LAB 1:
Requirements Modeling
Preparations and instructions

Björn Regnell

October 11, 2014

1 Introduction

1.1 Purpose

This document provides instructions on how to prepare for and run a computer lab session on requirements modeling. The lab session illustrates how computer-supported requirements modeling can help in requirements engineering, including elicitation, specification and validation. *The preparations in this document should be completed before the actual lab is run.*

1.2 Background

Requirements engineering is a dynamic process where knowledge and perception of an imagined future system (of systems) is evolving over time. During this evolution we can capture the knowledge and creative ideas we elicit in various ways, depending on how we imagine the (later) usage of this information: we could create and use, for example, spread sheets, post-it notes, emails, wikis, video clips, mockups, sketches, diagrams, mathematical specifications, etc.

If we want to keep track many different types of inter-related requirements and if we see a future benefit of more structure beyond just a flat list, we can use *requirements models* where requirements are expressed using relevant (meta-) **entities**, **attributes** and **relations** that can capture what we want to model.

The open source tool **reqT** enables scalable requirements modeling, ranging from small models of a couple of features to large models containing elaborate structures of thousands of requirements. In this lab you will learn how to get started with reqT and reflect on how you could use reqT in your own project.

2 Preparations

Before the lab session, please complete all preparations below and bring requested items to the lab. In particular you need to make sure that you can access the text files you prepare below at your lab session computer.

1. **Read reqT intro.** Read the section "Introduction to reqT" on this web page:
   `http://reqt.org/documentation.html#intro`

2. **Do the reqT "hello world".** Run the reqT hello world example on your computer, as explained here:
   `http://reqt.org/documentation.html#hello`

3. **Draw a context diagram.** Draw a context diagram that is relevant to your development project. Make the drawing on paper and bring to the lab. The context diagram should include entities outside of the system you are building that are communicating directly with the system under development.

4. **Create a context model.** Transform your context diagram into a reqT model, analogous to this example:

   ```
   Model(
     Product("hotelApp") interactsWith (
       User("receptionist"),
       User("guest"),
       System("telephony"),
       System("accounting")))
   ```

   Saved the model in a text file called `context.scala`

5. **Create a list of features.** Make a list of at least 5 features relevant to your project, divided into 2 sections, analogous to the example below. Use the exact formatting and indentation according to the example with hash tag before the name of each group and an asterisk before the name of each feature:

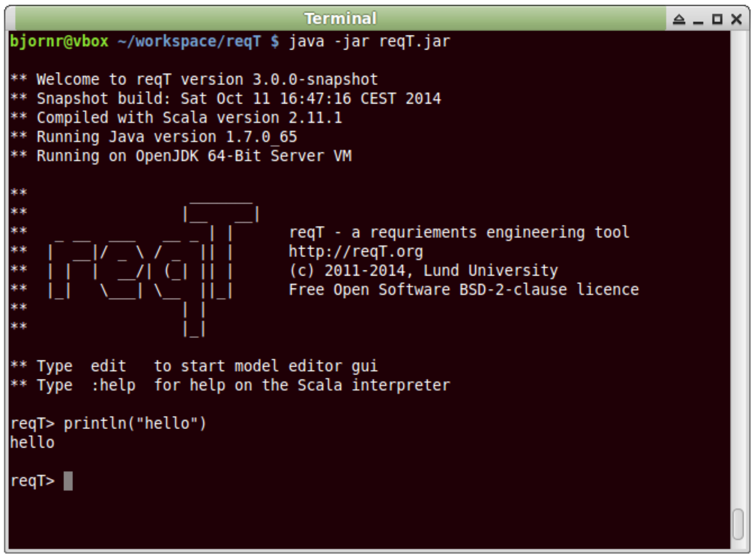
   ```
   # receptionFeatures
     * groupCheckIn
       Check in many guests arriving in bulk.
   # adminFeatures
     * roomForecast
       Calculate room allocation prediction.
     * exportAccounting
       Export accounting data to external system.
   ```

   Saved the list of features in a text file called `feat.txt`

3 Lab instructions

3.1 Start reqT

Download the reqT.jar file from `http://reqT.org` and start reqT in a terminal window using this command: `java -jar reqT.jar` as shown in Figure 1.



```
bjornr@vbox ~/workspace/reqT $ java -jar reqT.jar

** Welcome to reqT version 3.0.0-snapshot
** Snapshot build: Sat Oct 11 16:47:16 CEST 2014
** Compiled with Scala version 2.11.1
** Running Java version 1.7.0_65
** Running on OpenJDK 64-Bit Server VM

**
**                  _______
**                 |__   __|
**   _ __ ___    __ _ | |      reqT - a requriements engineering tool
**  | '__/ _ \ / _`  || |      http://reqT.org
**  | |  |  __/| (_| || |      (c) 2011-2014, Lund University
**  |_|   \___| \__  ||_|      Free Open Software BSD-2-clause licence
**                 | |
**                 |_|

** Type  edit   to start model editor gui
** Type  :help  for help on the Scala interpreter

reqT> println("hello")
hello

reqT> 
```

Figure 1: The reqT console output after starting reqT in a terminal window.

3.2 Create and update models using the reqT console

Type in the following lines in the reqT console after the `reqT>` prompt. Press enter after each line. The + operator is used to add elements to a model, and the ++ operator is used to append one model to another.

```
reqT> val m1 = Model(Req("a") has Spec("sss"))

reqT> val m2 = m1 + (Req("b") has Prio(2))

reqT> val m3 = Model(Stakeholder("x") requires Req("a"))

reqT> ((m3 + Stakeholder("y")) ++ m2).size
```

Q1: What is the integer value result of the last evaluation?
`((m3 + Stakeholder("y")) ++ m2).size`

____

Continue to type in the following lines in the reqT console after the `reqT>` prompt. Press enter after each line. The `for` keyword is used to make a for-loop. The `yield` keyword is used in a for-comprehension to construct a sequence of values. The `val` keyword is used to declare a name that refers to an immutable value (a constant) and the `var` keyword is used to declare a name that refers to a mutable value (a variable). The − operator is used to remove elements from a model. With the `transform` method you can make transformations of specific elements in a model.

```
reqT> var m4 = (for (i <- 1 to 10) yield Req("r"+i)).toModel

reqT> (1 to 10).map(i => Req("r"+i)).toModel //alternative to above

reqT> m4 = Model(Stakeholder("x") requires m4)

reqT> m4 = m4 - Req("r7")

reqT> m4 -=  Req("r3")

reqT> m4.pp   //pretty-print m4

reqT> m4 = m4.transform{case Req(id) => Feature(id) has Status(ELICITED)}
```

Q2: What is the size of the m4 model after the above transform?
`m4.size`

____

3.3 Investigate the reqT metamodel

A reqT model can be viewed as a vector of elements. Elements can be entities, attributes and relations. An entity has an id of type String. An attribute holds a value that can be of different types. A relation connects and entity via a link of a certain RelationType to a submodel that, in turn, can contain elements. A part of the reqT metamodel is shown in Figure 2.

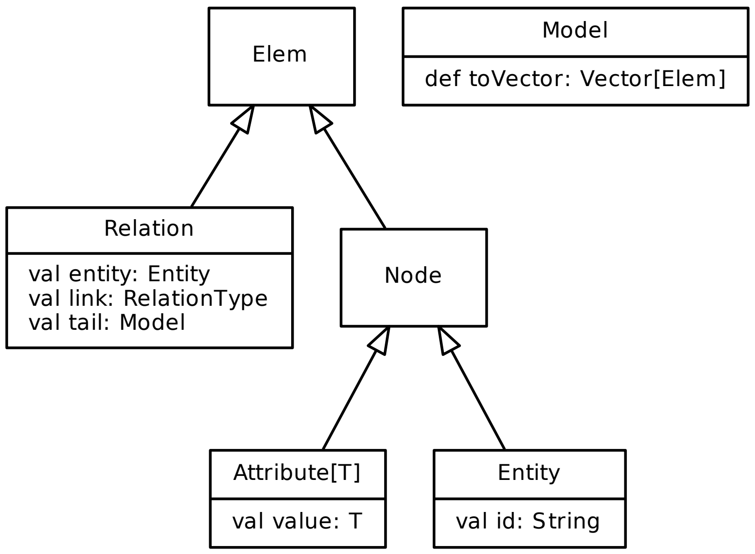


Figure 2: Some classes in the reqT metamodel.

Investigate what different entity types, attribute types and relation types that the reqT metamodel contains, using the evaluations in the reqT console below. The collect method collects selected parts of a model into a vector. In the last evaluation below we collect the integer 1 for each occurrence of a Meta entity and sum all ones.

```
reqT> reqT.metamodel.  // Press <TAB> after the dot

reqT> reqT.metamodel.ent // Press <TAB> after the t

reqT> reqT.metamodel.entityTypes

reqT> reqT.metamodel.entityTypes.size

reqT> reqT.meta.model.pp

reqT> reqT.meta.model.collect{case Meta(_) => 1}.sum
```

Q: How many different entity types, attribute types and relation types are there respectively in the reqT metamodel?

___

Q: How many `Meta` entity concepts are there in the reqT metamodel?

___

The meta model elements can be used in many different ways and there are no special restrictions on how to combine the elements, except for these three basic rules:

1. A model or submodel can only contain one attribute of a specific type at its top level. However, the same type of attribute can in different submodels. Try this in the reqT console:

```
reqT> Model(Spec("a"), Spec("b"))
res1: reqT.Model = Model(Spec("b"))

reqT> Model(Req("x") has (Spec("a"), Spec("b")))
res2: Model(Req("x") has Spec("b"))

reqT> Model(Req("x") has Spec("a"), Req("y") has Spec("b"))
res3: Model(Req("x") has Spec("a"), Req("y") has Spec("b"))
```

2. A model or a submodel can only contain one entity with a certain id and a certain relation at its top level. If you add an entity with the same id and the same relation on the top level of a model or submodel, it will merge the elements of each submodel, and if the above rule applies attributes will be overwritten if of the same type. Try this in the reqT console:

```
reqT> val m6 = Model(Req("x") has (Req("s1"), Spec("a")))

reqT> m6 + (Req("x") has (Req("s2"),Spec("b")))
res4: Model(Req("x") has (Req("s1"), Req("s2"), Spec("b")))
```

3. The has-relation is special, as an entity with no relation is equivalent to an entity with a has-relation to an empty submodel. Try this in the reqT console:

```
reqT> Model(Req("x") has ())
res5: reqT.Model = Model(Req("x"))

reqT> Model(Req("x") has Spec("x")) - Spec("x")
res6: reqT.Model = Model(Req("x"))
```

4 Additional quests if more time is available

```
m => m.transform{case Item(i) => Feature(i); case Text(i) => Gist(i)}
_.transform{case a: Attribute[_] => NoElem}
javax.swing.JOptionPane.showMessageDialog(null,"Hello Swing!")
```

5 Conclusion