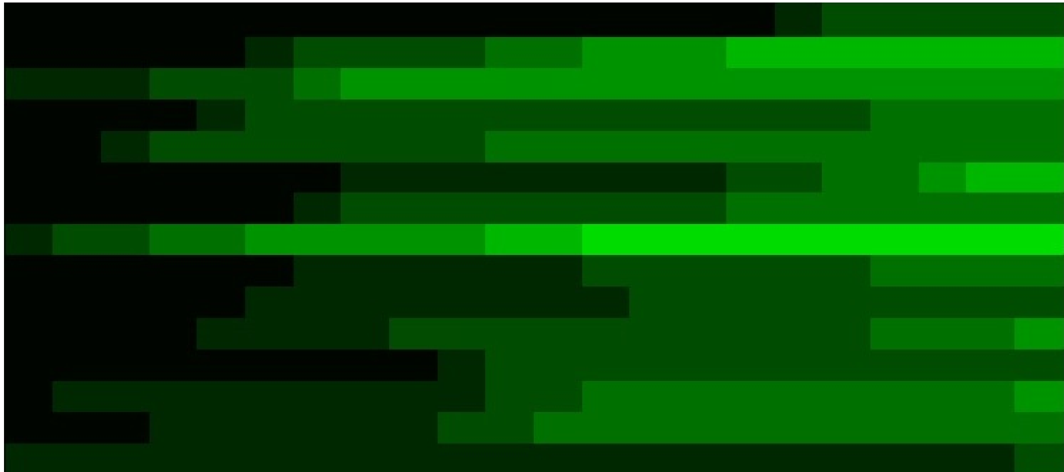
Model Software Requirements with reqT



Björn Regnell

INITIAL INCOMPLETE DRAFT November 5, 2013

Model Software Requirements with reqT

Björn Regnell

DRAFT - version 0.1 November 5, 2013

© 2013 Björn Regnell, Lund, Sweden.

This is a draft manuscript.
All rights reserved.
Copyright © Björn Regnell. 2013. Lund. Sweden.
Please send suggestions for improvement to bjorn.regnell@cs.lth.se

Contents

1 Introduction **1**
- 1.1 What is requirements engineering? . . . 1
- 1.2 The abstract ”requirement” . . . 2
- 1.3 Specification quality . . . 3
- 1.4 Requirements language . . . 4
- 1.5 reqT – a free requirements tool . . . 5

2 Create models **7**
- 2.1 Requirements as graphs . . . 7
- 2.2 Modeling goals . . . 9
- 2.3 Modeling data . . . 10
- 2.4 Spec+Why+Example . . . 11
- 2.5 Context diagram . . . 12
- 2.6 Quality requirements . . . 13
- 2.7 Deep models . . . 14

3 Compute models **15**
- 3.1 Model operators . . . 15
- 3.2 Modeling decision status . . . 17

4 Interoperate with other tools **19**
- 4.1 Save and load reqT models in Scala . . . 19
- 4.2 Model visualization with GraphViz . . . 20
- 4.3 Generate txt documents . . . 20
- 4.4 Generate html documents . . . 20
- 4.5 Export and import spreadsheet tables . . . 20

5 A Catalogue of Concepts **21**
- 5.1 Entities . . . 21
  - 5.1.1 Context Entities . . . 21
  - 5.1.2 Requirements Entities . . . 21
- 5.2 Relations . . . 23
- 5.3 Attributes . . . 25
  - 5.3.1 String value attributes . . . 25
  - 5.3.2 Integer value attributes . . . 25
  - 5.3.3 Special attributes . . . 25
- 5.4 The reqT metamodel . . . 26

A Scala by example **29**
A.1 Object-oriented Scala 29
A.2 Functional Scala 30

Preface

reqT is a free software requirements engineering tool developed with the goal to provide a flexible and scalable mix of natural-language requirements and graph-oriented data structures in an executable language. With reqT your requirements become code.

The development of reqT started at Lund University in Sweden 2011, and is used in software engineering education at MSc level.

For information on how to download, install and run reqT, please visit `http://reqT.org`

```
//reqT Scala-embedded DSL example 1-reqT-gist.scala

val m = Model(
  Product("reqT") has Gist(
    """An extensible requirements engineering tool
      |that flexibly combines natural language requirements
      |with graph-like structures into computable models."""
      .stripMargin),
  Product("reqT") helps Goal("beMasterOfRequirements")
)

m.toTxt.save("myModel.txt")
```

```
// file myModel.txt with the reqT/txt external DSL

MODEL(
  PRODUCT reqT HAS
    GIST: An extensible requirements engineering tool
      that flexibly combines natural language requirements
      with graph-like structures into computable models.

  PRODUCT reqT -- helps
    --> GOAL beMasterOfRequirements
)
```

Chapter 1

Introduction

1.1 What is requirements engineering?

Imagine that you are an experienced software engineer. You are working in an organization that provides value to users through its successful software-based products. The software is rapidly evolving through new and innovative features with the aim to further increase user value and stay ahead of competition. You have the responsibility to decide about which new software features to create for future releases. *How would you decide what software to create next?*

This is a key question in software engineering and the answer impacts the benefit of the resulting software innovations as well as the cost of their development. It is not an easy question and there is not a single, perfect answer on how to do good software engineering decision making. The answer depends on the context: the product, the market, the competitors, the abilities of the software engineers, the financial situations, etc.

When deciding what software to create next you are doing **requirements engineering**, or RE for short. The RE discipline includes a number of different activities that often are carried out iteratively and in parallel, throughout a system's life cycle. Central activities of RE are:

- **Elicitation**: the activity of generating candidate requirements and knowledge about their context. This involves answering questions such as: What are the needs of current and future users? How to create innovative (i.e. novel and valuable) ideas for future software development? When have we acquired enough knowledge to be able to make a good decision?

- **Specification**: the activity of documenting (candidate) requirements and all the information related to them (considered cost-efficient to document). To carry out this activity you typically need to answer these questions: In what form should you specify requirements? How detailed should the specifications be (amount of context information, level of abstration etc.)? Who will use the specifications and for how long will they persist in relevance?

- **Validation**: the activity of checking that (the documented representations of) requirements are good enough. Will these requirements lead to successful software? Is this specification useful for down-stream development activities including software design and testing?

- **Prioritization**: the activity of assessing candidate requirements based on criteria such as benefit, cost, risk and urgency. Which set of requirements will generate most benefit in a certain time frame in relation to their cost? How often should we re-assess our priorities? How to handle uncertainty?

- **Selection**: the activity of deciding which requirements to implement when, while taking into account priorities as well as inter-dependencies, resource constraints and timing issues. When is it best to invest a certain amount of development effort on a certain set of requirements? How often should we release new versions of our software? What is the best trade-off between investments in features that promise short-term profitability and investments in a robust software architecture for the long-term sustainability of our product?

These activities depend on each other and are often carried out in parallel and in an iterative manner. While you check your requirements (validation) you may come up with ideas for new features (elicitation) that need to be documented so that you don't forget about them (specification). The quality of the output of the selection activity rely on the quality of the other activities and you may find that it is impossible to arrive at a good release plan that schedules a justifiable subset of requirements without doing some more elicitation.

1.2 The abstract "requirement"

The term **requirement** is in software engineering often used in a very general sense, capturing the abstract notion of some kind of entity that may represents any potential subject of software implementation. There are many different conceivable requirements in any particular context and only some of them are implemented and used in executing software. Sometimes requirements are expressed as high-level goals or abstract needs of current or future stakeholders, and sometimes requirements are expressed as low-level, specific software solutions, colorful user interface designs or detailed technical capabilities in relation to current standards.

A requirement in this abstract sense can, for example, be:

- an unarticulated **wish**
- an urgent **must**
- an exciting **idea**
- a pressing **need**
- a postponed **feature** wanted later
- a mock-up screen **design**

- a **data** entity to be stored in the system's database
- a **function** operating on input data to produce some output data
- a cross-cutting **quality** aspect
- a challenging scheduling **constraint** on a set of features

All of these different forms of requirements, and many others, may have their justification depending on the context. Are you developing web content management support software in an open source project or a proprietary software to be used in a nuclear power plant you probably have very different approaches to how requirements are elicited, specified, validated, prioritized and selected. The context determine available resource and time constraints for requirements engineering, as well as the required quality of the resulting artifacts of requirements engineering such as product road-maps, release plans and validated feature specifications.

1.3 Specification quality

Figure 1.1 illustrates how the specification quality, in terms of e.g. completeness, may vary over time. As elicitation, specification, validation, prioritization and selection progresses iteratively over time, some requirements get more attention than others and therefore evolve faster in terms of specification quality. This reflect the fact that there are differences in risk of misinterpretation for different requirements and some can stay rather incomplete and ambiguous and still provide a good-enough basis for decision making and down-stream design.

Figure 1.1: An illustration of how the specification quality of a set of requirements can vary over time. Some requirements may improve faster in quality compared to others. The lighter areas represent higher specification quality.

1.4 Requirements language

Central to all the activities of elicitation, specification, validation and prioritization is the **requirements language** used to express the data, information and knowledge in the RE process. Ideally you would like a language that can express anything expressible in a complete an unambiguous way that is understandable to any stakeholder, but in practice you cannot get all these wishes to a reasonable cost. A high degree of completeness can in some cases require unjustifiable effort in elicitation and specification. A low degree of ambiguity may require a restriction of terminology to well-defined terms with one stipulated interpretation, but at the same time such restrictions limit what can be expressed.

The language you express requirements in should ideally be suitable in many different situations. You may want to discuss requirements in an online user forum or stipulate a template for requirements filed by users in an issue tracking system. You may want to version control your selected requirement in a data base or give examples of requirements inter-dependencies on a presentation slide when you show a product road-map to a steering committee of investors. It is likely that you need to adapt the language in which you express requirements to the particular context in which they are interpreted and make different trade-offs of completeness, ambiguity and expressiveness depending on who is interpreting your requirements descriptions and how much time you have available to formulate your requirements.

It is very likely that what is a good trade-off between completeness and ambiguity may be different for different subsets of requirements and also different at different points in time. Thus you may, at a given point in time, have some parts of your requirement repository that are expressed in comparably incomplete and ambiguous language as it may not be worth while to go further, while other parts are more elaborate and more carefully expressed as they may be more critical with respect to avoiding risks of misinterpretation.

In practice, requirements are often expressed in natural language, e.g. in English. The reason that natural language is so common in requirements engineering is that it has great flexibility and high expressive power and, if used carefully, natural language is often easy to understand by users familiar to the software application domain. However, natural language is often ambiguous and its flexibility allows different persons to express the same thing in may different ways. As the requirements engineering progresses, domain-specific terms arise and get a special meaning to the people involved in the process. Terms may thus get preciser definitions over time and thereby the risk of misinterpretation can be reduced.

There is often need for both expressing very precise aspects, such as the cardinality of data relations in a database model, to very 'fluffy' things such as the feeling that the user experiences when a new game level is reached. Thus, it can often be useful in RE to combine informal and formal language using different forms of expression (text, images, hypermedia, etc.).

Computer tools are heavily used in real-world RE. Often a mix of tools

are applied, such as word processors, spreadsheet programs, databases, issue trackers, configuration management systems, drawing tools, etc. A dedicated RE tool can provide specific support for managing requirements, such as keeping track of links among requirements and visualizing of priorities. There are nowadays numerous commercial RE tools available, but many are expensive, complex and not sufficiently open [2].

1.5 reqT – a free requirements tool

The open source tool called reqT [1] is developed with the particular aim to provide a flexible and scalable combination of informal text and formalized structure. The development of reqT started at Lund University in Sweden 2011, and is used in software engineering education at MSc level [6].

In reqT, a set of essential RE concepts can be captured in a language that combines natural language text into computational entities. With reqT, requirements are turned into code, that when executed creates a run-time requirements data structure that can be manipulated using scripts.

The following main goals and associated design strategies are the basis for the development of reqT:

1. **Semi-formal.** *Goal:* Provide a semi-formal representation of typical requirements modeling constructs that can illustrate a flexible combination of expressive natural language-style requirements with type-safe formalisms allowing static checks. *Design:* Use graph structures based on typed nodes representing typical requirement entities and attributes, and typed edges representing typical requirements relations, and implement the graph as an associative array (map). *Why?* Graphs are well-known to many software engineers. Maps are efficient from an implementation perspective and may be less complex to master compared to e.g. SQL databases.

2. **Open.** *Goal:* Provide a platform-independent requirements tool that is free of charge. *Design:* Use Java Virtual Machine technology and release the code under an open source license. Use tab-separated, tabular text-files for import and export. Use HTML for document generation. *Why?* There are many free libraries available that runs on a JVM. Tab-sep and HTML support interoperability.

3. **Scalable.** *Goal:* Provide an extensible requirements modeling language that can scale from small, concise models to large families of models with thousands of requirements entities and relations. *Design:* Implement reqT as an embedded DSL (Domain-Specific Language) [8] in the Scala programming language [5]. Use Map and Set from Scala collections to represent requirements graphs. *Why?* Scala is a modern, statically typed language with an efficient collections library. Scala offers scripting abilities that provide general extensibility without recompilation. Integrated development environments [7], as well as in-

teractive scripting tools are available [3]. When requirements are expressed as code, general software engineering tools can be applied when scaling to large projects, e.g. configuration management tools and cloud-based code repositories to facilitate requirements evolution and collaboration over Internet.

Chapter 2

Create models

The reqT tool includes a DSL for requirements modeling [1, 6], allowing domain experts to specify and analyze requirements. The metamodel of the reqT DSL aims to provide RE-specific concepts that give flexible expressiveness to domain experts by allowing a mix of informal natural language text and graph-oriented formalizations of typed requirements entities, attributes and relations. The use of a DSL allows requirements to be represented as textual, computational entities that can be stored together with production code and test scripts in a common version management system.

By embedding the DSL [8] into Scala, reqT can utilize Scala's collection library. This enables domain experts to combine their model specifications with Scala scripts that manipulate their models using existing collection operations. Models can also be traversed for various semantical checks, e.g. to investigate cycles and specific combinations of attributes etc.

2.1 Requirements as graphs

A requirements model in reqT can be constructed using a sequence of triplets:

<Entity> <Edge> <NodeSet>

Such a triplet can e.g. be constructed by connecting, for example, a feature entity with a list of two attributes, e.g. a gist and a spec, via the edge *has*, using the following syntax:

```
Feature("hello") has (
  Gist("print greeting"),
  Spec("Send 'hello' to the console.")
)
```

The feature above has a string-valued identifier "hello", and each of the attributes gist and spec has a boxed string value. If only one attribute is connected to an entity, there is no need for the extra parenthesis pair, as in this example:

```
Feature("hello") has Gist("print greeting")
```

By boxing comma-separated triplets like this: `Model(`*<list of triplets>*`)` a collection of requirements is stored in a model. The use of three such triplets collected in a model is exemplified in Listing 2.1.

```
Model(
  Feature("F1") has (
    Spec("The system shall..."),
    Status(IMPLEMENTED)
  ),
  UserStory("US1") has (
    Spec("As a user I want..."),
    Status(ELICITED)
  ),
  UserStory("US1") requires Feature("F1")
)
```

Listing 2.1: A model with a feature and a user story. Entities are called **sources** if the stand to the left of an edge (has edge or relation edge). Entities to the right of an edge are called **destinations**.

The *has* edge connects **source entities** with **destination attributes**, while other edges are relations that connect source entities with (a set of) **destination entities**, such as the *requires* edge in Listing 2.6. Other examples of relations between entities are: owns, excludes, helps, hurts and preceedes.

The model has one requires relation between the two entities UserStory US1 and Feature F1, where the former has the attributes Gist and Status, while the latter has Spec and Status attributes. Entities, relations and attributes can be flexibly connected using the concepts of the reqT metamodel, depicted in Figure 5.1 on page 27.

A model can be visualized using GraphViz [**?**]. When calling the method `toGraphViz` on a model, a dot language [**?**] export of the model is generated that can be rendered using GraphViz as shown in Figure 2.1.

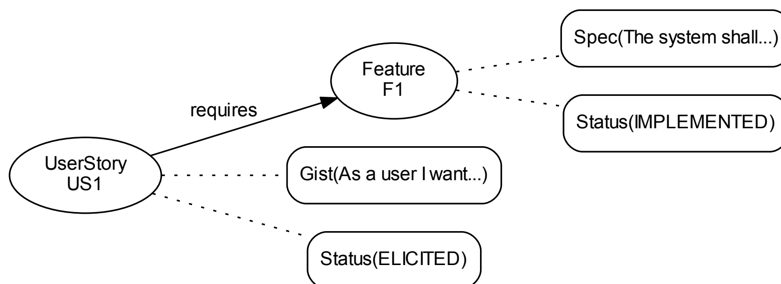


Figure 2.1: The corresponding graph of the example in Listing 2.6. Oval nodes represent entities. Rounded rectangles represent attributes. Solid arrowed lines represent typed relation edges. Dashed lines represent edges to attributes.

2.2 Modeling goals

```
Model(
  Product("shipyardQuoting") has
    Spec("Support cost recording and quotation with experience data."),
  Goal("precalcAccuracy") has
    Spec("Our pre-calculations shall hit within 5%."),
  Product("shipyard quoting") helps Goal(" precalcAccuracy ")
)
```

Listing 2.2: A small goal model.

2.3 Modeling data

```
Model(
  Data("roomDB") has Image("roomDB-ER-diagram.png"),
  Class("Guest") relatesToOneOrMany Class("Stay"),
  Class("Stay") relatesToOneOrMany Class("RoomState"),
  Class("Room") relatesToOneOrMany Class("RoomState"),
  Class("RoomState") has Image("room-state-diagram.png")
)
```

Listing 2.3: Modeling cardinality of relations.

```
Model(
  Data("createGuest") has (
    Spec(
      "The product shall store guest data according to the
      virtual window 'create guest data'."),
    Image("create-guest-data.png")
  )
)
```

Listing 2.4: Virtual Window example.

```
Model(
  Class("Guest") has (
    Spec(
      "The guest is the person or company who has to pay the bill.
      A guest has one or more stay records. 'Customer' is a synonym
      for guest but in the database we only use 'guest'. The persons
      staying in the rooms are also called guests but are not guests
      in database terms."),
    Example(
      "(1) A guest who stays one night.
       (2) A company with employees staying now and then, each of
           them with his own stay record where his name is recorded.
       (3) A guest with several rooms within the same stay.")
  ),
  Member("name") has
    Spec(
      "Text, 50 chars. The name stated by the guest. For companies the
      official name since the bill is sent there. Longer names exist but
      better truncate at registration time than at print out time."),
      Member("passport") has
    Spec(
      "Text 12 chars. Recorded for guests who are obviously foreigners.
      Used for police reports in case the guest does not pay."),
  Class("Guest") owns (
    Member("name"),
    Member("passport")
  )
)
```

Listing 2.5: A data model. Example adapted from Lauesen [4].

2.4 Spec+Why+Example

It is often useful to include both rationale (why) and example design information (how) in a specification of what the system should do. In many cases, designs are explicitly marked as just examples and not as mandatory solutions in order to keep the freedom of inventing better solutions at a later point in time.

```
Model(
  Feature("navigate") has (
    Spec(
      "It shall be possible to perform the commands start, stop, ...
      with both hands at the patient."),
    Why(
      "Measuring neural response is a bit painful to the  patient.
      Electrodes must be kept in place ... So both hands should be
      at the patient during a measurement."),
    Example(
      "Might be done with mini keyboard (wrist keys), foot pedal,
      voice recognition, etc.")
  )
)
```

Listing 2.6: Including help for readers to better understand the spec. Example adapted from Lauesen [4]

2.5 Context diagram

```
Model(
  Product("HotelSystem") owns (
    Interface("ReceptionUI"), Interface("GuestUI"),
    Interface("TelephonyAPI"), Interface("AccountingAPI")
  ),
  Product("HotelSystem") has Image("context-diagram.png"),
  Interface("ReceptionUI") has (
    Input("booking, check-out"), Output("service note"),
    Image("recetionUI-screen.png")
  ),
  Interface("GuestUI") has (
    Output("confirmation, invoice"),
    Image("guestUI-screen.png")),
  Actor("Receptionist") requires Interface("ReceptionUI"),
  Actor("Guest") requires Interface("GuestUI"),
  Actor("Receptionist") requires Interface("ReceptionUI"),
  Actor("Telephony System") requires Interface("TelephonyAPI"),
  Actor("Accounting System") requires Interface("AccountingAPI")
)
```

Listing 2.7: Contex diagram example.

2.6 Quality requirements

```
Model(
  Quality("ttpm") has (
    Gist("Time to play music"),
    Spec("Measured in milliseconds using Test Case X"),
    Utility(10000), Differentiation(1000), Saturation(50),
    Submodel(
      Target("easy") has QualityLevel(140),
      Barrier("first") has (Min(1400), Max(1600)),
      Barrier("second") has QualityLevel(500),
      Product("competitorX") has QualityLevel(2000),
      Product("competitorY") has QualityLevel(2000)
    )
  )
)
```

Listing 2.8: A quality requirement.

2.7 Deep models

Any entity in a Model can contain a Submodel attribute that in turn can contain a Model, hence enabling a hierarchy of models in a recursive tree structure. A hierarchical modeling approach can be used for scalability reasons when there is a need to modularize large models, but also for expressing models where e.g. different stakeholder have different priorities for the same set of features, as is shown in Listing **??** in Section **??**. References to values of an attribute of a certain entity is created using the bang operator, e.g. the expression `(Feature("x")!Prio)` constructs a reference to the `Prio` attribute value of `Feature("x")`. If models are contained inside models, references including submodel paths of arbitrary lengths can be constructed by successive application of the bang operator.

For example, the expression `(Stakeholder("s")!Feature("x")!Prio)` refers to the submodel of `Stakeholder("s")` and the `Prio` attribute value of `Feature("x")` in that submodel.

```
var myReqs = Model(
  Feature("nice") has Spec("this is a nice feature"),
  Feature("cool") has Spec("this is a cool feature"),
  Stakeholder("Anna") has Submodel(
    Feature("nice") has Prio(1),
    Feature("cool") has Prio(2)
  ),
  Stakeholder("Martin") has Submodel(
    Feature("nice") has Prio(2),
    Feature("cool") has Prio(1)
  )
)
```

You can reference values of attribute in deeply nested submodel structures using the ! operator.

```
val m = Model(
  Feature("f") has Prio(1),
  Stakeholder("a") has Submodel(
    Feature("g") has Benefit(2),
    Resource("x") has Submodel(
      Feature("h") has Cost(3)
    )
  )
)

m(Feature("f")!Prio) == 1
m(Stakeholder("a")!Feature("g")!Benefit) == 2
m(Stakeholder("a")!Resource("x")!Feature("h")!Cost) == 3
```

Chapter 3

Compute models

3.1 Model operators

Users of reqT can carve out parts of models with special operators, including the restrict `/` and exclude `\` operators [6]. Given a model `m`, the expression `m / Feature` evaluates to a new model restricted to keys only containing entities of type Feature.

The exclude operator used in the complementary expression `m \ Feature` yields a new Model with all keys of m that do *not* have a Feature entity.

It is also possible to use the restrict and exclude operators over attributes and relations. There are several other operators for, e.g., aggregation of models and for extracting parts of a model using depth first search by following relation edges. Table **??** show some of the methods available on models. For a complete account of model methods, see `http://reqt.org/api/index.html#reqt.Model`.

Table 3.1: Some examples of applying operators on models
The m variable is of type Model.
m += x is the same as m = m + x
owns is a special relation: an entity can only have one direct owner
all operations are immutable; new model is created

Example	Effect
restrict on sources	
`m / Feature("a")`	Restrict m to sources with entity `Feature("a")`
`m / Feature`	Restrict m to sources with enities of type `Feature`
`m / Gist`	Restrict m to sources has attribute Gist
restrict on destinations	
`m /-> Feature("a")`	Restrict m to sources with Feature("a") as a destination
restrict extended	
`m /+ Feature("a")`	Restrict m to keys with entity Feature("a")

<table>
<tr><th>To do this...</th><th>...code this...</th></tr>
<tr><td>Create empty model</td><td><code>var m = Model()</code></td></tr>
<tr><td>Add entity</td><td><code>m += Feature("hi") has Spec("print greeting")</code></td></tr>
<tr><td>Add entity</td><td><code>m += Feature("f1") has (Gist("g1"), Spec("s1"))</code></td></tr>
<tr><td>Overwriting attribute</td><td><code>m += Feature("f1") has Gist("g2")</code></td></tr>
<tr><td>Add an owns-relation</td><td><code>m += Product("p1") owns (Feature("f1"), Feature("hi"))</code></td></tr>
<tr><td>Remove an entity</td><td><code>var m2 = m - Feature("f1")</code></td></tr>
<tr><td>Restrict operator</td><td><code>m / Feature("f1")</code>
<code>m / Feature</code>
<code>m / Spec</code>
<code>m / Context</code>
<code>m / Feature / Gist</code></td></tr>
<tr><td>Restrict to destinations</td><td><code>m /-> Feature("f1")</code></td></tr>
<tr><td>Extended restrict adds destinations, thus both sources and destinations of feature f1</td><td><code>m /+ Feature("f1")</code></td></tr>
<tr><td>Depth first search</td><td><code>m /--> Product("p1")</code></td></tr>
<tr><td>Partition
Partition extended
Partition on destinations
Partition depth first search</td><td><code>var (mx, my) = m | Feature</code>
<code>var (mx, my) = m |+ Feature</code>
<code>var (mx, my) = m |-> Feature</code>
<code>var (mx, my) = m |--> Feature</code></td></tr>
<tr><td>Aggregate</td><td><code>mx ++ my</code></td></tr>
<tr><td>Difference</td><td><code>mx -- my</code></td></tr>
<tr><td>Intersect</td><td><code>mx & my</code></td></tr>
<tr><td>Exclude</td><td><code>m \ Feature("f1")</code></td></tr>
<tr><td>Other Complement operators</td><td><code>m \-> Feature("f1")</code>
<code>m \+ Feature("f1")</code>
<code>m \--> Feature("f1")</code></td></tr>
<tr><td>Add same attribute to all entities</td><td><code>m + Gist("same same")</code></td></tr>
<tr><td>Remove all Gist attributes</td><td><code>m - Gist</code></td></tr>
</table>

3.2 Modeling decision status

```
reqT> val s = Status.init
s: reqT.Status = Status(ELICITED)

reqT> s.up
res1: reqT.Status = Status(SPECIFIED)

reqT> s.down
res2: reqT.Status = Status(DROPPED)

reqT> var m = Model(Feature("x") has Status.init, Feature("y") has Status.init)
m: reqT.Model =
Model(
  Feature("x") has Status(ELICITED),
  Feature("y") has Status(ELICITED)
)

reqT> m.up
res3: reqT.Model =
Model(
  Feature("x") has Status(SPECIFIED),
  Feature("y") has Status(SPECIFIED)
)

reqT> (m / Feature("x")).up ++ (m \ Feature("x"))
res4: reqT.Model = Model(
  Feature("x") has Status(SPECIFIED),
  Feature("y") has Status(ELICITED)
)

reqT> m.up(Feature("x"))   //same as above but shorter
```

Listing 3.1: Promoting a status level up the salmon ladder.

Chapter 4

Interoperate with other tools

4.1 Save and load reqT models in Scala

```
reqT> val m = Model(Req("x") relatesTo Req("y"))
m: reqt.Model = Model(Req("x") relatesTo Req("y"))

reqT> m.save("my-model.scala")
<console>:19: error: value save is not a member of reqt.Model
            m.save("my-model.scala")


reqT> "any string can be saved".save("str.txt")
Saved to file: C:\Users\bjornr\workspace\reqT/str.txt               ^

reqT> m.toString.save("my-model.scala")   //same as m.toScala.save("my-model.scala")
Saved to file: C:/Users/bjornr/workspace/reqT/gists/my-model.scala


reqT> val m2 = Model.load("my-model.scala")  //load model from file
m2: reqt.Model = Model(Req("x") relatesTo Req("y"))

reqT> val str = load("my-model.scala")   //load file as string
str: String = Model(Req("x") relatesTo Req("y"))

reqT> val m3 = str.toModel  //interpret str using scala interpreter, Model expected
m3: reqt.Model = Model(Req("x") relatesTo Req("y"))

reqT> ls   //list files in workDir
my-model.scala

reqT> pwd  //print workDir path
workDir == C:/Users/bjornr/workspace/reqT/gists

reqT> cd("..")  //change dir
workDir == C:/Users/bjornr/workspace/reqT
```

Listing 4.1: Save and load models.

4.2 Model visualization with GraphViz

4.3 Generate txt documents

4.4 Generate html documents

4.5 Export and import spreadsheet tables

Chapter 5

A Catalogue of Concepts

5.1 Entities

5.1.1 Context Entities

- **Product** A software (and hardware?) system for which requirements are modeled.
- **Release** A version of a product offered to stakeholders at a certain point in time.
- **Stakeholder** A role or person that have a stake in a product's requirements, e.g. a specific customer.
- **Actor** A human user, machine or system that interacts with a product through an interface.
- **Resource** An asset used in the development of a product, e.g. development team effort, test team effort, monetary investments.
- **Subdomain** A decomposition of the modeled domain.
- **Component** A subsystem that may be part of a product, e.g. a platform, library, API or java package.

5.1.2 Requirements Entities

Intentional Requirements

- **Goal** Objective behind a product. Goals can be refined and related by helps and hurts relations.
- **Wish** Something desired, but not yet formulated as a candidate requirement.

Generic Requirements

- **Req** An abstract entity without specific nature. Often used to make abstract reasoning about requirements relations in principal. Can be used when type of requirement is unknown.
- **Idea** An initial, perhaps creative, suggestion or other result of e.g. brainstorming, not yet formulated as a candidate requirement.

- **Feature** A decision unit describing a property of a product that can be selected in a release if it has a good benefit/cost ratio.
- **Label** A generic entity that labels (a set of) requirements into a certain category. Often used together with a comment attribute.

ToDoReq

- **Issue** A problem, bug, change request, improvement proposal or similar. Often found in customer feedback and support systems.
- **Ticket** A work item that can be assigned to a resource. Something that should or could be done, e.g. a requirements engineering task such as further interviews with stakeholder X.

Quality Requirements

- **Quality** A quality requirement or aspect. Often involves a scale and a measure of quality level on that scale.
- **Barrier** Getting beyond this quality level gives a steep increase in cost.
- **Target** A quality level that should or could be reached, e.g. in a specific release for a specific quality aspect for a specific feature.

Functional Requirements

- **Function** A requirement on how input data is mapped to output data. Functionality of the system in terms of input, processing and output.
- **Interface** A system border where interaction with surrounding actors are taking place, for example be a user interface or a communication link with a protocol, transfer data to the surrounding world.
- **Design** A specific realization or internal aspects of system and architecture. Captures 'how'.

Scenario Requirements

- **UserStory** Short description of what user roles want and why.
- **UseCase** A product-level description of how user roles (actors) reaches their goals.
- **TestCase** A specification of acceptance criteria in terms of input and expected output.
- **Task** A domain-level description of how user roles (actors) reaches their goals.
- **Scenario** An example-based usage description, for example a vivid narrative of current work or a futuristic story with envisioned details.

Data Requirements

- **Data** A data requirements describes what data to be stored in the system.

- **Class** A class defines a type of data that can be instantiated in the domain or in the product.
- **Member** A member of a class, an attribute or method.
- **Relationship** An specific association between data entities or classes. Often used together with relations with cardinality, e.g.:

```
Model(
  Data("libraryDatabase") has Image("libraryDatabase-ER.svg"),
  Class("book") holds Member("title"),
  Class("author") holds Member("name"),
  Class("book") relatesToOneOrMany Class("author"),
  Relationship("writtenBy") holds (
    Class("book"), Class("author")
  ),
  Class("author") relatesToOneOrMany Class("book"),
  Relationship("authorOf") holds (
    Class("author"), Class("book")
  )
)
```

Listing 5.1: A relationships among classes in a library system.

Product Line Requirements

- **VariationPoint** To express product line variability. A variation point relates to a set of variants that can be bound in specific products.
- **Variant** A specific choice of a variation point, e.g.:

```
Model(
  Component("myPlatform") has Submodel(
    VariationPoint("color") relatesToOneOrMany (
      Variant("blue"), Variant("red"), Variant("green")
    ),
    VariationPoint("shape") relatesToOne (
      Variant("round"), Variant("square")
    ),
    Variant("round") excludes Variant("red")
  ),
  Product("coolApp") binds Component("myPlatform"),
  Product("coolApp") has Submodel(
    VariationPoint("color") binds (
      Variant("red"), Variant("yellow")
    ),
    VariationPoint("shape") binds Variant("round")
  )
)
```

Listing 5.2: A variability model.

5.2 Relations

- **owns** A relation expressing exclusive ownership between entities. If an entity x owns another entity y, then y cannot be owned by another entity.

- **holds** A relation expressing non-exclusive "ownership" between entities. Both entity x and entity z can hold entity y.
- **requires** A relation expressing that an entity requires (a set of) entities. If entity x requires entity y, then if x is included in the product (release) then so should also y. The requires relation is not mutual. A two-way requires relations is modeled using two requires relations, e.g.:

```
Model(
  Req("x") requires Req("y"),
  Req("y") requires Req("x")
)
```

Listing 5.3: A mutual requires relationship.

- **excludes** A relation expressing that an entity excludes (a set of) entities. If entity x excludes entity y, then if x is selected to be included in the product (release) then y cannot be selected and vice versa.
- **releases** This relation can be used to express that a release entity has a set of requirements entities in its release plan.
- **helps** Goals support other goals and the helps relation can, e.g., express that the goals g_1 and g_2 both support the achievement of goal g_0:

```
Model(
  Goal("g1") helps Goal("g0"),
  Goal("g2") helps Goal("g0"),
  Goal("g3") hurts Goal("g0")
)
```

Listing 5.4: A goal model with helps and hurts relations.

- **hurts** Similarly to helps, hurts can express that a goal is hindering the achievement of another goal. In Listing 5.4 goal g_3 is negatively contributing to the achievement of g_0.
- **precedes** This relation expresses temporal ordering among requirements. If requirement r precedes requirement s then r should be implemented before s.
- **inherits**
- **binds**
- **implements**
- **verifies**
- **deprecates**

Relations with Cardinality

- **relatesTo**
- **relatesToOne**
- **relatesToOneOrMany**
- **relatesToZeroOrMany**
- **relatesToZeroOrOne**

5.3 Attributes

5.3.1 String value attributes

- **Gist**
- **Spec**
- **Why**
- **Example**
- **Expectation**
- **Input**
- **Output**
- **Trigger**
- **Precond**
- **Frequency**
- **Critical**
- **Problem**
- **Comment**
- **Deprecated**

5.3.2 Integer value attributes

- **Prio**
- **Order**
- **Cost**
- **Benefit**
- **Capacity**
- **Urgency**
- **Utility**
- **Differentiation**
- **Saturation**
- **Value**
- **QualityLevel**
- **Min**
- **Max**
- **Label**

5.3.3 Special attributes

- **Status**
- **Submodel**
- **Code**
- **Constraints**
- **Image**
- **External**

5.4 The reqT metamodel

A bag of entities for you to combine with a bag of attributes, and with other entities using a bag of relations. See what you can play with in Figure 5.1. You can mix and match as you please.

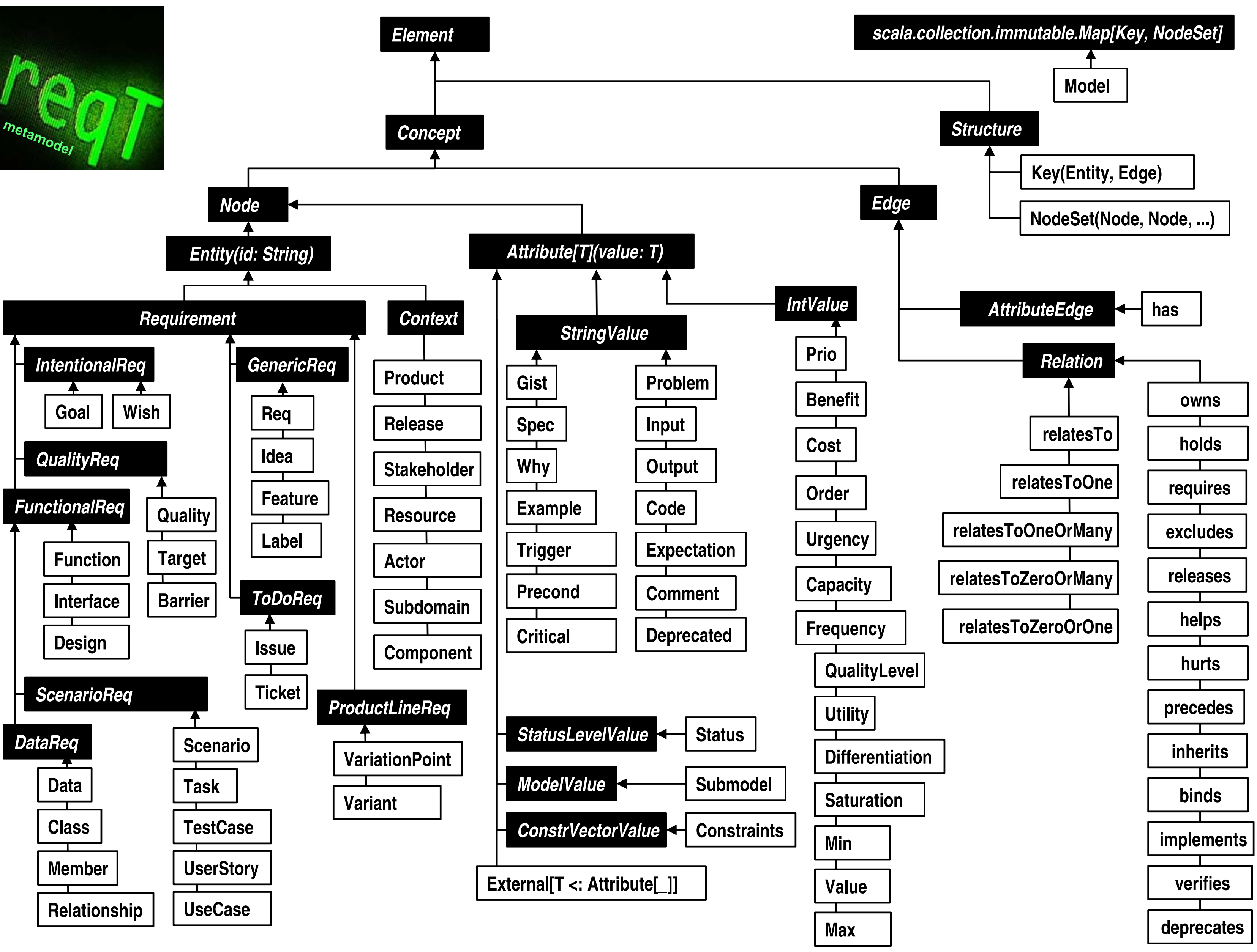


Figure 5.1: The reqT metamodel.

Appendix A

Scala by example

A.1 Object-oriented Scala

```
//a singleton object with one method (corresponds to static in java)

object hello { def to(s: String) = "Hello " + s }

hello.to("world")   //> res2: String = Hello world

//same as above, but operator notation
hello to "world"    //> res3: String = Hello world

//classes in Scala
class Hello2(val s: String) {  //automatic constructor generated, s is public
  val dot = "!"      //an assign-once variable that cannot be changed
  var extra = ""
  lazy val demo = { println("NOW"); "wake up" }   //assign when/if needed
  def greet(s2: String) = s +" " + s2 + dot + extra
}

val h2 = new Hello2("Hi")
h2.dot = ","  //compilation error: re-assignment to val
h2.extra = "?"
println(h2.greet("world"))
println(h2.demo)

case class Hello3(s: String, dot: String = ".", extra: String = "") {
  def hello(s2: String) = s +" " + s2 + dot + extra
}

val h3 = Hello3("Hi there", extra = "..") //new is not needed
println(h3 hello "friend")
```

A.2 Functional Scala

```
//** Functional Scala

//scala has a powerful collection library including immutable List, Vector, Set, Map
val xs1 = List(1,2,3)           //> xs1: List[Int] = List(1, 2, 3)
val prepend = 0 :: xs1          //> prepend: List[Int] = List(0, 1, 2, 3)
val xs2 = xs :+ 4               //> xs2: List[Int] = List(1, 2, 3, 4)
val xs3 = xs ++ List(1,2,3)     //> xs3: List[Int] = List(1, 2, 3, 1, 2, 3)

//functions are first class values that can be passed as parameters
def twice(i: Int) = i*2         //> twice: (i: Int)Int
val xs4 = xs3.map(twice)        //> xs4: List[Int] = List(2, 4, 6, 2, 4, 6)

//lambda notation for nameless concise functions
val xs5 = xs3.map(x => 2 * x)   //> xs5: List[Int] = List(2, 4, 6, 2, 4, 6)
val xs6 = xs3.map(2 * _ )       //> xs6: List[Int] = List(2, 4, 6, 2, 4, 6)

val trice = (x : Int) => x * 3  //> trice: Int => Int = <function1>
val xs7 = xs3 map trice         //> xs7: List[Int] = List(3, 6, 9, 3, 6, 9)

//function application is actually a method call to method apply
object quad {
  def apply(x: Int) = x * 4
}
val x1 = quad.apply(4)          //> x1: Int = 16
val x2 = quad(4)                //> x2: Int = 16

//functions are objects
object quint extends Function1[Int,Int] {
  def apply(x: Int) = x * 5
}
val xs8 = xs3 map quint         //> xs8: List[Int] = List(5, 10, 15, 5, 10, 15)
```

Bibliography

[1] reqT web page, `http://reqT.org/`, visited June 2013.

[2] Carrillo de Gea, J., Nicolas, J., Aleman, J., Toval, A., Ebert, C., Vizcaino, A.: Requirements engineering tools. Software, IEEE 28(4), 86 –91 (july-aug 2011)

[3] Kogics: Kojo, `http://www.kogics.net/kojo`, visited Nov 2012.

[4] Lauesen, S.: Software Requirements - Styles and Techniques. Addison-Wesley (2002)

[5] Odersky, M., et al.: An overview of the Scala programming language. Tech. rep. (2004), `http://lampwww.epfl.ch/~odersky/papers/ScalaOverview.html`

[6] Regnell, B.: Reqt.org – towards a semi-formal, open and scalable requirements modeling tool. In: Doerr, J., Opdahl, A. (eds.) Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, vol. 7830, pp. 112–118. Springer Berlin Heidelberg (2013)

[7] Scala Eclipse IDE: `http://scala-ide.org/`, visited Nov 2012.

[8] Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM Sigplan Notices 35(6), 26–36 (2000)