Lecture 7: Organization and quality of software requirements

- Non-functional requirements types
- Properties of a good software requirements specification
- Techniques for user/software requirements elicitation

Non-functional requirements

Non-functional requirements describe the quality features of the prospective system.

- How fast should be the system?
- How reliable should be the system?
- How safe should be the system?
- How user-friendly should be the system?
- What norms should the system comply to?

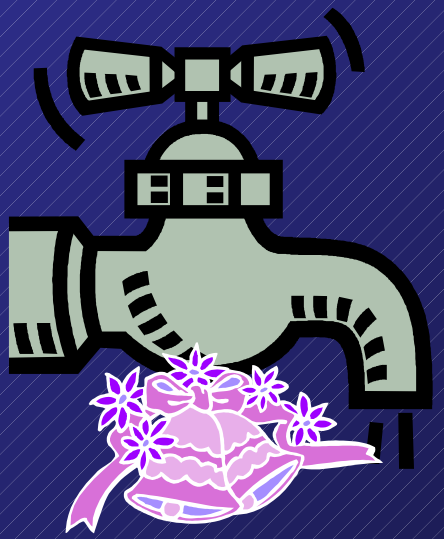
Non-functional requirements can be global (pertain to the whole system) or local (directly pertain only to specific functional requirements).

Non-functional requirements types

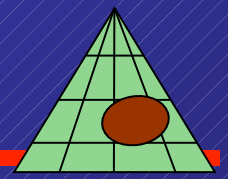
Non-functional requirements determine important quality characteristics of the system that are outside of its functionality.

The following system's characteristics should be covered:

- Look-and-feel – how does the system look for the users
- Usability – how easy it is to use and learn the system
- Reliability – how hard it is for the system to "break"
- Performance – how fast and capable the system is
- Supportability – how easy it is to service and extend the system
- Security – how well the system prevents from unauthorised use
- Corporate compatibility – political and legal issues



Metrics for non-functional requirements



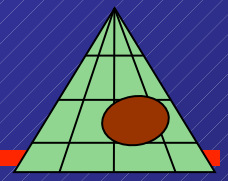
Look-and-feel – level of compatibility with the corporate standard, level of readability of text/icons (e.g. can it be read by the CEO without glasses), level of compliance with the rules of ergonomy, …

Usability – time necessary to learn the system, time necessary to perform typical actions, the number of users using the system after a month, the number of user satisfied with the system, number of mouse clicks to perform typical tasks, …

Reliability – maximum time the system is unavailable for maintenance, maximum recovery time, maximum amount of data that can be lost (has to be re-introduced), …

Performance – speed of data transfer, typical or specific response time of the system, acceptable increase in response time for rush hours or with increase of stored data, …

Metrics for non-functional requirements (2)



Supportability – the amount of time spent by the administrator to support the system, the amount of effort necessary to extend the system, the level of compatibility with software standards, the amount of time to install the system…

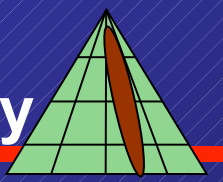
Security – the amount of money or time spent by specialists to break into the system without success, the level of data encryption, the size of passwords, …

Corporate compatibility – norms, regulations the system should conform to…

NOTE(1): metrics can be subjective! Testing should be done by independent specialists in the given area. Some metrics can be collected through polls.

NOTE(2): you have to remember about the budget for testing (e.g. the money for a company that would try to break into the system)

Non-functional reqs – ISO-compliant taxonomy



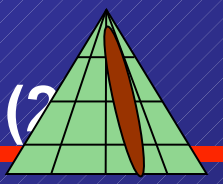
Functionality: A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs **(suitability, accuracy, interoperability, compliance, security)**

Reliability: A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time **(maturity, recoverability, fault tolerance)**

Usability: A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users **(learnability, efficiency of use, operability)**

Efficiency: A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions **(time behaviour, resource behaviour)**

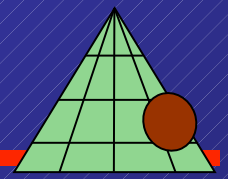
Non-functional reqs – ISO-compliant taxonomy (2



Maintainability: A set of attributes that bear on the effort needed to make specified modifications **(stability, analyzability, changeability, testability)**

Portability: A set of attributes that bear on the ability of software to be transferred from one environment to another **(installability, replaceability, adaptability)**

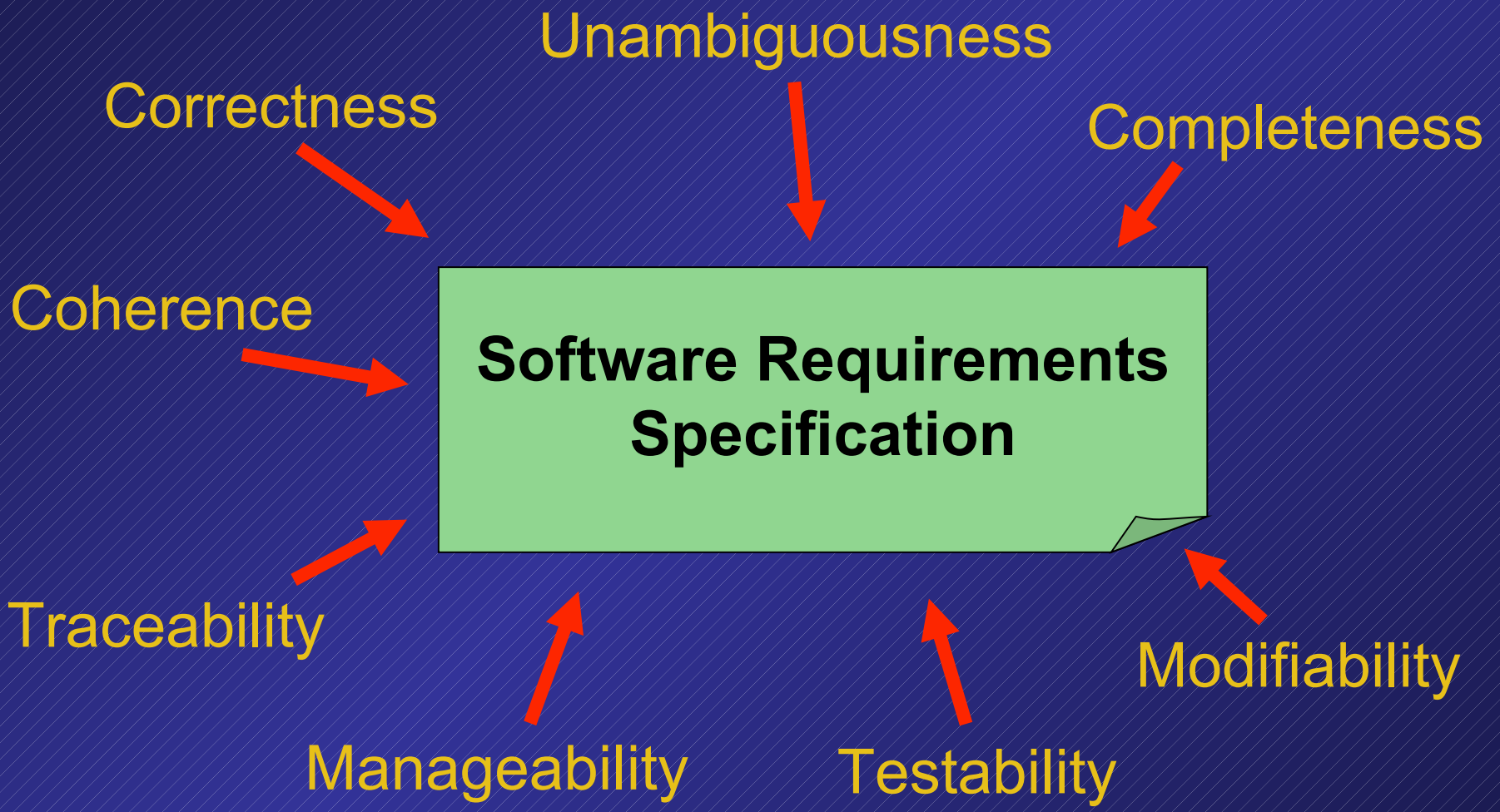
Constraint types



Constraints describe the current (given) environment that will influence the design of the system.

- Given software package – software that will be used to implement the system
- Given workstation hardware and operating system
- Given network system and its architecture
- Given hardware and software for the servers, peripheral equipment etc.
- Existing data – size and structure, availability to be copied
- Existing architectural standards
- Geographical location of existing equipment
- Existing communication protocols
- Volumes of data, users.

Qualities of a good software requirements specification



Completeness

Do we have all the requirements?

- Constantly ask yourself (and your client): is there anything else? did we miss anything?
- Use techniques appropriate for discovering the “undiscovered” requirements (see further slides)
- Remember to include requirements covering all four areas (functional, non-functional, vocabulary, constraint)

Software requirements are complete when we have:

- All the important scenarios for all the use cases,
- All the non-functional requirements covering all areas (see further),
- All the attributes and operations for notions used in scenarios and non-functional requirements,
- All the constraints covering all areas (see further)

Unambiguousness

Do all the requirements have only one interpretation?

- Are the requirements clear to the user? Are they clear to the developer?
- Talk to the clients, write down your models, present them to the client and again talk to your clients…
- Best way to really verify that requirements are unambiguous: develop the system – that's why we have iterative lifecycle

We have clear and unambiguous requirements when we:

- Write use case names that describe a clear and important goal
- Write use case scenarios with simple sentences (SVO)
- Write storyboards with precisely described user interface
- Write precise definitions of all the sentence objects (classes and attributes) and verbs (operations)
- Give necessary constraints to use case and class diagrams
- …..

Correctness

Should these be really the properties of the system?

- Constantly ask yourself, and clarify it with the user: should the system really behave like this? should it really handle this data (notions)?

Software requirements are correct when:

- Well, really they can be seen as correct, when the user agrees to pay you for the system… i.e. when the system is ready and the client acknowledges that this is what he/she/they wanted

To assure correctness we should:

- Write storyboards that give the user a feeling of how the system should behave and what data will it manipulate
- Compare requirements with the client's business
- Receive a detailed feedback from the client on storyboards

Coherence

Are software requirements consistent with the user requirements and not contradictory between themselves?

- Constantly verify that every requirement is not in contradiction with any other requirement.
- NOTE: this task is hard to do with “novel-like” requirements!

In order to have coherent requirements:

- Assure that every notion used in scenarios or non-functional requirements is defined in the class model.
- Assure that you don’t have synonyms (classes with different names but same contents) or homonyms (classes with the same name but different contents).
- Assure that operations in classes are associated with verbs in scenarios
- Assure that scenarios have proper branching (use activity diagrams)
- …

Traceability

Do we have identifiers for all the requirements and can we induce all the identifiers from the identifiers of the higher level (user requirements or business description)?

- Make sure that you give identifiers to all the requirements (use cases, scenarios, classes).
- Make sure that every identifier on the software requirements level has a source identifier.

Traceable requirements mean that:

- Every scenario (activity diagram) is associated with an appropriate use case
- Every class is associated with an appropriate notion or feature
- In general – every requirement has its source and can be translated into design

NOTE: traceability is not a 1 to 1 relationship – several requirements might be traced into one or more other requirements…

Manageability

Has all the requirements have attributes that allow for their manipulation in the software development project?

- Make sure that requirements allow for placing them clearly in the software development lifecycle.

In order to manage requirements properly you would need to attach several attributes to every requirement:

- User priority (how important it is for the user)
- Technical difficulty (how important it is for the architecture)
- Assignment (responsible person)
- Version (number in the sequence of changes)
- Target release (iteration where it should it be implemented)
- Stability (probability that it would change)
- …

Testability

Has every requirement got a quality measure that allows for verifying the final system (see also “correctness”)?

- Always ask yourself (and the client): how will we verify that this requirements is met by the final system?
- Always bear in mind that implementation of a requirement needs some proof (usually a test) that the system is correct.

Testability for different types of requirements mean that:

- Functional requirements (use cases) have procedures (test cases) for testers to check proper behaviour of the system.
- Vocabulary requirements (classes) have associated test data to verify proper data entry, processing and output by the system.
- Non-functional requirements have measures (precise numbers) that can be checked objectively with appropriate test procedures.

Modifiability

Is it easy to introduce (and trace) changes in the software requirements specification?

- Remember that changes in requirements happen in every (well, almost every…) project!
- Make sure that your requirements allow to point precisely to the place where a change was made.

Modifiable requirements are divided into separate "chunks":

- Use cases that have precisely distinguished scenarios, sentences in scenarios and elements of sentences (just follow the structure proposed in this course!).
- Classes that have lists of attributes and operations.

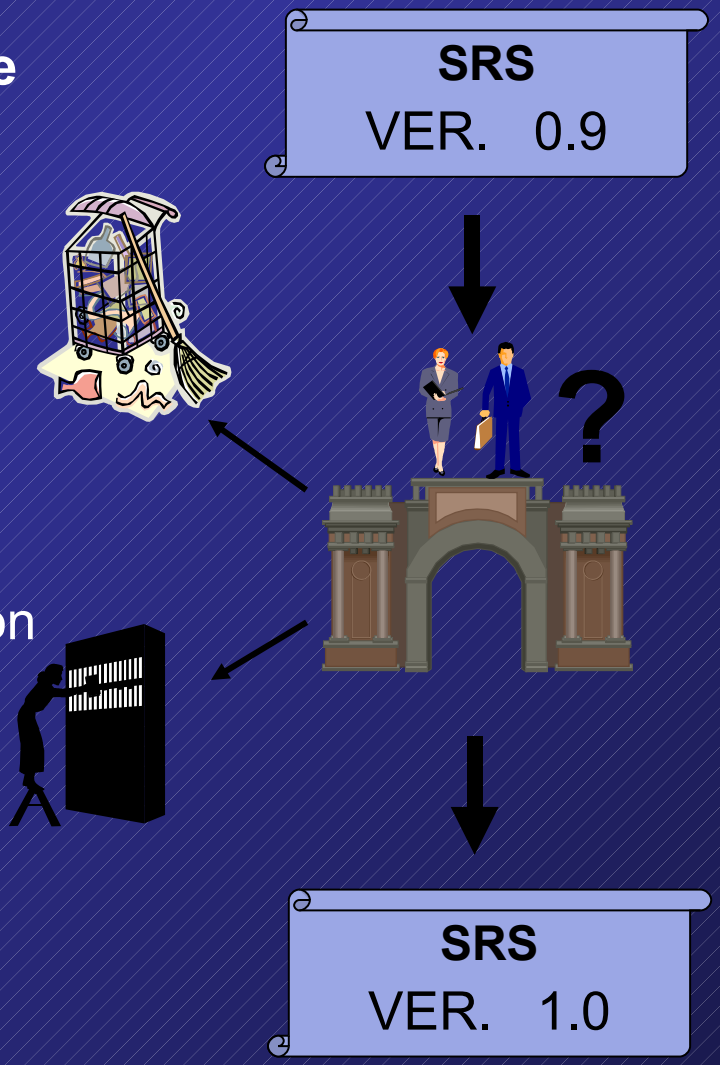
Checking quality of requirements specification

After every Software Requirements Phase in a development process we have to allow the clients and the developers to agree on the quality of requirements.

Quality gate for requirements.

We reject (or temporality postpone) requirements that:

- Are not consistent with the system vision
- Do not meet quality criteria (see previous slides)
- Do not fit into scope due to budget limitations
- Constitute “golden taps” or “gadgets”
- Impossible or too expensive to test



Requirements specification techniques

In order to meet quality criteria, we need to use appropriate techniques to gather and specify requirements. Techniques support creative processes necessary for requirements specification.

- Techniques for working with documents
- Techniques for working with individuals
- Techniques for working with groups

General guidelines:

- Try to look at the business processes and seek for ways to automate them
- Try to seek for rules, re-occurring procedures, products, etc.
- Always seek for clients' replies/opinions to the requirements models
- Try to be creative in discovering requirements

Techniques for working with documents

Underlining notions

- Very simple but underestimated technique
- We underline noun and verb groups – candidates for elements of the vocabulary or actions in the process/scenarios.
- We clarify notions and seek for possible synonyms/homonyms with other techniques.

Marking requirements

- We mark fragments of text to become distinguished requirements “chunks”.
- For electronic documentation – we introduce text fragments into a CASE tool, and add attributes for easy management

Seeking for use cases

- User manuals of legacy/substituted systems can contain sections describing user interactions with the system.

Techniques for working with individuals

Apprenticeship: master and his student

- The student gets to know the master's work by observing activities performed by the master and by asking relevant questions.
- The master is the potential user of the system. The student – analyst, gathers knowledge about the everyday user's work.
- We ask questions: "why did you do it?", "how often does this happen?", etc.

Interviews: questions and answers

- Interviews should be prepared: the user knows the subject and prepares in advance.
- Use of "anchors": use case scenarios control the conversation – don't allow for "detours" (the user talks about irrelevant things).
- Keeping terminology precise: the user talks with his/her own language, but we always clarify the meaning of terms and update the vocabulary (class model).

Requirements workshops

JAD (Joint Application Development) type sessions. Analysts build the specification in close cooperation with a group of key users.

Common notation – use cases, scenarios, classes.

A moderator – person that tries to reach an agreement between all the participants: this is often a very difficult task.

A secretary – person that knows notation well and denotes all the results of the session.

Concentration on the result: to determine the goal of every use case, to write its scenarios with possible endings, to come out with a consistent vocabulary (classes).

NOTE: requirements workshops can last for several days in every iteration – very good means to communicate! Hint: company meetings in a remote place…

Brainstorms

Well known technique that increases innovation; here: innovation in discovering requirements.

We gather a group of open-minded people and ask them to supply us with as many ideas for the new system as possible.

Guidelines:

- Don’t criticize any, even the most impossible idea.
- Don’t discuss ideas, don’t stop in finding new ideas.
- Write down all ideas at once (“ideas evaporate faster than water”).
- Give priority to quantity – quality will come later.
- Build ideas on previous ones (jump from an idea into another one).
- If you get stuck – start with a work randomly chosen from a dictionary and associated somehow with our system.

Ideas that come out of a brainstorm can be processed later with other techniques.

Summary – how will WE do it?

We should supplement functional and vocabulary requirements by various types of non-functional requirements and constraints.

We should write complete, unambiguous, correct, coherent, traceable, manageable, testable and modifiable requirements.

These qualities of SRS can be assured by its proper organization – see previous lectures!

Quality of requirements should be checked every iteration.

Quality of requirements is really verified only after the system is implemented – that's why we should implement the system iteratively. Only then we can assure success of the project.

In order to discover real needs of the client we should use proper requirements elicitation techniques: individual, group, documents…