Lecture 19: Verification and Validation

→ Some Refreshers:
  % Summary of Modelling Techniques seen so far
  % Recap on definitions for V&V  
→ Validation Techniques  
  % Inspection (see lecture 6)  
  % Model Checking (see lecture 16) 
  % Prototyping
→ Verification Techniques  
  % Consistency Checking  
  % Making Specifications Traceable (see lecture 21)
→ Independent V&V

We’ve looked at the following non-UML diagrams:
  % Goal Models  
  % Fault Tree Models (an example risk analysis technique)  
  % Strategic Dependency Models (I*)  
  % Entity-Relationship Models  
  % Mode Class Tables, Event Tables and Condition Tables (SCR)

The story so far

→ We’ve looked at the following UML diagrams:
  % Activity diagrams  
  % Class Diagrams  
  % Statecharts  
  % Use Cases  
  % Sequence Diagrams (collaboration diagrams are similar)

→ Validation:  
  % “Are we building the right system?”  
  % Does our problem statement accurately capture the real problem?  
  % Did we account for the needs of all the stakeholders?

→ Verification:  
  % “Are we building the system right?”  
  % Does our design meet the spec?  
  % Does our implementation meet the spec?  
  % Does the delivered system do what we said it would do?
  % Are our requirements models consistent with one another?
Some distinctions:
- Domain Properties: things in the application domain that are true anyway
- Requirements: things in the application domain that we wish to be made true
- Specification: a description of the behaviors the program must have in order to meet the requirements

Two verification criteria:
- The Program running on a particular Computer satisfies the Specification
- The Specification, given the Domain properties, satisfies the Requirements

Two validation criteria:
- Did we discover (and understand) all the important Requirements?
- Did we discover (and understand) all the relevant Domain properties?

V&V Example

Example:
- Requirement R:
  - "Reverse thrust shall only be enabled when the aircraft is moving on the runway"
- Domain Properties D:
  - Wheel pulses on if and only if wheels turning
  - Wheels turning if and only if moving on runway
- Specification S:
  - Reverse thrust enabled if and only if wheel pulses on

Verification
- Does the flight software, P, running on the aircraft flight computer, C, correctly implement S?
- Does S, in the context of assumptions D, satisfy R?

Validation
- Are our assumptions, D, about the domain correct? Did we miss any?
- Are the requirements, R, what is really needed? Did we miss any?

Inquiry Cycle

Initial hypotheses
- Prior Knowledge (e.g. customer feedback)
- Observe (what is wrong with the current system?)
  - Look for anomalies - what can't the current theory explain?
- Intervene (replace the old system)
  - Design experiments to test the new theory
  - Create/define a better theory
- Build a Prototype
  - Get users to try it
  - Evaluate the prototype
- Model (describe/explain the observed problems)
  - Check properties of the model
  - Analyze the model

Shortcuts in the inquiry cycle

Prior Knowledge (e.g. customer feedback)
- Observe (what is wrong with the model?)
- Intervene (replace the model)
- Build a Prototype
  - Invent a better system
  - Analyze the model
  - Get users to try it
A software prototype is a partial implementation constructed primarily to enable customers, users, or developers to learn more about a problem or its solution. [Davis 1990]

Prototyping is the process of building a working model of the system [Agresti 1986]

→ Approaches to prototyping
- Presentation Prototypes
  - explain, demonstrate and inform - then throw away
  - e.g. used for proof of concept; explaining design features; etc.
- Exploratory Prototypes
  - used to determine problems, elicit needs, clarify goals, compare design options
  - informal, unstructured and thrown away.
- Breadboards or Experimental Prototypes
  - explore technical feasibility; test suitability of a technology
  - Typically no user/customer involvement
- Evolutionary (e.g. "operational prototypes", "pilot systems"):
  - development seen as continuous process of adapting the system
  - "prototype" is an early deliverable, to be continually improved.

→ Throwaway Prototyping
- Purpose:
  - to learn more about the problem or its solution;
  - discard after desired knowledge is gained.
- Use:
  - early or late
- Approach:
  - horizontal - build only one layer (e.g. UI)
  - "quick and dirty"
- Advantages:
  - Learning medium for better convergence
  - Early delivery → early testing → less cost
  - Successful even if it fails
- Disadvantages:
  - Wasted effort if reqts change rapidly
  - Often replaces proper documentation of the requirements
  - May set customers' expectations too high
  - Can get developed into final product

→ Evolutionary Prototyping
- Purpose:
  - to learn more about the problem or its solution...
  - ...and reduce risk by building parts early
- Use:
  - incremental; evolutionary
- Approach:
  - vertical - partial impl. of all layers;
  - designed to be extended/adopted
- Advantages:
  - Requirements not frozen
  - Return to last increment if error is found
  - Flexible
- Disadvantages:
  - Can end up with complex, unstructured system which is hard to maintain
  - Early architectural choice may be poor
  - Optimal solutions not guaranteed
  - Lacks control and direction

Brooks: "Plan to throw one away - you will anyway!"

→ Verification
- Is the model well-formed?
- Are the parts of the model consistent with one another?

→ Validation:
- Animation of the model on small examples
- Formal checks:
  - "if the model is correct then the following property should hold..."
- What if questions:
  - reasoning about the consequences of particular requirements;
  - reasoning about the effect of possible changes
  - "will the system ever do the following?"...
- State exploration
  - E.g. use a model checking to find traces that satisfy same property

→ Basic Cross-Checks for UML

Use Case Diagrams
- Does each use case have a user?
- Does each user have at least one use case?
- Is each use case documented?
- Using sequence diagrams or equivalent

Class Diagrams
- Does the class diagram capture all the classes mentioned in other diagrams?
- Does every class have methods to get/set its attributes?

Sequence Diagrams
- Is each class in the class diagram?
- Can each message be sent?
  - Is there an association connecting sender and receiver classes on the class diagram?
  - Is there a method call in the sending class for each sent message?
  - Is there a method call in the receiving class for each received message?

StateChart Diagrams
- Does each statechart diagram capture (the states of) a single class?
- Is that class in the class diagram?
- Does each transition have a trigger event?
- Is it clear which object initiates each event?
- Is each event listed as an operation for that object's class in the class diagram?
- Does each state represent a distinct combination of attribute values?
- Is it clear which combination of attribute values?
- Are all those attributes shown on the class diagram?
- Are there method calls in the class diagram for each transition?
  - a method call that will update attribute values for the new state?
  - method calls that will test any conditions on the transition?
  - method calls that will carry out any actions on the transition?
Independent V&V

→ V&V performed by a separate contractor
  ✪ Independent V&V fulfills the need for an independent technical opinion.
  ✪ Cost between 5% and 15% of development costs
  ✪ Studies show up to fivefold return on investment:
    > Errors found earlier, cheaper to fix, cheaper to re-test
    > Clearer specifications
    > Developer more likely to use best practices

→ Three types of independence:
  ✪ Managerial Independence:
    > separate responsibility from that of developing the software
    > can decide when and where to focus the V&V effort
  ✪ Financial Independence:
    > Costed and funded separately
    > No risk of diverting resources when the going gets tough
  ✪ Technical Independence:
    > Different personnel, to avoid analyst bias
    > Use of different tools and techniques

Some philosophical views of validation

→ logical positivist view:
  > "There is an objective world that can be modeled by building a consistent body of knowledge grounded in empirical observation"
  ✪ In RE, assumes there is an objective problem that exists in the world
  > Build a consistent model; make sufficient empirical observations to check validity
  > Use tools that test consistency and completeness of the model
  > Use reviews, prototyping, etc to demonstrate the model is "valid"

→ Popper’s modification to logical positivism:
  > "Theories can’t be proven correct, they can only be refuted by finding exceptions”
  ✪ In RE, design your requirements models to be refutable
  > Look for evidence that the model is wrong
  > E.g. collect scenarios and check the model supports them

→ post-modernist view:
  > "There is no privileged viewpoint; all observation is value-laden; scientific investigation is culturally embedded”
  ✪ In RE, validation is always subjective and contextualised
  > Use stakeholder involvement so that they own the requirements models
  > Use ethnographic techniques to understand the weltanschaugungen