Modeling Requirements
Requirements Verification and Validation

Nicolas Sannier
* EDF R&D – STEP, 6 Quai Watier BP49
  78401 Chatou, France
  nicolas.sannier@edf.fr
** Inria, Campus Universitaire de Beaulieu,
  35042, Rennes Cedex, France
  nicolas.sannier@inria.fr
Where does this lecture come from?

- Make of work, experiences, slides, stuff borrowed from / inspired by / …
  - Scott Adams
  - Daniel Amyot
  - Benoit Baudry
  - Steve Easterbrook
  - Daniel Lucas-Hirtz
  - Neil Maiden
  - Gunter Mussbacher
  - And so many others
Summary from Last Session

- Key terms about RE Elicitation and Specification
  - Elicitation is problem oriented and NOT solution oriented
    - Is about discovering the boundaries of the right system to build
    - Is about collaborative work between Stakeholders and Requirements Engineers
    - Is not spontaneous
    - Is about making explicit what is implicit
    - Different techniques (observations, interviews, brainstorming, modeling…)
  - Specification is about writing clear, precise, unambiguous, atomic, complete, testable, maintainable Requirements
    - Patterns for writing (Volere shells, EARS)
    - Tips/pitfalls to avoid (modal verbs, active voice, requirements on the system, not on users …)
What is the program for today?

• Introduction
• Requirements Modeling
  – Modeling Notations
  – What is the Best Model?
  – Goal Modeling
  – Modeling Structure, Behavior, Interactions
  – NFRs, Quality Measures and Modeling
• Requirements V&V
  – Two levels of requirements V&V
  – Some Requirements V&V Techniques
  – Model-based Requirements V&V
• Some Training Exercises
INTRODUCTION ON REQUIREMENTS ANALYSIS, MODELING, VERIFICATION AND VALIDATION
Requirements Analysis

• You said Requirements analysis?
  – The process of studying and analyzing the customer and the user needs to arrive at a definition of the problem domain and system (and software) requirements
  – Analysis and Elicitation feed each other

• Objectives
  – Detect and resolve conflicts between (user) requirements
  – Negotiate priorities of stakeholders
  – Prioritize and triage requirements
  – Elaborate system requirements,
    • To be documented in the requirement specification document
    • such that managers can give realistic project estimates
    • and such that developers can design, implement, and test
  – Classify requirements information into various categories and allocate requirements to sub-systems
  – Evaluate requirements for desirable qualities
  – Make sure that nothing major is forgotten
Requirements Modeling

• **This is an essential task in specifying requirements**
• Map elements obtained by elicitation to a more precise/“formal” form
• Help better understand the problem
• Help find what is missing or needs further discussion
• Different modeling languages
  – Informal:
    • natural language
  – Goal-oriented modeling (GRL)
  – Functional modeling:
    • UML / SysML
    • SDL (Specification and Description Language)
    • Logic, temporal logic
    • UCM (Use Case Maps)
  – Domain Modeling
    • DSLs, UML Profiles
  – ...
Requirements Verification and Validation

• Need to be performed at every stage during the (requirements) process
  – Elicitation
    • Checking back with the elicitation sources
    • “So, are you saying that . . . . . ?”
  – Analysis
    • Checking that the domain description and requirements are correct
  – Specification
    • Checking that the defined system requirement will meet the user requirements under the assumptions of the domain/environment
    • Checking conformity to well-formness rules, standards…

• Requirements Verification: Have we written the good requirements?
• Requirements Validation: Is the system conforming to its requirements?
Requirements Analysis vs Requirements V&V

• Both have several activities in common
  – Reading requirements, problem analysis, meetings and discussions...
• But inputs are different!

• Analysis works with raw, incomplete requirements as elicited from the system stakeholders
  – Develop a software requirements specification document
  – Emphasis on "we have the right requirements"

• Requirements V&V works with a software requirements specification and with negotiated and agreed (and presumably complete) domain requirements
  – Check that these specifications are accurate
  – Emphasis on "we have the right requirements well done"
MODELING REQUIREMENTS
You Said Modeling?

• “A model represents reality for a given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality.”

• Modeling is useful to:
  – Help understand complex problems and / or solutions
  – Communicate information about the problem / solution
  – Direct implement (especially in software)
  – Verify an assertion, simulate a behavior, …

• Qualities of a good model
  – Abstract
  – Understandable
  – Accurate
  – Inexpensive
  – Reusable
Which one is the Best?

- Different *fitness for purpose*

- “You are all different”
  Brian Cohen (Graham Chapman) – Life of Brian (1979)
Modeling Notations

• **Natural language**
  + No special training required
  - Ambiguous, verbose, vague, obscure ...
  - No automation

• **Ad hoc notation (bubbles, squares and arrows)**
  + No special training required
  - No syntax formally defined meaning not clear, ambiguous
  - No automation

• **Semi-formal notation (URN, UML, SysML...)**
  + Syntax (graphics) well defined
  + Partial common understanding, reasonably easy to learn
  + Partial automation
  - Meaning only defined informally
  - Still a risk of ambiguities

• **Formal notation (Logic, SDL, Petri nets, FSM ...)**
  + Syntax & semantics defined
  + Great automation (analysis and transformations)
  - More difficult to learn & understand
Informal and Formal Languages

• Informal language is better understood by all stakeholders
  – Good for user requirements, contract
  – But, language lacks precision
  – Possibility for ambiguities
  – Lack of tool support

• Formal languages are more precise
  – Fewer possibilities for ambiguities
  – Offer tool support (e.g. automated verification and transformation)
  – Intended for developers
  – Limited scope
Goal Modeling - Goal Oriented Requirements Engineering

• “A goal is the intentional description of a desired characteristic of the system to be”
• A goal is the most abstract description of requirements
• All requirements shall be traced back to a goal

• Goal hierarchy from high level goals to lower level (refined) goals

• Documenting and detailing goals is useful for:
  – Shared understanding of the system (vision)
  – **Requirements Elicitation** with goals
  – Identify **Alternatives**
  – Detect **inconsistent** requirements
  – **Justify** requirements
  – Exhibit requirements **Completeness**
  – Resolve **conflicts**
  – Goals are more **stable** than requirements
    • goals are about the problem definition
Goal Modeling

**AND composition**

G1  G2  ..  Gn

**OR composition**

G1  G2  ..  Gn

Goal G

Satisfy the registered user’s needs

Provide the library’s required books

Propose a rich and diversified index of books

Maintain Availability

« WHY »

Decrease unavailability of books

« HOW »

Conflict

Require
Goal Oriented Requirements Engineering with KAOS

- Knowledge Analysis in autOmated Specification
  University of Oregon and Université Catholique de Louvain (Belgium) in the 90’s
- Now in the industry with the tool Objectiver provided by Respect-it

Goals are refined according to several tactics: 
agent-driven, case-driven, milestone-driven
Modeling Structure and Domains

- Modeling Structure
- Concepts of Entities and their Relationships. Use one of the following notations:
  - ERD (Entity Relationship Diagram – the traditional version)
  - UML class diagrams
  - Relational tables
- Can be used for the following
  - Model of the problem domain (called “domain model”)
    - The two versions: existing and to-be
  - Model of input and output data structures of system-to-be
  - Model of the stored data (database)
    - not necessarily an image of the domain data
    - Additional data is introduced (e.g. user preferences)
  - Architectural design of the system-to-be
Modeling Dynamic Behavior

- Behavior modeling techniques
  - Text (plain, function statements, use cases)
  - Decision tables
  - Activity Diagrams / Use Case Maps
  - Finite state machines
    - Simple state machines (FSM): state diagrams or transition tables
    - Extended state machines (e.g. UML State Machines)
    - Harel’s State Charts (concepts included in UML notation)
    - Petri nets (allows for flexible concurrency, e.g. for data flow, similar to Activity Diagrams)
  - Logic (e.g. Z, B, CTL) for describing input-output assertions and relationships to internal object state that is updated by operations
- It is important to chose what best suits the problem
Entity-Relationship modeling (originally proposed by Peter Chen in 1976)

- **Concepts:**
  - **Entity:** represents a type of entity instances, defines the properties that hold for all such instances.
  - **Relationship:** represents relationship instances that hold between certain pairs of entity instances.
    - The related entity types are also called **roles**.
    - **Multiplicity information** indicate how many instances of the “other” side may be related to a given instance of “this” side.
  - **Attribute:** An entity or a relationship may have one or several attributes. Each attribute is identified by a name and its type, where such a type is usually some simple data type such as integer or character string.

**Note:** An entity type is normally not used as the type of an attribute, because such a situation is rather represented by a relationship between the given entity and the attribute type.
Modeling with UML

- According to UML Reference Manual
  - Structural
    - Class, object, composite structure, component, and use case diagrams
  - Dynamic (that is, describing dynamic behavior)
    - State machine, activity, sequence, communication, timing, and interaction overview diagrams
  - Physical
    - Deployment diagrams
  - Model Management
    - Package diagram
UML Models Useful in RE

- Use case diagram
  - Use cases structuring
  - System boundaries definition
- Class diagram
  - Domain modeling
- Activity diagram
  - Workflow and process modeling
  - Concepts much related to concepts of Use Case Maps
- Sequence diagram
  - Modeling of message exchange scenarios
- State machine diagram
  - Detailed behavioral specification (of objects, protocols, ports…)
  - System behaviour (black box)
Describing the System Boundaries with the UC Diagram

**System**: name and boundary of the system to be

**Actor**: Persons or Entities around the system that interact with the system to achieve a goal.

**Use Case**: describe one goal useful to one or several actors

**Communication**: Actor’s participation to a use-case
Misuse cases

- Present both positive and negative interactions and scenarios
RE and Class Diagrams
Modeling Domains
Nuclear Regulatory Background
Training: Modeling a portion of World of Warcraft
Training: (Meta)Modeling a portion of World of Warcraft

• Gathering, clustering and organizing concepts around:
  – Accounts
  – Realms
  – Regions
  – Dungeons
  – Factions
  – Classes
  – Races
  – Characters
  – Characteristics
  – Monsters
  – Equipment
  – Achievements
Training: (Meta)Modeling a portion of World of Warcraft
RE and Object Oriented Analysis

• Five main steps
  – Identify core classes within problem domain
  – Model relationships between classes
    • Class diagram
    – Define the attributes (properties) associated with each class
    – Determine relevant operations (actions) for each class
    – Define the messages that may be passed between objects
      • Interaction diagram, state machine diagram
Things that are Said and not Said / Hard to Express

• Modeling at the right level
  – Global perspectives vs local perspectives
  – Most OOA approaches actually address high-level design...
  – Class diagrams can however be used for analysis, especially for the description of domain concepts

• Further composition and decomposition problems
  – Related requirements cannot all be assigned to a single component or a single class
  – One scenario may affect several classes at once
  – OO modularization is not perfect either...
    ➔ Scattering and tangling effects - Motivation for aspect-oriented analysis and design
Problems with Class Diagrams

Scattering: design elements to support R1 in many components

Tangling: single component has elements for many requirements
Separation of Concerns with Aspect Oriented Programming
(For Information Only)
Scenarios Modeling

• A scenario describes one concrete example of the satisfaction (or failure to satisfy) of one goal (or a set of goals). It provides details on one or several goals. A scenario defines a sequence of interactions, performed to satisfy the goal, and link this sequence/these steps with the system context.


Documented information:
• Actors: persons or systems interacting with the system under evaluation
• Roles: Specific kinds of actors
• Goals: each scenario detail the satisfaction of one (or several) goal(s)
• Pre-conditions: pre-requisite to execute the scenario
• Post-conditions: conditions and properties after the scenario
• Resources: information on persons, documents, systems required to execute the scenario
• Localization: where the scenario is executed
Why using Scenarios?

- Stakeholders communicate more in terms of examples than in terms of concepts.
- Scenarios allow to:
  - Express concrete intents within examples.
  - **Link** goals (your vision) and requirements specifications.
  - Ease **Communication** and understanding.
  - Allow and ease requirements **refinements** and **Development**.
  - Put requirement in **context**.
  - **Ease negotiations** and conflict resolutions.
  - Propose different **perspectives** on requirements.

- When?
  - Elicitation step to go further more comprehension of the problem.
  - Toward test-cases generations.
11 Tips for Writing Better Scenarios

• Grammar
  – Use the present form
  – Use the active voice
  – Follow a simple subject-verb-complement pattern
  – Avoid modal verbs (shall, should, etc.)
    • You’re not writing requirements, but the operational behavior

• Structure
  – One complete and independent sentence per step
  – Enumerate each step of the scenario

• Content
  – One interaction sequence per scenario
  – Describe the scenario from an external perspective
  – Name specifically all involved actors
  – Name explicitly all goals satisfied by the scenario
  – Keep focus on how the goal is satisfied, avoid poorly related information
RE and Activity Diagrams

- Describe the dynamic behavior of a system as a flow of activities (workflow)
- Flow
  - Sequence
  - Alternative
  - Parallel
- Data flow modeling
- Managing concurrency
- Conditions on parallelism
  - (branches of an AND-fork)
- Constraints on action pins
- Integration with OCL
RE and Sequence Diagrams

- Describe the **dynamic behavior** as interactions **between so-called “participants”** (e.g. agents, actors, the system, system components). For each participant, there is a “lifeline”
- Good for Requirements Elicitation (stakeholders and system interactions)
RE and State Machine Diagrams

- Modeling the **behavior of ONE individual object** (with states and transitions)
- State machines are suitable for describing reactive systems
- Not appropriate to describe continuous systems (e.g., spacecraft trajectory control, stock market predictions)
- In RE, state machines are quite popular for describing the lifecycle of documents and other artifacts.
Training: describes this behaviour
What About Non Functional Requirements?

- Functional requirements describe what the system should do
  - things that can be captured in use cases
  - things that can be analyzed by drawing interaction diagrams, statecharts, etc.
  - Functional requirements will probably trace to individual chunks of a program

- Non-functional requirements are global constraints on a software system
  - e.g. development costs, operational costs, performance, reliability, maintainability, portability, robustness etc.
  - Usually cannot be implemented in a single module of a program

- The challenge of NFRs
  - Hard to model
  - Usually stated informally, and so are:
    - often contradictory,
    - difficult to enforce during development
    - difficult to evaluate for the customer prior to delivery
  - Hard to make them measurable requirements
    - We’d like to state them in a way that we can measure how they’ve been met
Software Quality

• Most definitions require compliance with requirements
• “Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.”¹
• Implication:
  – We need to be able to explicitly quantify requirements and verify that any solution meets them
  – We need measures
• An interesting phenomenon:
  Measurable objectives are usually achieved!

• Therefore, unless you have unrealistic values, requirements are usually met
• Important to know what measures exist!
• The chosen values, however, will have an impact on the amount of work during development as well as the number of alternatives and architectural designs from which developers may choose to meet the requirements
Examples of NFRs

- **Interface requirements**
  - how will the new system interface with its environment?
  - User interfaces and “user-friendliness”
  - Interfaces with other systems

- **Performance requirements**
  - time/space bounds
  - workloads, response time, throughput and available storage space
  - e.g. "the system must handle 1,000 transactions per second"

- **Reliability**
  - the availability of components
  - integrity of information maintained and supplied to the system
  - e.g. "system must have less than 1hr downtime per three months"

- **Survivability**
  - e.g. system will need to survive fire, natural catastrophes, etc

- **Operating requirements**
  - physical constraints (size, weight),
  - personnel availability & skill level
  - accessibility for maintenance
  - environmental conditions
Types of Non Functional Requirements

- Non-functional requirements
  - Product requirements
    - Efficiency requirements
      - Usability requirements
      - Performance requirements
    - Reliability requirements
    - Portability requirements
  - Organizational requirements
    - Interoperability requirements
    - Ethical requirements
      - Legislative requirements
  - External requirements
    - Delivery requirements
    - Implementation requirements
    - Standards requirements
    - Privacy requirements
    - Safety requirements

- Space requirements
Making Requirements Measurable

• Define ‘fit criteria’ for each requirement
• Give the ‘fit criteria’ alongside the requirement
• E.g. for new ATM software
  – Requirement: “The software shall be intuitive and self-explanatory”
  – Fit Criteria: “95% of existing bank customers shall be able to withdraw money and deposit checks within two minutes of encountering the product for the first time”

• Choosing good fit criteria
• Stakeholders are rarely this specific
• The right criteria might not be obvious:
  – Things that are easy to measure aren’t necessarily what the stakeholders want
  – Standard metrics aren’t necessary what stakeholders want
• Stakeholders need to construct their own mappings from requirements to fit criteria
Some Quality Measures
Performance

• Performance measures
  – Response time, number of events processed/denied in some interval of time, throughput, capacity, usage ratio, loss of information, latency, workload...
  – Usually with probabilities, confidence interval
  – Can be modeled and simulated (mainly at the architectural level) – performance prediction

• Example
  – The system shall be able to process 100 payment transactions per second in peak load.
  – In standard workload, the CPU usage shall be less than 50%.
  – Production of a simple report shall take less than 20 seconds for 95% of the cases.
  – Scrolling one page up or down in a 200 page document shall take at most 1 second.
Some Quality Measures

Reliability

- Reliability Measures
  - Measure degree to which the system performs as required
  - Includes resistance to failure
  - Ability to perform a required function under stated conditions for a specified period of time
  - Very important for critical, continuous, or scientific systems
  - Can be measured using
    - Defect rate
    - Degree of precision for computations

- Examples
  - The precision of calculations shall be at least $1/10^6$.
  - The system defect rate shall be less than 1 failure per 1000 hours of operation.
Some Quality Measures
Availability

- Availability measures
  - Definition: Percentage of time that the system is up and running correctly
    - Probability that the system is up and running when needed!
  - Can be calculated based on Mean-Time to Failure (MTBF) and Mean-Time to Repair (MTTR)
    - MTBF: Length of time between failures
    - MTTR: Length of time needed to resume operation after a failure
    - Availability = MTBF/(MTBF+MTTR)

- Availability Downtime
  - 90% 36.5 days/year
  - 99% 3.65 days/year
  - 99.9% 8.76 hours/year
  - 99.99% 52 minutes/year
  - 99.999% 5 minutes/year
  - 99.9999% 31 seconds/year
Some Quality Measures

Usability

- **Learnability**: functionalities or tasks mastered after a given training time
- **Efficiency**:
  - Acceptable response time
  - Number of tasks performed or problems resolved in a given time
  - Number of mouse clicks needed to get to information or functionality
- **Memorability**: Number of learned tasks that can still be performed after not using the system for a given time period
- **Error avoidance**: Number of error per time period and user class

- Examples
  - The system should enable at least 80% of users to book a guest within 5 minutes after a 2-hour introduction to the system.
  - The system shall enable **novice** users to perform tasks X and Y in less than 15 minutes.
  - The system shall enable **expert** users to perform tasks X and Y in less than 2 minutes.
Many other NFRs

• What about NFRs such as “fun” or “cool” or “beautiful” or “exciting”? How can these be measured?

• The lists of existing quality attributes are interesting but they do not include all NFRs. It is sometimes better to let customers do their brainstorming before proposing the conventional NFR categories.

• In any case, we must also refine those goals into measurable requirements.
Software Quality with ISO/IEC 9126

Internal and External Software Quality Model (ISO/IEC 9126)

**Quality Characteristics**

- **Functionality**
  - Suitability
  - Accuracy
  - Interoperability
  - Security
  - Compliance

- **Reliability**
  - Maturity
  - Fault tolerance
  - Recoverability
  - Compliance

- **Usability**
  - Understandability
  - Learnability
  - Operability
  - Comp
  - Attractiveness

- **Efficiency**
  - Time behavior
  - Resource utilization
  - Compliance

- **Maintainability**
  - Analyzability
  - Changeability
  - Stability
  - Testability
  - Compliance

- **Portability**
  - Adaptability
  - Installability
  - Co-existence
  - Replaceability
  - Comp
Modeling and evaluating NFRs?

- Boehm’s NFR List

```
General utility
  As-is utility
    portability
    reliability
    efficiency
    usability
    testability
  Maintainability
    understandability
    modifiability

device-independence
self-containedness
accuracy
completeness
robustness/integrity
consistency
accountability
device efficiency
accessibility
communicativeness
self-descriptiveness
structuredness
conciseness
legibility
augmentability
```
Modeling and evaluating NFRs?

- McCall's NFR List

![Diagram showing the McCall's NFR List with subcategories for Product operation, Product revision, and Product transition, including categories like usability, integrity, efficiency, correctness, reliability, maintainability, testability, flexibility, reusability, portability, and interoperability.](image-url)
Modeling NFRs Using the NFR Framework
Chung, Nixon, Yu, Mylopoulos (1999)

- Goal types:
  - Non-functional Requirement
  - Satisficing Technique
    - e.g. a design choice
  - Claim
    - supporting/explaining a choice

- Contribution Types:
  - AND links (decomposition)
  - OR links (alternatives)
  - Sup links (supports)
  - Sub links (necessary subgoal)

- Evaluation of goals
  - Satisficed
  - Denied
  - Conflicting
  - Undetermined
REQUIREMENTS VERIFICATION AND VALIDATION
Software Verification vs Validation
Jezequel, Plouzeau, Le Traon, “Développement de logiciels à objets avec UML”, Poly ISTIC C119, 2005

• “Software is done. Now, we’re just trying to get it to work.”
  Anonymous

• Verification: “La vérification est l'ensemble des actions de revue, inspection, test, preuve automatique, ou autres techniques appropriées permettant d'établir et de documenter la conformité des artefacts du développement vis-à-vis de critères préétablis”.

  Verification aims to answer the question: “is the system being built right?”. 

• Validation: “La validation consiste à évaluer l'adéquation du système développé vis-à-vis des besoins exprimés par ses futurs utilisateurs.”

  Validation aims to answer the question: “is the right system being built?”. 
Requirements Engineering and V&V

- Do we have elicited, understood and specified ALL requirements?
- Have we reached a consensus among ALL stakeholders?
- Are there still conflicts remaining?
- Do we have specified ALL requirements in the RIGHT way? At the good level of details? With the good precision?
Cost of Requirements errors

Without validation during requirements engineering:
- Low effort for fixing errors in requirements.
- Effort increases significantly as we move from requirements to test.

With validation during requirements engineering:
- Initial medium effort for fixing errors.
- Effort decreases significantly as validation occurs in multiple stages, reducing overall cost.
Two Levels of Requirements V&V

- At the system level
  - Ensures that the software being developed (or changed) will satisfy its stakeholders (Validation)
  - Ensures that each step followed in the process of building the software yields the right products (Verification)

- At the Requirements specification level
  - Checks the software requirements specification against stakeholders goals and requirements (Validation)
  - Checks consistency of the software requirements specification artifacts and other software development products (design, implementation, ...) against the specification (Verification)
When to perform Requirements V&V?

- Need to be performed at every stage during the (requirements) process
  - Elicitation
    - Checking back with the elicitation sources
    - “So, are you saying that . . . . ?”
  - Analysis
    - Checking that the domain description and requirements are correct
  - Specification
    - Checking that the defined system requirement will meet the user requirements under the assumptions of the domain/environment
    - Checking conformity to well-formedness rules, standards…
6 Principles to Requirements Validation

• The aim is to ensure that as much as errors as possible are identified
• 6 principles
  – Involve with the correct stakeholders
  – Separate errors identification from their correction
    • Independent V&V
  – Validate requirements from different perspectives
    • Reach a consensus
  – Try alternative formalisms
    • Use cases to understand goals and interactions with actors
    • State machines for transitions analysis
    • Class diagrams for architecture concerns
    • May highlight ambiguity
  – Build development items from requirements
    • Test cases, user manual, architecture diagrams
  – Repeat the validation step (continuous validation)
Requirements V&V Techniques

- Simple checks
  - Traceability, well-written requirements
- Prototyping
- Functional test design
- User manual development
- Reviews and inspections
- Model-Based V&V
Simple Checks

- Various checks can be done using traceability techniques
  - Given the requirements document, verify that all elicitation notes are covered
  - Tracing between different levels of requirements
    - Checking goals against tasks, features, requirements...
  - Involves developing a traceability matrix
    - Ensures that requirements have been taken into consideration
      (if not, there should be a reason)
    - Ensures that everything in the specification is justified
- Verify that the requirements are well written
  (according to the criteria already discussed)
Prototyping

- Excellent for validation by users and customers
  - More accessible than specification
  - Demonstrate the requirements and help stakeholders discover problems
- Come in all different shapes and sizes
  - From paper prototype of a computerized system to formal executable models/specifications
  - Horizontal, vertical
  - Evolutive, disposable (more or less)
- Important to choose scenarios or use cases for elicitation session
Functional Test Design

• Functional tests at the system level must be developed sooner or later...
  – Can (and should) be derived from the requirements specification
  – Each (functional) requirement should have an associated test
  – Non-functional (e.g., reliability) or exclusive (e.g., define what should not happen) requirements are harder to validate with testing
  – Each requirements test case must be traced to its requirements
  – Inventing requirements tests is an effective validation technique

• Designing these tests may reveal errors in the specification (even before designing and building the system)!
  – Missing or ambiguous information in the requirements description may make it difficult to formulate tests

• Some software development processes (e.g., agile methods) begin with tests before programming ➔ Test-Driven Development (TDD)
User Manual

• Same reasoning as for functional test design
  – Has to be done at some point
  – Reveals problems earlier
• Forces a detailed look at requirements
• Particularly useful to describe functional requirements and detect problems
• Particularly useful if the application is rich in user interfaces / for usability requirements

• Typical information in a user manual
  – Description of the functionality
  – How to get out of trouble
  – How to install and get started with the system

• Careful: “The best designed products won't even need manuals.”
  [Don Norman, 2001]
Reviews and Inspections

• A group of people read and analyze requirements, look for potential problems, meet to discuss the problems, and agree on a list of action items needed to address these problems
• A widely used requirements validation technique
  – Lots of evidence of effectiveness of the technique
  – Standardisation: IEEE 1028 Standard for Software Review and Audits
• Can be expensive
  – Careful planning and preparation
  – Pre-review checking + appropriate checklists
• Different types of reviews with varying degrees of formality
  – Reading the document: A person other than the author of the document
  – Reading and approval (sign-off): Encourages the reader to be more careful
  – Walkthroughs
    • Informal, often high-level overview
    • Can be led by author/expert to educate others on his/her work
  – Formal inspections
    • Very structured and detailed review, defined roles for participants
Reviews Checklists

- Essential tool for an effective review process
  - List common problem areas and guide reviewers
  - May include questions on several quality aspects of the document: comprehensibility, redundancy, completeness, ambiguity, consistency, organization, standards compliance, traceability ...
- There are general checklists and checklists for particular modeling and specification languages
- Checklists are supposed to be developed and maintained

- See example on course website
Elements of a Checklist

- **Comprehensibility** – can readers of the document understand what the requirements mean?
- **Redundancy** – is information unnecessarily repeated in the requirements document?
- **Completeness** – does the checker know of any missing requirements or is there any information missing from individual requirement descriptions?
- **Ambiguity** – are the requirements expressed using terms which are clearly defined?
- **Consistency** – do the descriptions of different requirements include contradictions? Are there contradictions between individual requirements and overall system requirements?
- **Organisation** – is the document structured in a sensible way? Are requirements organised so that related requirements are grouped?
- **Conformance to standards** – does the requirements document and individual requirements conform to defined standards? Are departures from the standards justified?
- **Traceability** – are requirements unambiguously identified? Do they include links to related requirements and to the reasons why these requirements have been included?
Model-based (Formal) V&V

- Modeling paradigms
  - Entity-Relationship modeling – e.g. UML Class diagrams
  - Workflow modeling notations – there are many different “dialects”, such as UML Activity diagrams, UCM, BPML, Petri nets (a very simple formal model), Colored Petri nets
  - State machines – e.g. Finite State Machines (FSM – a very simple formal model), extended FSMs, such as UML State diagrams

- Formal approaches
  - First-order logic – notations such as Z, VDM, UML-OCL, etc.
    - Can be used as an add-on with the other paradigms above, by providing information about data objects and relationships (possibly in the form of “assertions” or “invariants” that hold at certain points during the dynamic execution of the model)
    - Can be used alone, expressing structural models and behavioral models (there are many examples of using Z for such purpose)
Operations on Models

- Available V&V techniques will vary from one modeling paradigms to another and will also depend on the available tools (that usually only apply to a particular “dialect” of the modeling paradigm).

- The following functions may be provided through tools:
  - Completeness checking: only according to certain syntax rules, templates.
  - Consistency checking: given model M, show that M does not imply a contradiction and does not have any other undesirable general property (e.g. deadlock possibility).
  - Refinement checking: given two models M and M’, show that the properties of M imply the properties of M’.
  - Model checking: given a model M and some properties P, show that any system implementation satisfying M will have the properties P.
  - Theorem proving: prove (through induction or other approach) the correctness and validity of theorems against a model, for all situations.