ITI8520 Real-time Software Engineering

Lab 1. Model-based Software Engineering - Basic Process

Tõnu Näks[1,2] Andres Toom[2]

[1] Tallinn University of Technology
Department of Software Science
Laboratory of Proactive Technologies
<firstname>.<lastname>@ttu.ee

[2] IB Krates OÜ
Group of software tools
<firstname>@krates.ee

Spring term 2017

System/software development in a nutshell

In essence the development of a safety-critical/high-integrity software and systems involves the following sub-processes:

- Specification
- Design
- Implementation
- Verification

This materialises in a set of artifacts that need to be *consistent* and *maintainable*.
For example, the DO-178C avionic software guideline considers following main kinds of artifacts:

- Requirements
  - System Requirements Allocated to Software
  - Software High Level Requirements
  - Software Low Level Requirements
- Software Architecture
- Source Code
- Executable Object Code
- Test Cases ← What cases must be tested and what is the expected result?
- Test Procedures ← How exactly are the tests composed?
- Test Results

System/software development in a nutshell

In essence the development of a safety-critical/high-integrity software and systems involves the following sub-processes:

- Specification
- Design
- Implementation
- Verification

This materialises in a set of artifacts that need to be *consistent* and *maintainable*.

For example, the DO-178C avionic software guideline considers following main kinds of artifacts:

- Requirements
  - System Requirements Allocated to Software
  - Software High Level Requirements
  - Software Low Level Requirements
- Software Architecture
- Source Code
- Executable Object Code
- Test Cases ← What cases must be tested and what is the expected result?
- Test Procedures ← How exactly are the tests composed?
- Test Results

System/software development in a nutshell

In essence the development of a safety-critical/high-integrity software and systems involves the following sub-processes:

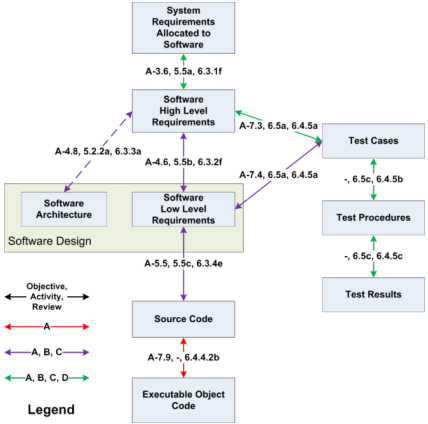
- Specification
- Design
- Implementation
- Verification

This materialises in a set of artifacts that need to be *consistent* and *maintainable*.
For example, the DO-178C avionic software guideline considers following main kinds of artifacts:

- Requirements
  - System Requirements Allocated to Software
  - Software High Level Requirements
  - Software Low Level Requirements
- Software Architecture
- Source Code
- Executable Object Code
- Test Cases ← What cases must be tested and what is the expected result?
- Test Procedures ← How exactly are the tests composed?
- Test Results

Traceability in system/software development

An important property that helps to achieve *consistency* between the various artifacts and their *maintainability* is *traceability*.



Steven H. VanderLeest `https://en.wikipedia.org/wiki/DO-178C`

About requirements

A good requirement should be:

- Concise
- Unambiguous
- Implementable
- Verifiable
- Traceable

About modelling - (A recap from the lecture)

Why do we model?

- To understand the problem and possibilities to solve it
- To communicate our understanding
- To communicate the planned solution
- To experiment with solutions
- To document the design/implementation
- To verify/validate the design

About modelling (2) - (A recap from the lecture)

Practical considerations for modelling:

- It may be cheaper to construct a model when compared to constructing the real system
- The systems are often too complex to process all aspects at once – to understand them we would like to concentrate a certain view at a time
- We may do not have access to the real system or can not experiment with it
- We may want to experiment in conditions that are not possible in real system (change the speed of process, try behaviour in broder conditions or different failure combnations etc.)
- Analytical reasoning is often possible with limited (and well defined) set of properties

- Furthermore, designs expressed in formally defined modelling languages become amenable to *automated analysis* and *transformations*. Potentially, also to the *automatic generation* of software code.

About modelling (2) - (A recap from the lecture)

Practical considerations for modelling:

- It may be cheaper to construct a model when compared to constructing the real system
- The systems are often too complex to process all aspects at once – to understand them we would like to concentrate a certain view at a time
- We may do not have access to the real system or can not experiment with it
- We may want to experiment in conditions that are not possible in real system (change the speed of process, try behaviour in broder conditions or different failure combnations etc.)
- Analytical reasoning is often possible with limited (and well defined) set of properties
- Furthermore, designs expressed in formally defined modelling languages become amenable to *automated analysis* and *transformations*. Potentially, also to the *automatic generation* of software code.

Project definition

- The task is to develop software for a *traffic light system* (demonstrator).
- The development process shall use model-based principles and workflow.
- The high-level description of the task (*vision*) and high-level requirements (HLR) will be discussed during the class and published after it.
- Design, low-level requirements (LLR), test cases (TC), test procedures (TP), software implementation and verification shall be performed individually.

Technical aspects

For this exercise we shall use a following largely model-based process:

- Software High Level Requirements – Text or similar
- Software Architecture – UML Model
- Software Low Level Requirements – UML Model. Complemented with text, if needed
- Test Cases – UML Model. Complemented with text, if needed
- Implementation
  - Ecore model – Automatically generated from the UML model.
  - Source code – Automatically generated from the Ecore model. Complemented with manually written Java code
- Test Procedures – Harness and test stubs generated from the Ecore model. Complemented with manually written Java code
- Test Results and verification reports – Semi-automatic

Technical aspects

For this exercise we shall use a following largely model-based process:

- Software High Level Requirements – Text or similar
- Software Architecture – UML Model
- Software Low Level Requirements – UML Model. Complemented with text, if needed
- Test Cases – UML Model. Complemented with text, if needed
- Implementation
  - Ecore model – Automatically generated from the UML model.
  - Source code – Automatically generated from the Ecore model. Complemented with manually written Java code
- Test Procedures – Harness and test stubs generated from the Ecore model. Complemented with manually written Java code
- Test Results and verification reports – Semi-automatic

Technical aspects

For this exercise we shall use a following largely model-based process:

- Software High Level Requirements – Text or similar
- Software Architecture – UML Model
- Software Low Level Requirements – UML Model. Complemented with text, if needed
- Test Cases – UML Model. Complemented with text, if needed
- Implementation
  - Ecore model – Automatically generated from the UML model.
  - Source code – Automatically generated from the Ecore model. Complemented with manually written Java code
- Test Procedures – Harness and test stubs generated from the Ecore model. Complemented with manually written Java code
- Test Results and verification reports – Semi-automatic

Technical aspects

For this exercise we shall use a following largely model-based process:

- Software High Level Requirements – Text or similar
- Software Architecture – UML Model
- Software Low Level Requirements – UML Model. Complemented with text, if needed
- Test Cases – UML Model. Complemented with text, if needed
- Implementation
  - Ecore model – Automatically generated from the UML model.
  - Source code – Automatically generated from the Ecore model. Complemented with manually written Java code
- Test Procedures – Harness and test stubs generated from the Ecore model. Complemented with manually written Java code
- Test Results and verification reports – Semi-automatic

Obtaining EMF and Papyrus UML

- Make sure you have Java 8 installed and available. Preferably, Oracle JDK (`http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`)
- Go to: `http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/neon2`
- Download the appropriate version for your OS
- Unpack and launch the eclipse executable
- Choose a suitable workspace location and close the welcome screen
- Install following additional modelling components via Help > Install Modeling Components:
  - Papyrus
- Relaunch Eclipse
- Install following additional Papyrus components via Help > Install Papyrus Additional Components:
  - MARTE, SysML, Papyrus for Requirements, Designer-JAVA, Designer-CPP, Papyrus Compare, Integrated ALF Editor
- Pre-bundled package for Windows-64 is available here: `https://www.dropbox.com/s/ejvl4lp8y4287ix/eclipse-modeling-neon-2-win32-x86_64_papyrus_ext.zip?dl=0`

Modelling in Papyrus UML

- The task is to model the system's architecture and low level requirements in Papyrus UML.
- You don't need the full power of UML for that.
- It is probably sufficient to
  - Define the logical architecture of the software components (class diagram), including the main functional operations
  - Create sequence diagrams for the main operational scenarios
- If you are unfamiliar with Papyrus, please consult the following documents and tutorials
  - `https://eclipse.org/papyrus/documentation.html`
  - `https://eclipse.org/papyrus/resources/TutorialOnPapyrusUSE_d20101001.pdf`
  - `http://www.eclipse.org/papyrus/resources/PapyrusTutorial_OnSequenceDiagrams_v0.1_d2010100.pdf`
- Hint: When creating a new project, choose a profile that already contains the standard datatypes (booleans, integers etc.).

Working with EMF and Ecore

- Ecore is the central “core” language in EMF.
- Being a core language, its scope is a much narrower language than that of UML.
- Broadly, it is intended for modelling logical data relations, similarly to class diagrams in UML.
- Hence, UML class diagrams, database schemas etc. can be rather easily mapped to Ecore.
- Good tutorials about the EMF are e.g. here:
  - `http://eclipsesource.com/blogs/tutorials/emf-tutorial/`
  - `http://www.vogella.com/tutorials/EclipseEMF/article.html`
- For our current task, please create a new project in Eclipse that will contain the Ecore version of your UML model. When creating the project, choose the “Empty EMF Project” wizard.
- In the new project, please go to the “model” folder and choose New > Other > EMF Generator Model.
- Give the model a name like <YourUMLModel>.genmodel. version of your UML model. When creating the project, choose the “Empty EMF Project” wizard.
- Locate the UML model on the next screen.
- Continue and select the root packages to import on the next screen.
- That should suffice to create the <YourUMLModel>.ecore.

Assignment - To submit

- Summary report (odt, word, pdf).
  - Containing:
    - Vision
    - Requirements (HLR, LLR, Architecture)
    - Test Case descriptions
    - Test Procedure descriptions
    - Coverage report
    - Verification reports
    - (etc.)
  - Submit to: Moodle (https://ained.ttu.ee)
  - File name: <surname>_lab<lab_number>_summary_report.<odt, doc, etc>
- Models, project sources and configuration files
  - Submit to: Git (https://git.ttu.ee) – *Details to be announced*
- *(Optional)* Other documentation (detailed verification reports, etc.).
  - Submit to: Git (https://git.ttu.ee) – *Details to be announced*
- Deadline: 15.03 (Week 7).

Assignment - A tip for organising the work and reporting

The “**ODSI**” principle:

- **O**rganize yourself in your own way
- **D**ocument how you have organized yourself
- **S**ubmit this document to your customer for approval
- **I**mplement this organization (once approved)

Borrowed from the recommendation (rule) for complying with the European Space Agency (ESA) standards in actual projects.