strategies:**
 - Deadlocks and livelocks occur due to unusual combinations of conditions that are hard to anticipate or reproduce.
 - It is often most effective to use *inspection* to detect such defects, rather than testing alone.
 - However, when testing:
 - Vary the time consumption of different threads.
 - Run a large number of threads concurrently.
 - Deliberately deny resources to one or more threads.

Defects in Timing and Co-ordination

- **Critical races**
 - **Defects:**
 - One thread experiences a failure because another thread interferes with the 'normal' sequence of events.
 - **Testing strategies:**
 - It is particularly hard to test for critical races using black box testing alone.
 - One possible, although invasive, strategy is to deliberately slow down one of the threads.
 - Use inspection.

Defects in Handling Stress and Unusual Situations

- Insufficient throughput or response time on minimal configurations
- Incompatibility with specific configurations of hardware or software
- Defects in handling peak loads or missing resources
- Inappropriate management of resources
- Defects in the process of recovering from a crash

Writing Formal Test Cases and Test Plans

- A *test case* is an explicit set of instructions designed to detect a particular class of defects in a software system.
- A test case can give rise to many tests.
- Each test is a particular running of the test case on a particular version of the system.

Test plans

- A *test plan* is a document that contains a complete set of test cases for a system
 - Along with other information about the testing process.
- The test plan is one of the standard forms of documentation.
- If a project does not have a test plan:
 - Testing will inevitably be done in an ad-hoc manner.
 - Leading to poor quality software.
- The test plan should be written long before the testing starts.
- You can start to develop the test plan once you have developed the requirements.

Information to include in a test case

- A. Identification and classification:
- Each test case should have a number, and may also be given a descriptive title.
 - The system, subsystem or module being tested should also be clearly indicated.
 - The importance of the test case should be indicated.
- B. Instructions:
- Tell the tester exactly what to do.
 - The tester should not normally have to refer to any documentation in order to execute the instructions.
- C. Expected result:
- Tells the tester what the system should do in response to the instructions.
 - The tester reports a failure if the expected result is not encountered.
- D. Cleanup (when needed):
- Tells the tester how to make the system go 'back to normal' or shut down after the test.

Strategies for Testing Large Systems

- Big bang testing versus integration testing
 - In *big bang* testing, you take the entire system and test it as a unit
 - A better strategy in most cases is *incremental testing*:
 - You test each individual subsystem in isolation
 - Continue testing as you add more and more subsystems to the final product
 - Incremental testing can be performed *horizontally* or *vertically*, depending on the architecture
 - Horizontal testing can be used when the system is divided into separate sub-applications

Top-down testing

- Start by testing just the user interface.
- The underlying functionality are simulated by *stubs*.
 - Pieces of code that have the same interface as the lower level functionality.
 - Do not perform any real computations or manipulate any real data.
- Then you work downwards, integrating lower and lower layers.
- The big drawback to top down testing is the cost of writing the stubs.

Bottom-up testing

- Start by testing the very lowest levels of the software.
- You needs *drivers* to test the lower layers of software.
 - Drivers are simple programs designed specifically for testing that make calls to the lower layers.
- Drivers in bottom-up testing have a similar role to stubs in top-down testing, and are time-consuming to write.

Sandwich testing

- Sandwich testing is a hybrid between bottom-up and top down testing.
- Test the user interface in isolation, using stubs.
- Test the very lowest level functions, using drivers.
- When the complete system is integrated, only the middle layer remains on which to perform the final set of tests.

The test-fix-test cycle

- When a failure occurs during testing:
 - Each failure report is entered into a failure tracking system.
 - It is then screened and assigned a priority.
 - Low-priority failures might be put on a *known bugs list* that is included with the software's *release notes*.
 - Some failure reports might be merged if they appear to result from the same defects.
 - Somebody is assigned to investigate a failure.
 - That person tracks down the defect and fixes it.
 - Finally a new version of the system is created, ready to be tested again.

The ripple effect

- There is a high probability that the efforts to remove the defects may have actually added new defects
 - The maintainer tries to fix problems without fully understanding the implications of the changes
 - The maintainer makes ordinary human errors
 - The system *regresses* into a more and more failure-prone state

Regression testing

- It tends to be far too expensive to re-run every single test case every time a change is made to software.
- Hence only a subset of the previously-successful test cases is actually re-run.
- This process is called *regression testing*.
 - The tests that are re-run are called regression tests.
- Regression test cases are carefully selected to cover as much of the system as possible.
- The “law of conservation of bugs”:
 - *The number of bugs remaining in a large system is proportional to the number of bugs already fixed*

Deciding when to stop testing

- All of the level 1 test cases must have been successfully executed.
- Certain pre-defined percentages of level 2 and level 3 test cases must have been executed successfully.
- The targets must have been achieved and are maintained for at least two cycles of ‘builds’.
 - A *build* involves compiling and integrating all the components.
 - Failure rates can fluctuate from build to build as:
 - Different sets of regression tests are run.
 - New defects are introduced.

The roles of people involved in testing

- The first pass of unit and integration testing is called *developer testing*.
 - Preliminary testing performed by the software developers who do the design.
- *Independent testing* is performed by a separate group.
 - They do not have a vested interest in seeing as many test cases pass as possible.
 - They develop specific expertise in how to do good testing, and how to use testing tools.

Testing performed by users and clients

- **Alpha testing**
 - Performed by the user or client, but under the supervision of the software development team.
- **Beta testing**
 - Performed by the user or client in a normal work environment.
 - Recruited from the potential user population.
 - An *open beta release* is the release of low-quality software to the general population.
- **Acceptance testing**
 - Performed by users and customers.
 - However, the customers do it on their own initiative.

Inspections

- An inspection is an activity in which one or more people systematically
 - Examine source code or documentation, looking for defects.
 - Normally, inspection involves a meeting...
 - Although participants can also inspect alone at their desks.

Inspecting compared to testing

- Both testing and inspection rely on different aspects of human intelligence.
- Testing can find defects whose consequences are obvious but which are buried in complex code.
- Inspecting can find defects that relate to maintainability or efficiency.
- The chances of mistakes are reduced if both activities are performed.

Testing or inspecting, which comes first?

- It is important to inspect software *before* extensively testing it.
- The reason for this is that inspecting allows you to quickly get rid of many defects.
- If you test first, and inspectors recommend that redesign is needed, the testing work has been wasted.
 - There is a growing consensus that it is most efficient to inspect software *before* any testing is done.
- Even before developer testing

Measure quality and strive for continual improvement

- **Things you can measure regarding the quality of a software product, and indirectly of the quality of the process**
 - The number of failures encountered by users.
 - The number of failures found when testing a product.
 - The number of defects found when inspecting a product.
 - The percentage of code that is reused.
 - More is better, but don't count clones.
 - The number of questions posed by users to the help desk.
 - As a measure of usability and the quality of documentation.