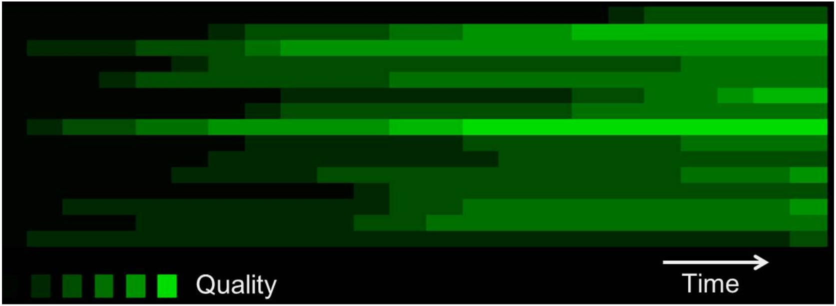
Flexible Requirements Modeling with reqT

`http://reqT.org`

bjornregnell.se, Lund University, Sweden

26 november 2013

Good enough Requirements Engineering?



3 software engineering trends: Decentralize, Distribute, Document less

- Agile teams + SW ecosystems -> No centrally controlled, detailed "master plan"
- Continuous integration & deployment
- Increased parallelization
- Distributed Version Control, e.g. Git
- A large part of the requirements engineering is done by developers

A challenge

How to best help code-focused,

agile software engineers to do

good enough **requirements engineering**

in a decentralized, distributed, and

documentation-hostile setting?

reqT: Requirements => ComputationalEntities

In reqT ...

- requirements are computational entities
- requirements are serialized as self-generating code
- the meta-model and its semantics are flexible:

```
> allow me to flexibly mix text with structure
> warn me, don't force me
```

Goals of reqT – a Requirements Engineering Tool

Design an interesting tool based on these goals:

Goal	Design choices	Rationale
Semi-formal	• Use graph structures • Mix human Natural Language (NL) with essential RE semantics	• Graphs are well-known by software engineers and powerful for expressing structure and flexible for search. • NL is well-known and powerful...
Open	• Free, permissive OSS license • Cross-platform: JVM	• Allow integration of existing code bases in JVM-based languages • Enable academic usage and contribution in teaching and research
Scalable	• Internal DSL in Scala `www.scala-lang.org`	• Open-ended language • Scala is scalable, powerful, concise, typesafe, scriptable, ...

Paper at REFSQ2013: `http://www.reqt.org/reqT-REFSQ2013-paper.pdf`

Why embedd a DSL in Scala?

- Object-functional abstraction (traits, implicits, generics)
- Type inference
- Scala collections library
- Interactive and compiled (Scala REPL)
- Free and platform independent (runs on JVMs)
- Clean and flexible syntax

```
scala> object hello { def to(s: String) = "Hello " + s }
defined module hello

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

scala> hello to "world"
res3: String = Hello world

scala>
```

What can you do with reqT?

- Create and manage requirements models using versatile collections
- Combine natural language expressiveness with type-safe modeling
- Interoperate with spread sheet applications
- Auto-generate requirements documents for web publishing
- Do powerful scripting of requirements models with the Scala-embedded DSL
- Model and solve combinatorial decision problems in RE such as Prioritisation and Release Planning using a sub-DSL for Constraint Satisfaction Programming wrapping the JaCoP solver

`http://reqT.org`

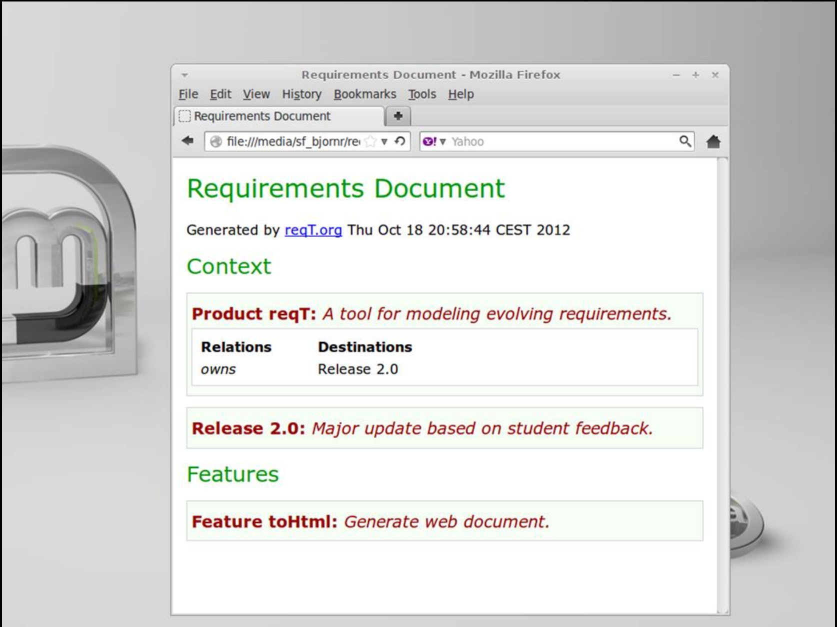
`http://www.jacop.eu`

reqT Gist

```
var m = Model(
  Product("reqT") has Gist("A tool for modeling evolving requirements."),
  Release("2.0") has Gist("Major update based on student feedback."),
  Product("reqT") owns Release("2.0")
)

m += Feature("toHtml") has Gist("Generate web document.")

println(m)
m.toHtml.save("reqT.html")
m.toTable.save("reqT.txt")
```



Requirements Document

Generated by reqT.org Thu Oct 18 20:58:44 CEST 2012

Context

Product reqT: *A tool for modeling evolving requirements.*

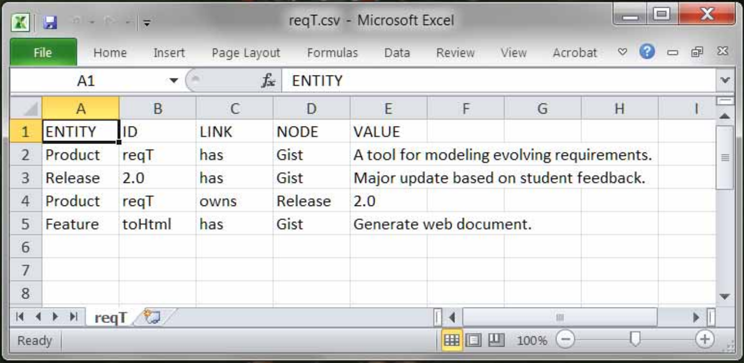
Relations	Destinations
owns	Release 2.0

Release 2.0: *Major update based on student feedback.*

Features

Feature toHtml: *Generate web document.*

Feature("toTable")



ENTITY	ID	LINK	NODE	VALUE
Product	reqT	has	Gist	A tool for modeling evolving requirements.
Release	2.0	has	Gist	Major update based on student feedback.
Product	reqT	owns	Release	2.0
Feature	toHtml	has	Gist	Generate web document.

```
m.toTable(";").save("reqT.csv")
loadTable("reqT.csv",";")
```

The reqT DSL (Domain Specific Language)

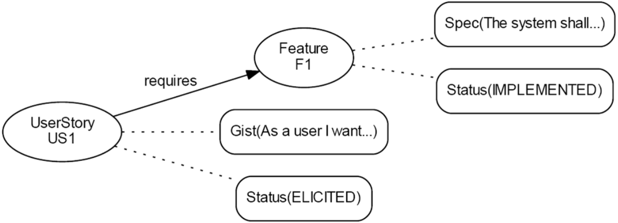
A reqT model includes a sequence of graph parts
<Entity> <Edge> <NodeSet>
separated by comma and boxed inside a Model( )

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

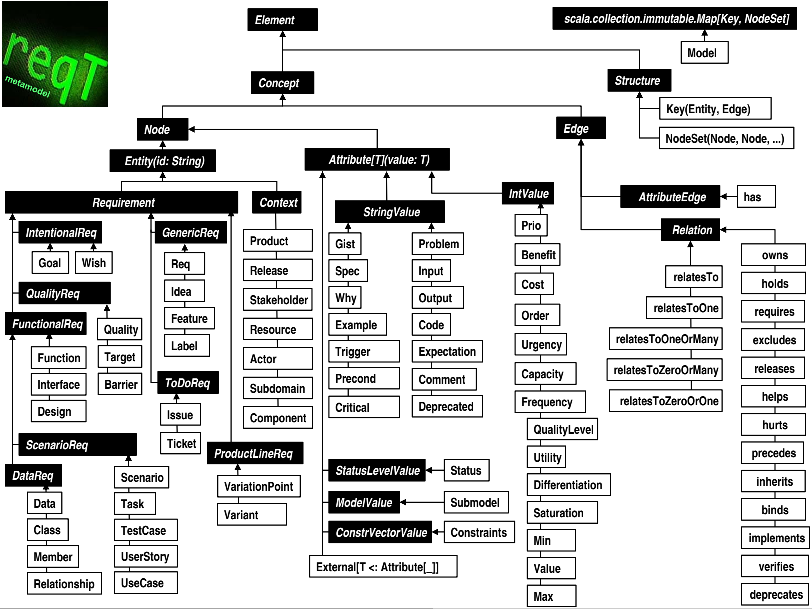
reqT models are graph structures with Entities & Attributes (nodes) and Relations (edges)

```
reqT> var myReqs = Model(
  Feature("F1") has (Spec("The system shall..."), Status(IMPLEMENTED)),
  UserStory("US1") has (Gist("As a user I want..."), Status(ELICITED)),
  UserStory("US1") requires Feature("F1")
)
reqT>  myReqs.toGraphViz.save("myGraph.dot")

$ dot -Tpdf myGraph.dot -o myGraph.pdf
```



The reqT metamodel

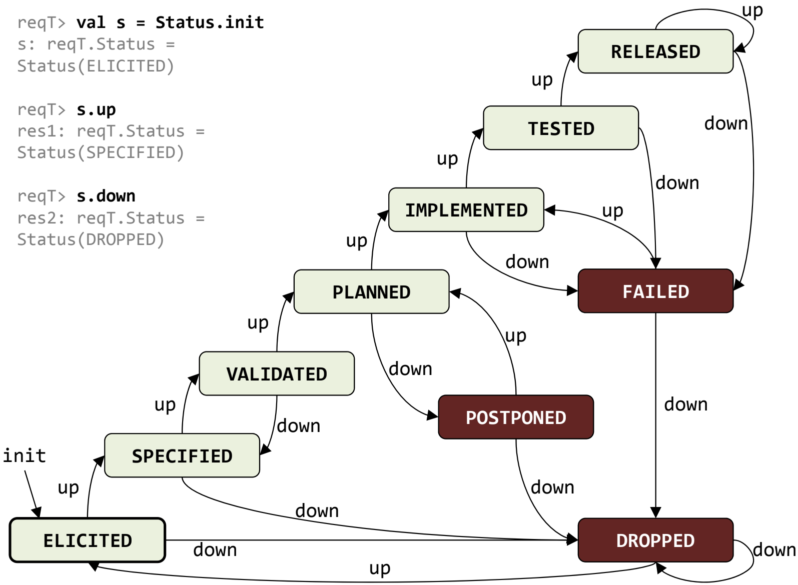


reqT Level values of the Status attribute

```
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 update of Status attribute

```
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
res8: reqT.Model =
Model(
  Feature("x") has Status(SPECIFIED),
  Feature("y") has Status(SPECIFIED)
)

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

reqT> m = m up Feature("x")          //equivalent shorter way to do previous
```

reqT models can be hierarchical with recursive submodels in a tree structure

```
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)
  )
)
```

reqT 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
```

reqT Entities can have a Constraints attribute containing a sequence of constraints.

```
var myReqs = Model(
  Feature("nice") has Spec("this is a nice feature"),
  Feature("cool") has Spec("this is a cool feature"),
  Stakeholder("Anna") has Constraints(
    (Feature("nice")!Prio) #< 10,
    (Feature("nice")!Prio) #>= 1,
    (Feature("cool")!Prio)::{2 to 7}
  ),
  Stakeholder("Martin") has Constraints(
    (Feature("nice")!Prio) #< 3,
    (Feature("nice")!Prio) #!= 1,
    (Feature("cool")!Prio)::{5 to 10}
  )
)

myReqs.impose(myReqs.constraints).solve(Satisfy)  //invoke JaCoP

myReqs.satisfy    //same as above
```

reqT context description example

You can use reqT with your favourite RE text book.

```
Model(
  Product("Hotel system") owns (
    Interface("ReceptionUI"), Interface("GuestUI"),
    Interface("TelephonyAPI"), Interface("AccountingAPI")
  ),
  Product("Hotel system") has Image("context-diagram.png"),
  Interface("ReceptionUI") has (
    Input("booking, check-out"), Output("service note"),
    Image("receptionUI-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")
)
```

[Example modified from Lauesen: Software Requirements – Styles and Techniques]

reqT task description example

```
Model(
  Task("reception work") owns (Task("check in"), Task("booking")),
  Task("check in") has (
    Why("Guest wants room."),
    Trigger("A guest arrives"),
    Frequency(3),
    Spec("Give guest a room. Mark it as occupied. Start account."+
         "Frequency scale: median #check-ins/room/week"),
    Critical("Group tour with 50 guests.")
  ),
  Task("check in") owns (
    Task("find room"), Task("record guest"), Task("deliver key")
  ),
  Task("record guest") has Spec(
    "variants: a) Guest has booked in advance, b) No suitable room"
  )
)
```

[Example modified from Lauesen: Software Requirements – Styles and Techniques]

reqT quality requirements example

```
Model(
  Quality("capacity database") has
    Spec("#guests < 10,000 growing 20% per year, #rooms < 1,000"),
  Quality("accuracy calendar") has
    Spec("Bookings shall be possible at least two years ahead."),
  Quality("performance forecast") has
    Spec("Product shall compute a room occupation forecast
          within ___ minutes. (Customer expects one minute.)"),
  Quality("usability task time") has
    Spec("Novice users shall perform tasks Q and R in 15 minutes.
          Experienced users tasks Q, R, S in 2 minutes."),
  Quality("usability task time") requires (Task("Q"), Task("R"), Task("S"))
  Quality("performance peak load") has
    Spec("Product shall be able to process 100 payment transactions
          per second in peak load.")
)
```

[Example modified from Lauesen: Software Requirements – Styles and Techniques]

reqT QUPER example

```
Model(
  Quality("ttpm") has (
    Gist("Time to play music"),
    Spec("Measured in milliseconds using Test Case X"),
    Image("QUPER-timeToMusic.jpg"),
    Utility(4000), Differentiation(1500), Saturation(200),
    Submodel(
      Target("ttpmBasic") has (
        QualityLevel(2000),
        Comment("Probably possible with existing architecture.")),
      Target("ttpmStrech") has (
        QualityLevel(1100),
        Comment("Probably needs new architecture.")),
      Barrier("first") has (Min(1900), Max(2100)),
      Barrier("second") has QualityLevel(1000),
      Product("competitorX") has QualityLevel(2000),
      Product("competitorY") has QualityLevel(3000)
    )
  )
)
```

Example modified from "Setting quality targets for coming releases with QUPER: an industrial case study", R. Berntsson Svensson, Y. Sprockel, B. Regnell, S. Brinkkemper, Requirements Engineering, November 2012, Volume 17, Issue 4, pp 283-298

reqT models can contain code and execute test cases

```
val m1 = Model(
  TestCase("add1") has (Code("1 + 42"), Expectation("42")),
  TestCase("add2") has (Code("{1 to 42}.sum"), Expectation("903")),
  TestCase("mul1") has (External[Code]("filename.scala"), Expectation("true"))
)

reqT> m1.loadExternals.run
res1: scala.collection.immutable.Map[reqt.Entity,String] =
Map(TestCase(add1) -> 43, TestCase(add2) -> 903, TestCase(mul1) -> true)

reqT> (m1 / "add2").tested
res2: reqt.Model =
Model(
  TestCase("add2") has (Expectation("903"), Output("903"), Code("{1 to 42}.sum"))
)

reqT> m1.loadExternals.isTestOk
*** FAILED: TestCase(add1)
  Output:    43
  Expectation: 42
res3: Boolean = false
```

Some example operations on reqT models

To do this...	...code this...
Create empty model	`var m = Model()`
Add entity with one attribute [1]	`m += Feature("hello") has Spec("print da stuff")`
Add entity with two attributes	`m += Feature("f1") has (Gist("g1"), Spec("s1"))`
Overwriting existing attribute	`m += Feature("f1") has Gist("g2")`
Add an owns-relation [2]	`m += Product("p1") owns (Feature("f1"), Feature("hello"))`
Remove an entity [3]	`var m2 = m - Feature("f1")`

Restrict operator	`m / Feature("f1")`
	`m / Feature`
	`m / Spec`
	`m / Context`
	`m / Feature / Gist`

Restrict to destinations	`m /-> Feature("f1")`
Extended restrict adds destinations	`m /+ Feature("f1")`
Depth first search	`m /--> Product("p1")`

reqT set operations, complement to restrictions, etc.

<table>
<tr><th>To do this...</th><th>...code this...</th></tr>
<tr><td>Partition</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>

Why Constraint Solving in Requirements Engineering?

Some potential benefits of Constraint Satisfaction Programming (CSP) in RE:

- Flexible specification of decision problems
  - Prioritization
  - Release Planning
- Interactive exploration of the solution space
- Out-of-the-box optimization support

Some challenges:

- How to integrate CSP with RE technology and make it user friendly in the domain?
- How to model CSP problems at the right abstraction level given great uncertainties?

reqT Release Planning: User Input Data Model

```
val m = Model(
  Release("Alfa") has Order(1),  Release("Beta") has Order(2),
  Stakeholder("Martin") has (Prio(10), Submodel(
      Feature("F1") has Benefit(20),
      Feature("F2") has Benefit(20),
      Feature("F3") has Benefit(20)
    ),
    Constraints((Feature("F2")!Order) #< (Feature("F3")!Order))  //precedence constraint
  ),
  Stakeholder("Anna") has (Prio(20), Submodel(
      Feature("F1") has Benefit(5),
      Feature("F2") has Benefit(15),
      Feature("F3") has Benefit(35)
    ),
    Constraints((Feature("F1")!Order) #== (Feature("F2")!Order))  //coupling constraint
  ),
  Resource("DevTeam") has Submodel(
    Release("Alfa") has Capacity(100),
    Release("Beta") has Capacity(100),
    Feature("F1") has Cost(10),
    Feature("F2") has Cost(70),
    Feature("F3") has Cost(20)
  ),
  Resource("TestTeam") has Submodel(
    Release("Alfa") has Capacity(100),
    Release("Beta") has Capacity(100),
    Feature("F1") has Cost(40),
    Feature("F2") has Cost(10),
    Feature("F3") has Cost(50)
  )
)
```

reqT Release Planning: Implementing a specific modeling approach as a library function

```
def releasePlanningConstraints(m: Model): Constraints = {
  val features = (m.flatten / Feature).sourceVector
  val releases = (m / Release).sourceVector
  val resources = (m / Resource).sourceVector
  val stakeholders = (m / Stakeholder).sourceVector
  m.constraints ++ Constraints( ??? )
}
```

??? is replaced by constraints that capture the specific modeling approach as described subsequently.

One possible way to model the release planning problem using constraints.

1. Let **benefit(s,f)** denote the benefit of feature f according to stakeholder s multiplied by the priority of stakeholder s.
2. Let **benefit(f)** of a feature f be the sum of benefit(s, f) over all stakeholders.
3. If feature f is allocated to release r then **benefit(r,f)** = benefit(f) else benefit(r,f) = 0.
4. Let **totBenefit(r)** of a release r be the sum over all features of benefit(r, f).
5. For all releases rel, for all features f, for all resources res: If f is allocated to rel then let **cost(rel, f, res)** be the cost of that feature needed by that resource, else the cost is zero.
6. For all resources res, for all releases rel: Let **totCost(rel, res)** be the sum of cost(rel, f, res) over all features.
7. For all resources res, for all releases rel: **totCost(res, rel)** must be lest than or equal to the available capacity.
8. Let the total cost of a release rel, denoted **totCost(rel)**, be the sum of totalCost(res, rel) over all resources.

For a specific release alfa, search for a solution that maximizes totBenefit(alfa).

Release planning constraints encoded in reqT/CSP

```
Constraints(
  forAll(features) { f => (f!Order)::{1 to releases.size} } ++
  forAll(stakeholders, features) { (s, f) =>
    (s!f!Benefit) * (s!Prio) #== Var(s"benefit($s,$f)") } ++
  forAll(features) { f =>
    sumForAll(stakeholders)(s => Var(s"benefit($s,$f)")) #== Var(s"benefit($f)") } ++
  forAll(releases, features) { (r, f) =>
    IfThenElse((f!Order) #== (r!Order),
      Var(s"benefit($r,$f)") #== Var(s"benefit($f)"),
      Var(s"benefit($r,$f)") #== 0) } ++
  forAll(releases) { r =>
    sumForAll(features)(f => Var(s"benefit($r,$f)")) #== Var(s"totBenefit($r)") } ++
  forAll(releases,features, resources) { (rel, f, res) =>
    IfThenElse((f!Order) #== (rel!Order),
      Var(s"cost($rel,$f,$res)") #== (res!f!Cost),
      Var(s"cost($rel,$f,$res)") #== 0) } ++
  forAll(resources, releases) { (res, rel) =>
    sumForAll(features)(f => Var(s"cost($rel,$f,$res)")) #== Var(s"totCost($rel,$res)") } ++
  forAll(resources,releases) { (res, rel) =>
    Var(s"totCost($rel,$res)") #<= (res!rel!Capacity) } ++
  forAll(releases) { rel =>
    sumForAll(resources)(res => Var(s"totCost($rel,$res)")) #== Var(s"totCost($rel)") }
)
```

reqT Release Planning Optimization

```
val utility = Var("totBenefit(Release(Alfa))")
val (m2, result) =
  m.impose(releasePlanningConstraints(m)).solve(Maximize(utility))
val featureAllocation = m2 / Feature
val costOfAlfa = result.lastSolution(Var("totCost(Release(Alfa))"))
```

```
featureAllocation: reqt.Model =
  Model(
    Feature("F3") has Order(2),
    Feature("F1") has Order(1),
    Feature("F2") has Order(1)
  )
costOfAlfa: Int = 130
```

Conclusion

- Contributions, bug reports, pull requests are welcome.
  `https://github.com/reqT/reqT/`

Any feedback, questions, etc. welcome!!