

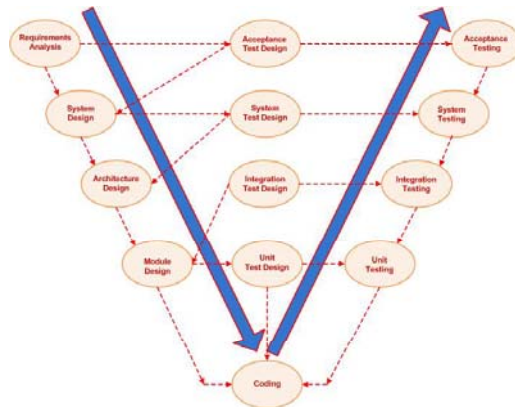
Vragen?

- Noem een aantal niet functionele requirements
- Noem een aantal elicitering technieken
 - Wat is het risico van taak analyse?
- Geef een aantal eigenschappen waaraan requirements moeten voldoen
- Wat voor technieken zijn er voor het valideren van requirements?

Software Design Chapter 12

Mark van den Brand

V model for software development



Software design

- Programmer's approach:
 - Skip requirements engineering and design phases
 - Start writing code
- Why?
 - Design is a waste of time
 - We need to show something to the customer real quick
 - We are judged by the amount of LOC/month
 - We expect or know that the schedule is too tight

Software design

- Design is a trial-and-error process
- There is an interaction between requirements engineering, architecting, and design
- Design traps:
 - There is no definite formulation
 - There is no stopping rule
 - Solutions are not simply true or false
 - There may be a whole range of possible (good) solutions

Process of design

- *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 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.
 - Types 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
 - Use frameworks/libraries as much as possible

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

- **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
 - Discussed more in Chapter 13
 - 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
 - Example: 75% of the code is used to parameter checking

Design principles

- Abstraction
- Modularity, coupling and cohesion
- Information hiding
- Limit complexity
- Hierarchical structure

Abstraction

- Procedural abstraction
 - natural consequence of stepwise refinement
 - name of procedure denotes sequence of actions
- Data abstraction
 - aimed at finding a hierarchy in the data

Modularity

- Structural criteria which tell us something about individual modules and their interconnections
- Modern programming languages support modularity
- Cohesion and coupling
 - cohesion: the glue that keeps a module together
 - coupling: the strength of the connection between modules
 - keep track of this via measuring!

Types of cohesion

- **coincidental cohesion**
 - elements are grouped into components in a random manner
- **logical cohesion**
 - elements realize logical related tasks
- **temporal cohesion**
 - elements are independent but are active at the same moment in time
- **procedural cohesion**
 - elements are executed in a given order

Types of cohesion

- **communicational cohesion**
 - elements operate on the same (external) data
- **sequential cohesion**
 - a sequence of elements where output of one is input for the other
- **functional cohesion**
 - elements contribute to a single function
- **data cohesion (for abstract data types)**

Types of coupling

- **content coupling**
 - change of data by another component
 - arbitrary control to another component
- **common coupling**
 - shared data
- **external coupling**
 - files
- **control coupling**
 - flags
- **stamp coupling**
 - shared knowledge on data formats
- **data coupling**

strong cohesion & weak coupling ⇒ simple interfaces ⇒

- **simpler communication**
- **simpler correctness proofs**
- **changes influence other modules less often**
- **reusability increases**
- **comprehensibility improves**

Information hiding

- Each module has a secret
- Design involves a series of decision: for each such decision, wonder who needs to know and who can be kept in the dark
- Information hiding is strongly related to
 - abstraction: if you hide something, the user may abstract from that fact
 - coupling: the secret decreases coupling between a module and its environment
 - cohesion: the secret is what binds the parts of the module together

Complexity

- Measure certain aspects of the software (lines of code, # of if-statements, depth of nesting, ...)
- Use these numbers as a criterion to assess a design, or to guide the design
- Interpretation: higher value \Rightarrow higher complexity \Rightarrow more effort required (= worse design)
- Two kinds:
 - intra-modular: inside one module
 - inter-modular: between modules