ETS170 Kravhantering

Tutorial on requirements modelling with reqT with brief introduction to Scala

reqT home page: http://reqt.org

reqT cheat sheet: http://reqt.org/reqT-cheat-sheet.pdf

(Last update 2024 January 24)

Björn Regnell
http://cs.lth.se/krav/reqt/

Some question for you

- How will you partition your req space?
- How will you synchronize your work?
- What entity id policy will you have?
- How will you manage versions?
- How will you build your document from requirements fragments?

Which tools are you going to use?

- Office apps e.g. LibreOffice, MS Office (spreadsheet, wordprocessor, database)
- Latex
- Web publishing
- Configuration management (git, gitHub, gitLab, ...)
- Prototyping tools, gui-builders
- Issue trackers / ticket managers / trello etc.
- **reqT**

- **Who will be tool responsible?**

Which **dogmas** do we preach in requirements engineering?

The idea behind reqT

- Methodology agnostic: 'bag of concepts'
- Scalable collection data structure, from 1 to 10E4
- Scriptable: the power of Scala and the JDK
- CLI + GUI for power users
- Integrates with git and similar code/text tools
- Constraint solving for integer problems
- Open source, permissible license

Pros and cons of reqT

+ Tailored to the course terminology
+ **Entities**, **attributes** and **<u>relations</u>**
+ Modularization and aggregation
+ Hierarchical decomposition
+ Export – import
  (txt, html, dot, csv, pdf, svg)
+ Plain text combines well with
  * configuration mgmt
  * latex
+ Requirements => Code
  * syntactic and semantic checks
  * scriptable models

- still a prototype
- limited to power users
- limited documentation (but code is king :))
- still on old Scala 2.12

It helps if you are interested in coding and in learning a little bit of Scala

RE on planet Earth in 5-10 years … ?

Some hypotheses

More continuous build, integration & deployment

Faster release cycles & Faster innovation

More SW eco systems, distributed developer communities, open source, AI-based coding

=>

More **decentralization**

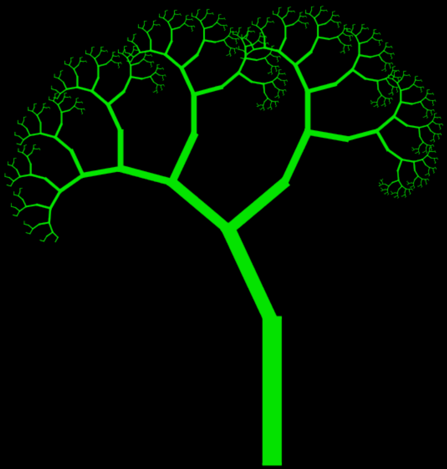
and fewer centrally controlled 'Master Plans'

More `coders`

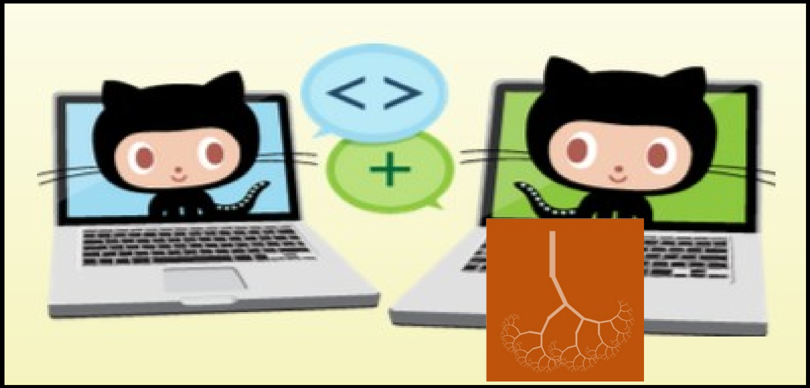
will do the bulk of requirements engineering

will use AI prompting in natural language

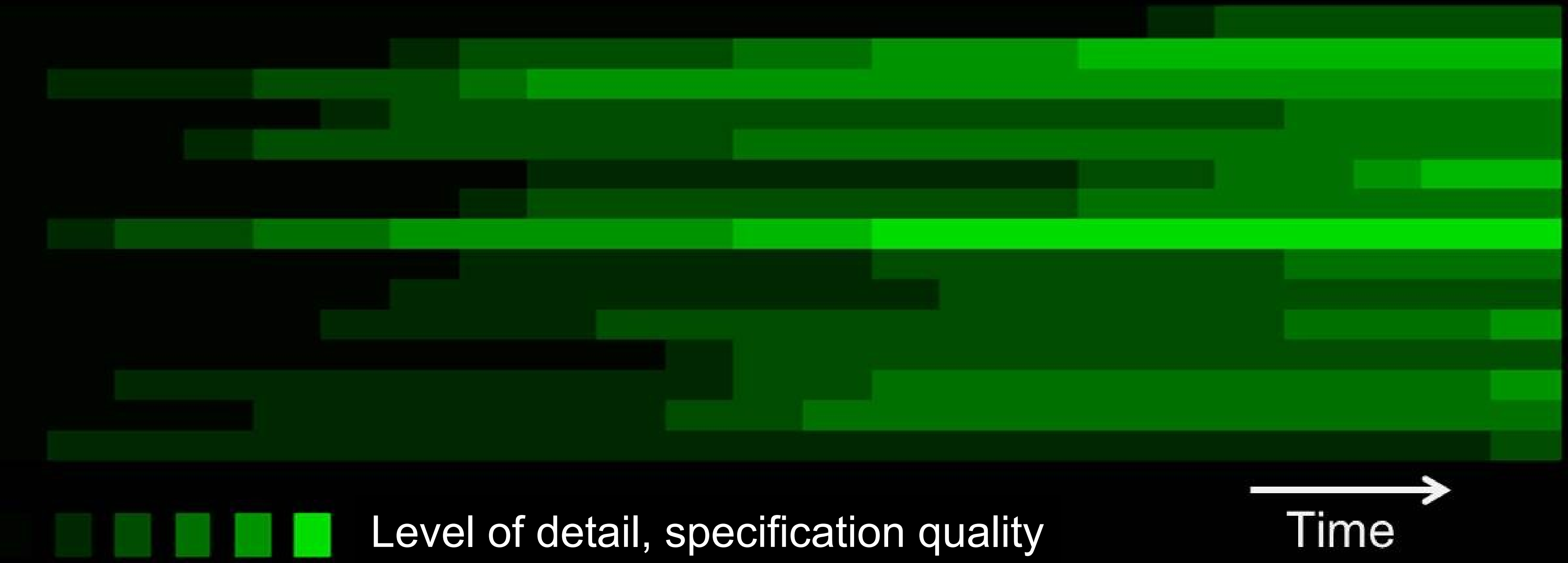
Scenario("Coders work in ecosystems with **req+code+test** in distributed git repos. Each stakeholder has its own, local understanding of ideas, roadmaps and acceptance criteria. Code is forked, pushed, pulled and merged **continuously** in the ecosystem. The 'ice berg' of **mixed, semi-formal models** is neither complete nor fully consistent. We manage local trees of req+code+test and mine sets of mixed, semi-formal models with big data technology on both dev repos and UX data. The **community culture** and repo governance determine success rather than process control.")



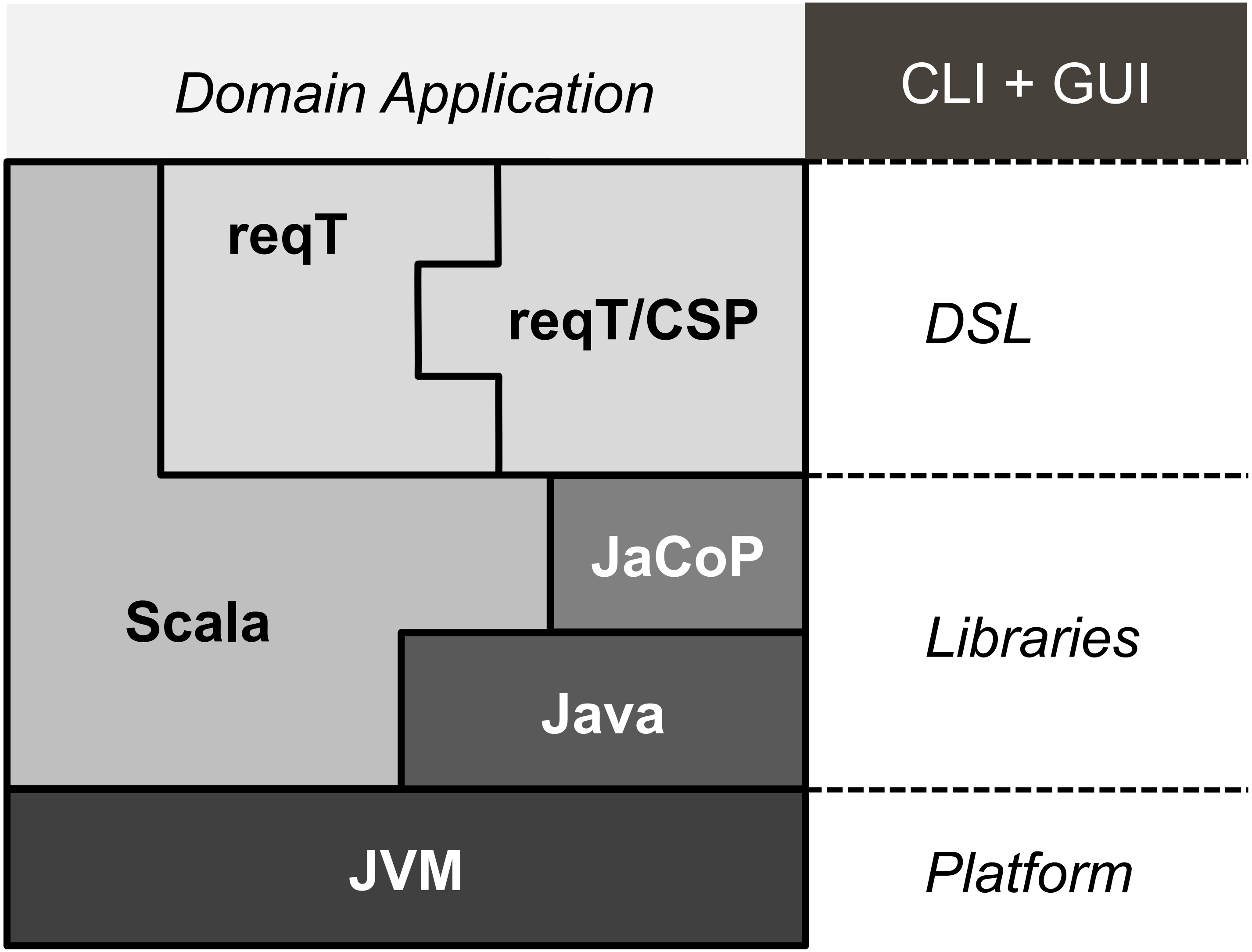
myModel ++ yourModel



Evolving mix of levels of detail & quality in continuous requirements engineering



reqT architecture



Open Source Software (OSS) in **reqT**

OSS	**Licence**
reqT	BSD-2-caluse
Scala libs & compiler	similar to BSD-2-caluse
JaCoP	GNU GPL v2 & v3
jLine	similar to BSD-2-caluse
RSyntaxTextArea	similar to BSD-2-caluse
jFlex	BSD-2-caluse
GraphViz	Eclipse Public License

Requirements as graph structures

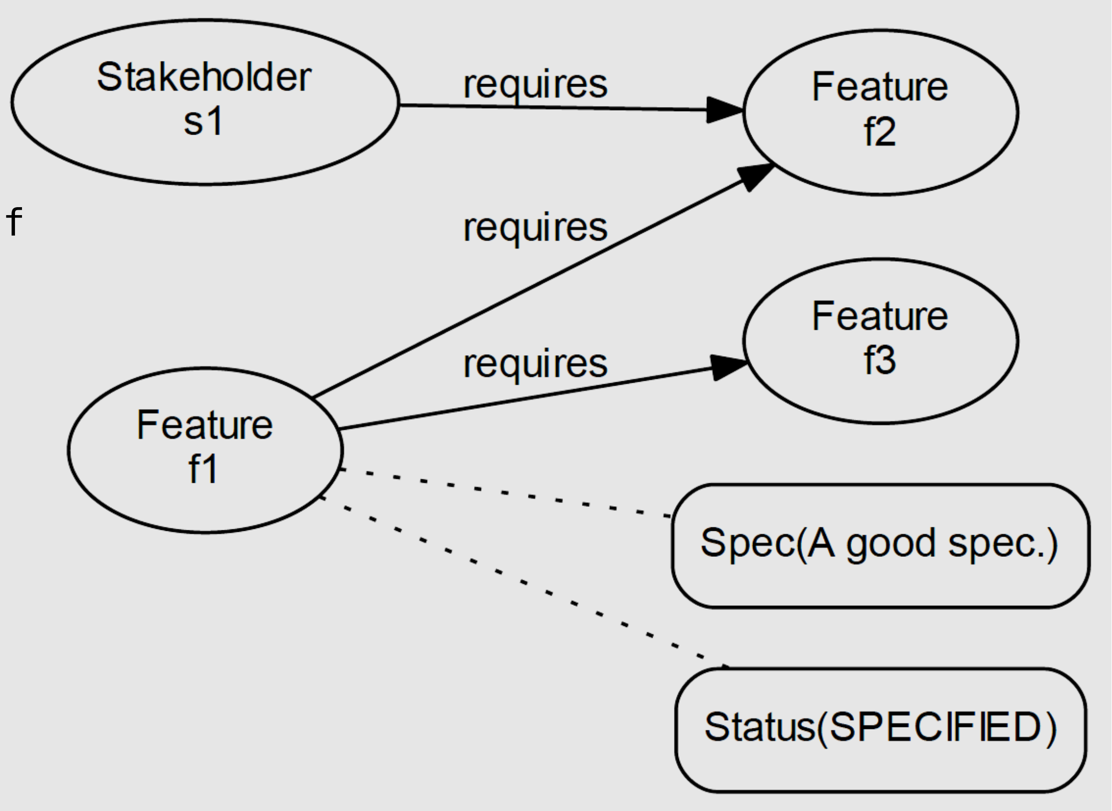
```
val m = Model(
  Feature("f1") has (Spec("A good spec."), Status(SPECIFIED)),
  Feature("f1") requires (Feature("f2"), Feature("f3")),
  Stakeholder("s1") requires Feature("f2")
)
```

```
m.toGraph.save("graph.dot")
```

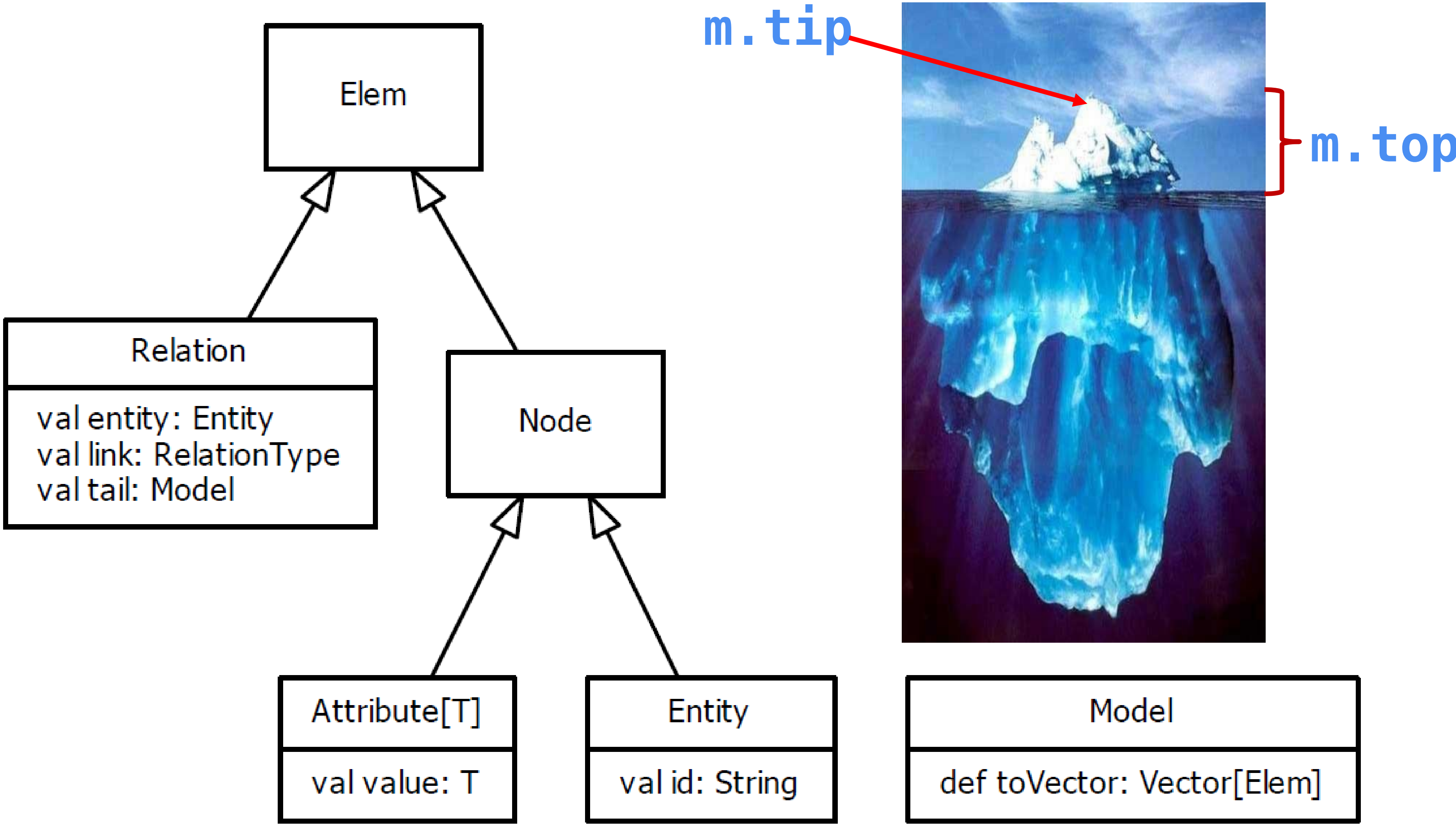
```
$ dot -Tpdf graph.dot -o graph.pdf
```

Download GraphVis:
http://graphviz.org

https://superuser.com/questions/949560/how-do-i-set-system-environment-variables-in-windows-10



The embedded DSL provides a recursive, tree-like data structure

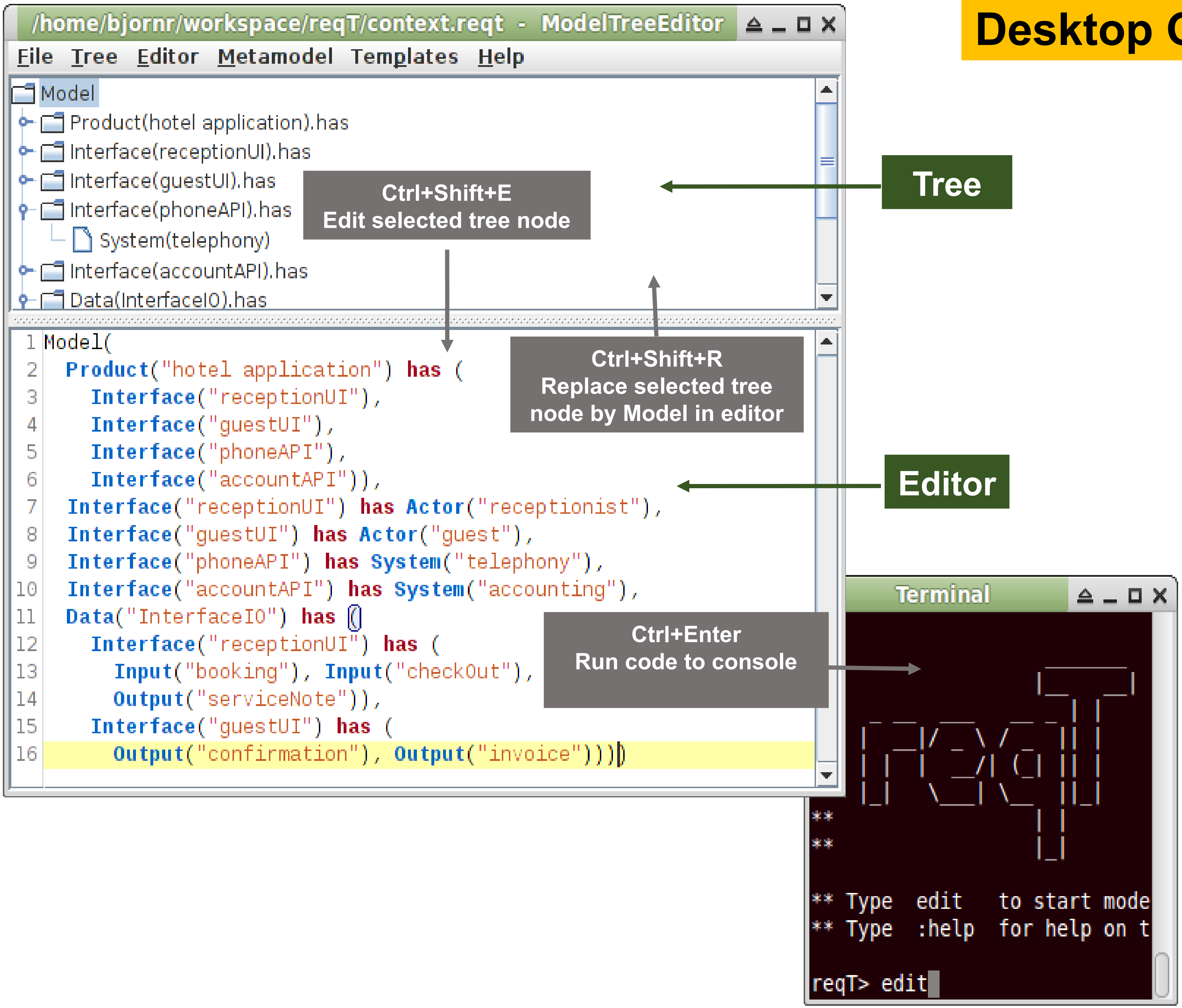


Requirements Entities

Examples from the reqT metamodel

Product, Interface,
Stakeholder, Idea, Goal,
Feature, Data, Function,
State, Event, Quality,
Design, Scenario, Story,
UseCase, Risk, Release,
Issue, Test, Variant, Req

Desktop GUI



/home/bjornr/workspace/reqT/context.reqt - ModelTreeEditor
File Tree Editor Metamodel Templates Help
Terminal

Model
- Product(hotel application).has
- Interface(receptionUI).has
- Interface(guestUI).has
- Interface(phoneAPI).has
  - System(telephony)
- Interface(accountAPI).has
- Data(InterfaceIO).has

```
Model(
  Product("hotel application") has (
    Interface("receptionUI"),
    Interface("guestUI"),
    Interface("phoneAPI"),
    Interface("accountAPI")),
  Interface("receptionUI") has Actor("receptionist"),
  Interface("guestUI") has Actor("guest"),
  Interface("phoneAPI") has System("telephony"),
  Interface("accountAPI") has System("accounting"),
  Data("InterfaceIO") has (
    Interface("receptionUI") has (
      Input("booking"), Input("checkOut"),
      Output("serviceNote")),
    Interface("guestUI") has (
      Output("confirmation"), Output("invoice")))
```

```
**
**

** Type  edit   to start mode
** Type  :help  for help on t

reqT> edit
```

Tree

Editor

Ctrl+Shift+E
Edit selected tree node

Ctrl+Shift+R
Replace selected tree node by Model in editor

Ctrl+Enter
Run code to console

Some essential requirements entitites and attributes

Req generic, abstract, undecided

Feature decision item with status

Stakeholder

Goal

UserStory, TestCase, Issue

Quality

Function

Data

...

Gist short one-liner

Spec txt descr

Why

Example

Prio

Cost

Benefit

Status

...

Some essential requirements **relations**

- Requirements entities have relations that turn the reqts into a **graph**

```
Model(
  Req("a") requires Req("b")
)
```

- **has**
- **requires**
- **excludes**
- **helps**
- **hurts**
- …

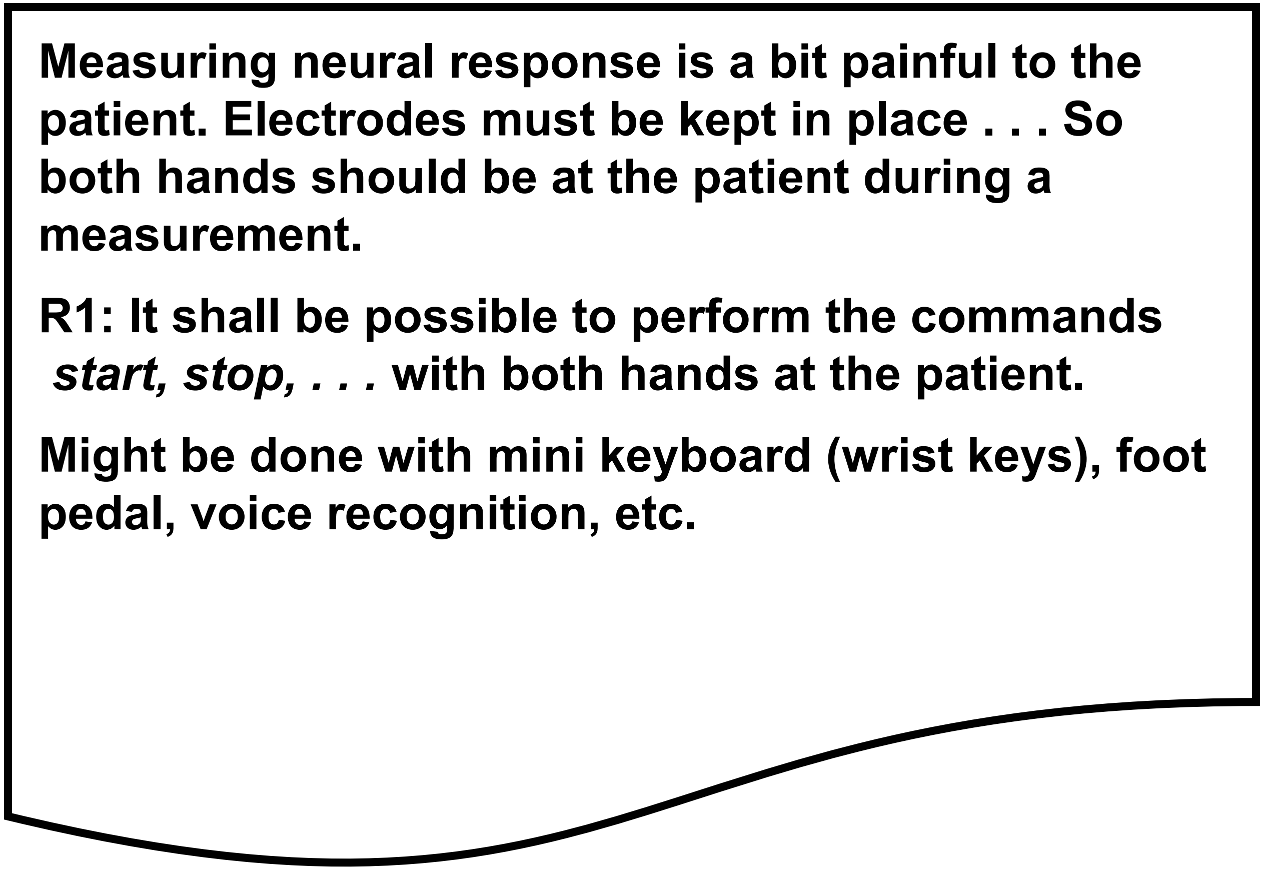
Split and merge

```
val myModel = Model(Req("x") has Spec("a"))

val yourModel = Model(Req("y") has Spec("b"))

val merged = myModel ++ yourModel

merged.toScala.save("newModel.scala")

Model(
  Req("x") has Spec("a"),
  Req("y") has Spec("b")
)
```

Short about Scala

- Scalable, concise, type safe
- Object-oriented meets functional
- Runs on the java virtual machine
- Can use any java byte code directly
- Statically typed: find bugs at compile time
- Type inference avoids boilerplate and keeps type safety
- Compile with `scalac` or run as scripts with `scala`
- The Scala 2.12 Read-Evaluate-Print-Loop (REPL) is wrapped **inside reqT** so you can make general programs in reqT while modeling requirements
- https://www.scala-lang.org/

Fig 1.6C Recommendation: why + how



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.

R1: It shall be possible to perform the commands *start, stop, . . .* with both hands at the patient.

Might be done with mini keyboard (wrist keys), foot pedal, voice recognition, etc.

Domain
- why

Req.

Example
- how

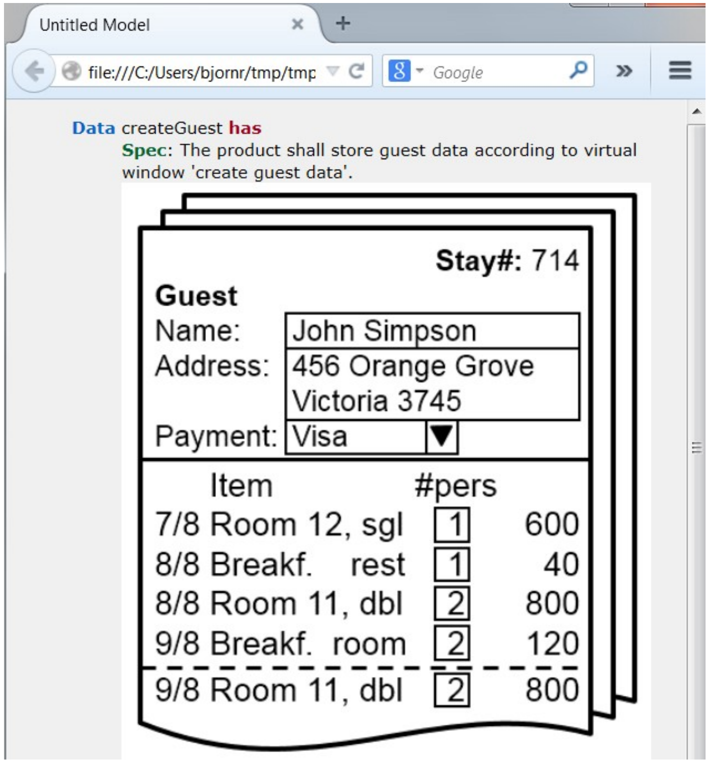
From: Soren Lauesen: Software Requirements

Why+How+Example

```
Model(
  Feature("navigate") has (
    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."),
    Spec(
    "It shall be possible to perform the commands start,
      stop, ... with both hands at the patient."),
    Example(
    "Might be done with mini keyboard (wrist keys), foot
      pedal, voice recognition, etc.")
  )
)
```

reqT Virtual Window example

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



Data createGuest **has**
Spec: The product shall store guest data according to virtual window 'create guest data'.

Stay#: 714

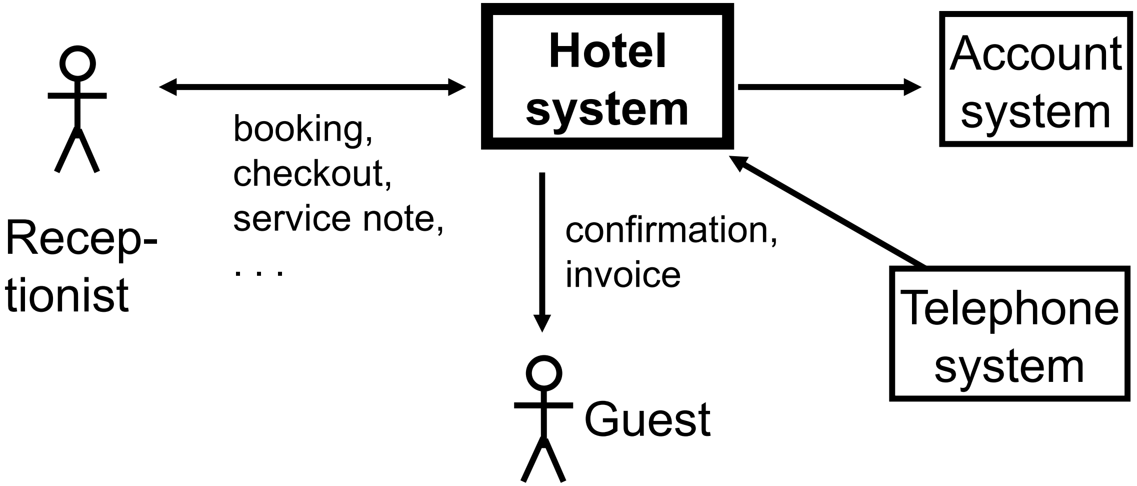
Guest

Name:	John Simpson
Address:	456 Orange Grove Victoria 3745
Payment:	Visa ▼

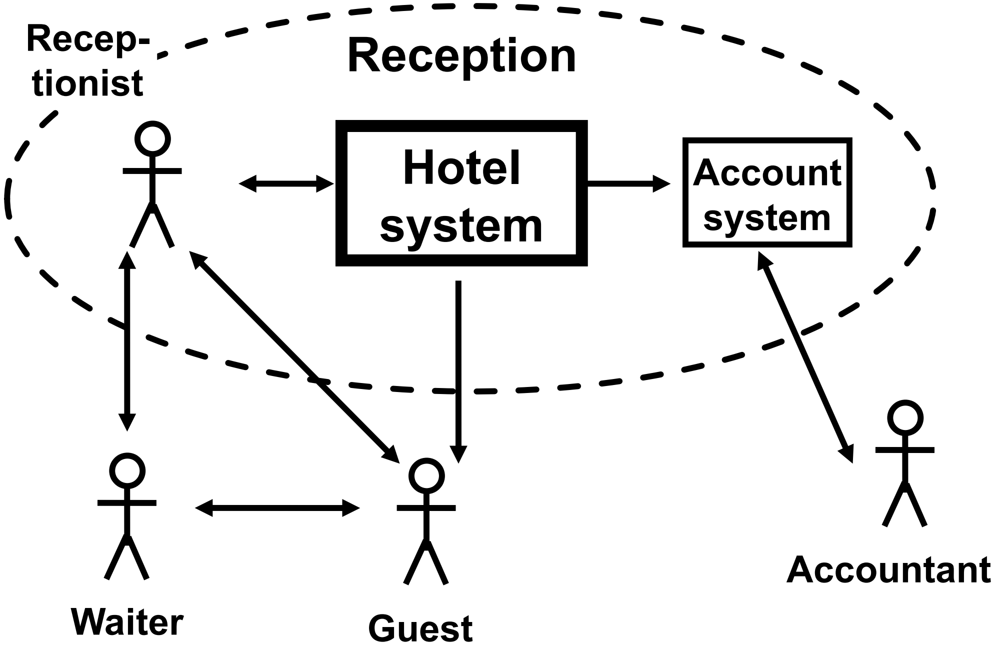
Item	#pers	
7/8 Room 12, sgl	1	600
8/8 Breakf. rest	1	40
8/8 Room 11, dbl	2	800
9/8 Breakf. room	2	120
9/8 Room 11, dbl	2	800

Fig 3.2 Context diagram

R1:
The product shall have the following interfaces:



R2 ??:
The reception domain communicates with the surroundings in this way:



From: Soren Lauesen: Software Requirements

reqT Context Diagram Example

```
Model(
  Product("HotelApp") has (
    Interface("receptionUI") has
      Actor("Receptionist"),
        Interface("guestUI") has Actor("Guest"),
        Interface("phoneAPI") has System("Telephony"),
        Interface("accountAPI") has System("Accounting")),
      Data("InterfaceIO") has (
        Interface("receptionUI") has (
          Input("booking"), Input("checkOut"),
          Output("serviceNote")),
    Interface("guestUI") has (
      Output("confirmation"), Output("invoice"))))
```

Fig 2.3 Data dictionary

Class: Guest [Notes a, b ... refer to guidelines]

The guest is the person or company who has to pay the bill. A guest has one or more stay records. A company may have none [b, c]. “Customer” is a synonym for guest, but in the database we only use “guest” [a]. The persons staying in the rooms are also called guests, but are not guests in database terms [a].

Examples

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 [d].
3.A guest with several rooms within the same stay.

Attributes

name: Text, 50 chars [h]

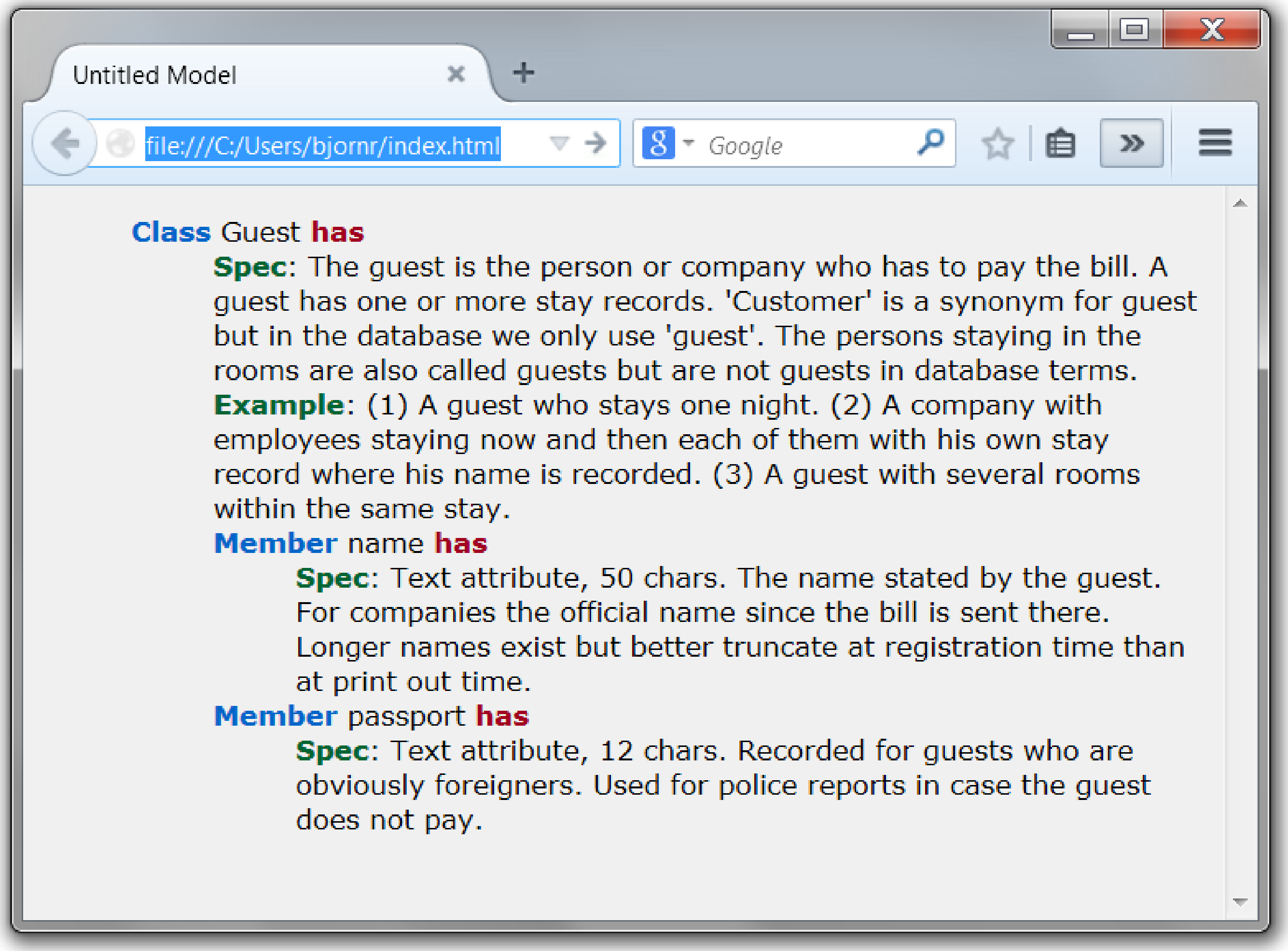
The name stated by the guest [f]. For companies the official name since the bill is sent there [g]. Longer names exist, but better truncate at registration time than at print out time [g, j].

passport: Text, 12 chars [h]

Recorded for guests who are obviously foreigners [f, i]. Used for police reports in case the guest doesn’t pay [g] . . .

From: Soren Lauesen: Software Requirements

Data dictionary example



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**

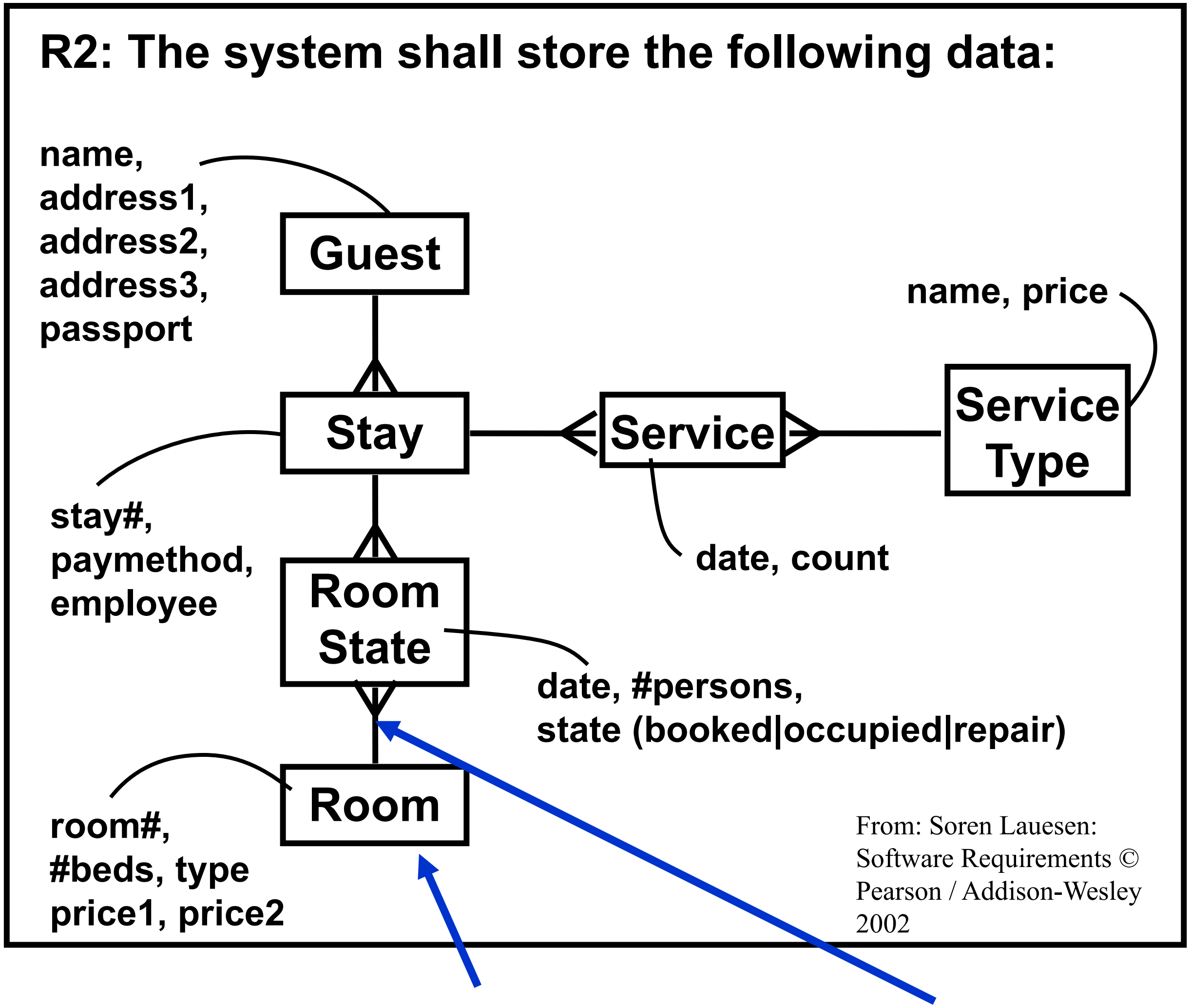
Spec: Text attribute, 50 chars. The name stated by the guest. For companies the official name since the bill is sent there. Longer names exist but better truncate at registration time than at print out time.

Member passport **has**

Spec: Text attribute, 12 chars. Recorded for guests who are obviously foreigners. Used for police reports in case the guest does not pay.

Lau: fig 2.3

Fig 2.2A Data model (E/R-diagram)



R2: The system shall store the following data:

Guest

name,
address1,
address2,
address3,
passport

Stay

stay#,
paymethod,
employee

Service

date, count

Service Type

name, price

Room State

date, #persons,
state (booked|occupied|repair)

Room

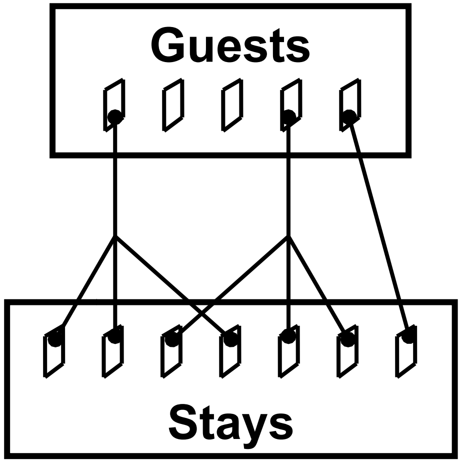
room#,
#beds, type
price1, price2

From: Soren Lauesen:
Software Requirements ©
Pearson / Addison-Wesley
2002

Entities and Relationships

One-to-many (1:m)

Each guest
connected to
zero or more stays



Guests

Stays

Each stay
connected to
one guest record

Cardinality of relations

http://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model

reqT Data model example

```
Model(
  Section("relations") has (
    Class("Guest") relatesTo (Class("Stay") has Min(1)),
    Class("Stay") relatesTo (
      Class("RoomState") has (Class("Service") has Min(1)),
    Class("ServiceType") relatesTo (Class("Service") has  Min(1)),
    Class("Room") relatesTo (Class("RoomState") has  Min(1))),
  Section("attributes") has (
    Class("Guest") has (
      Member("name"), Member("address1"), Member("address2"),
      Member("address3"), Member("passport")),
    Class("Stay") has (Member("stayId"), Member("paymethod"),
Member("employee")),
    Class("ServiceType") has (Member("name"), Member("price")),
    Class("Service") has (Member("serviceDate"), Member("serviceCount")),
    Class("Room") has (
      Member("roomId"), Member("bedCount"), Member("roomType"),
      Member("price1"), Member("price2")),
    Class("RoomState") has (
      Member("date"), Member("personCount"), Member("state")))))
```

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

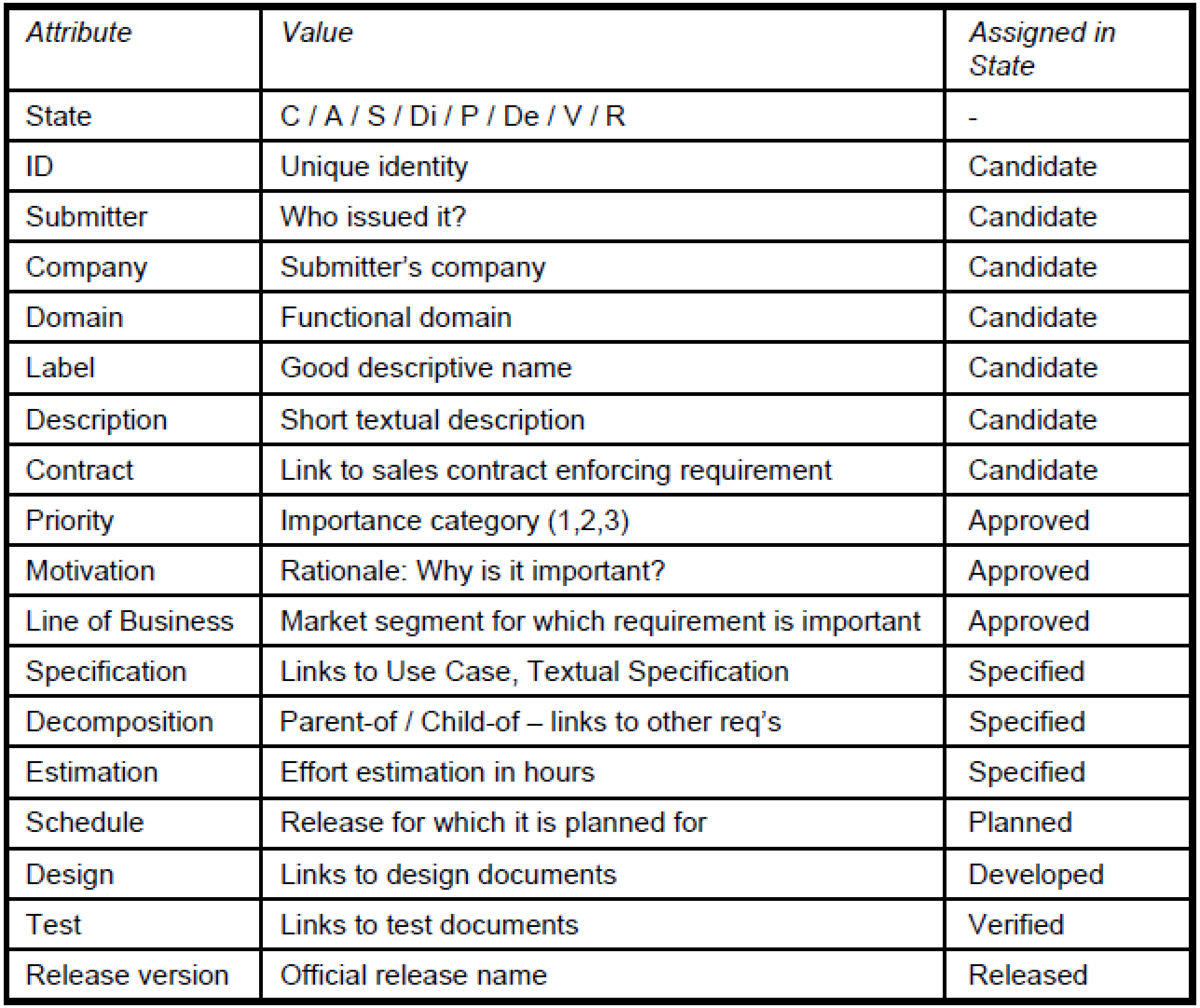
What is a ‘feature’?

Some possible definitions:

1. A textual shall-statement requirement
2. A releasable characteristic of a (software-intensive) product
3. A (high-level, coherent) bundle of requirements
4. A ‘decision unit’ that can be ‘in’ or ‘out’ of a release plan depending on:
    - What it gives (investment return)
    - What it takes (investment costs)
    - Politics, Beliefs, Loyalties, Preferences ...

```
reqT> Feature ?
res1: String = A releasable characteristic of a product. A
(high-level, coherent) bundle of requirements.
```

Example of attributes of features in a req. database



Attribute	*Value*	*Assigned in State*
State	C / A / S / Di / P / De / V / R	-
ID	Unique identity	Candidate
Submitter	Who issued it?	Candidate
Company	Submitter's company	Candidate
Domain	Functional domain	Candidate
Label	Good descriptive name	Candidate
Description	Short textual description	Candidate
Contract	Link to sales contract enforcing requirement	Candidate
Priority	Importance category (1,2,3)	Approved
Motivation	Rationale: Why is it important?	Approved
Line of Business	Market segment for which requirement is important	Approved
Specification	Links to Use Case, Textual Specification	Specified
Decomposition	Parent-of / Child-of – links to other req's	Specified
Estimation	Effort estimation in hours	Specified
Schedule	Release for which it is planned for	Planned
Design	Links to design documents	Developed
Test	Links to test documents	Verified
Release version	Official release name	Released

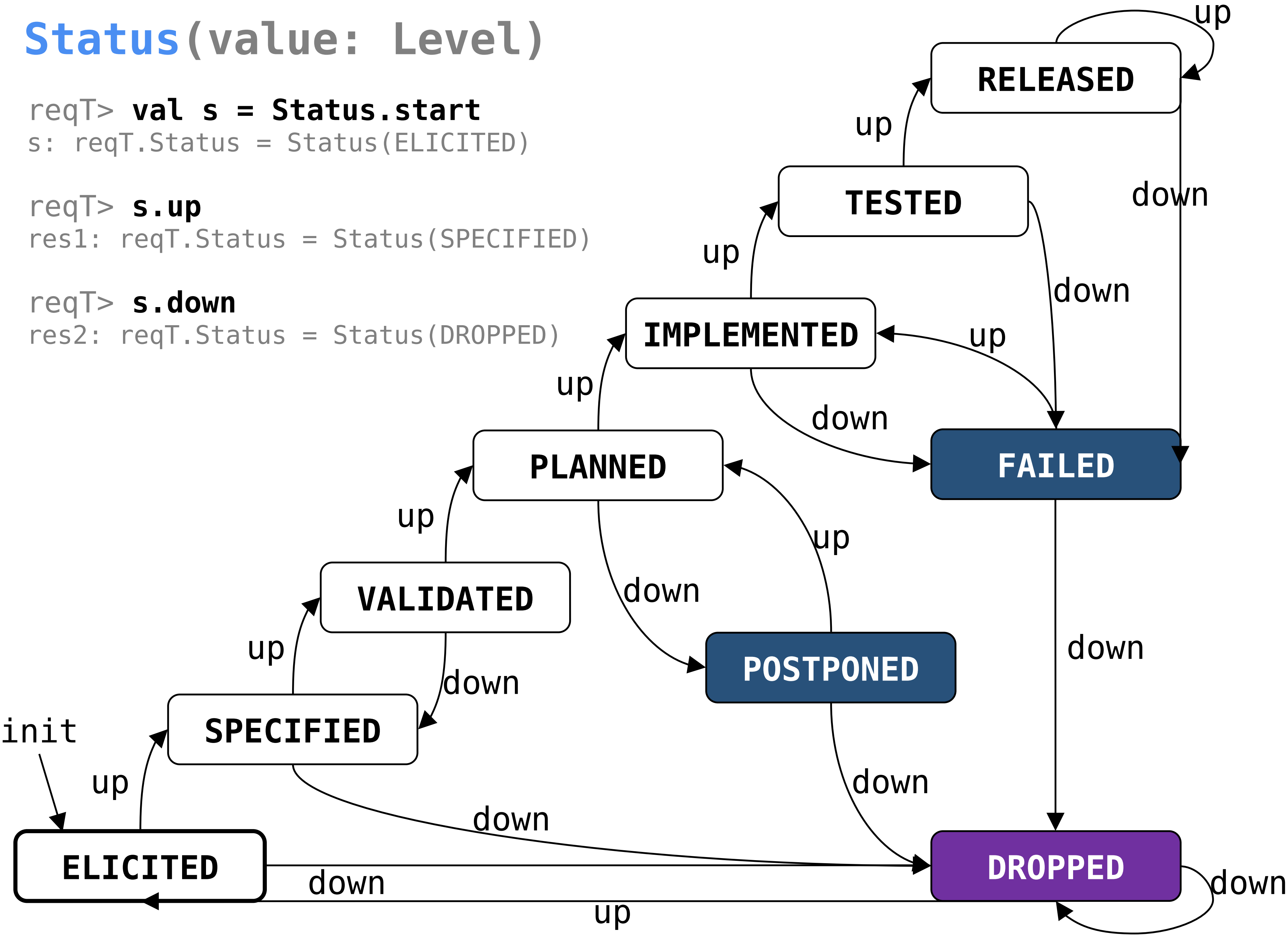
[MDRE]

Feature promotion ladder

[MDRE]

Status(value: Level)

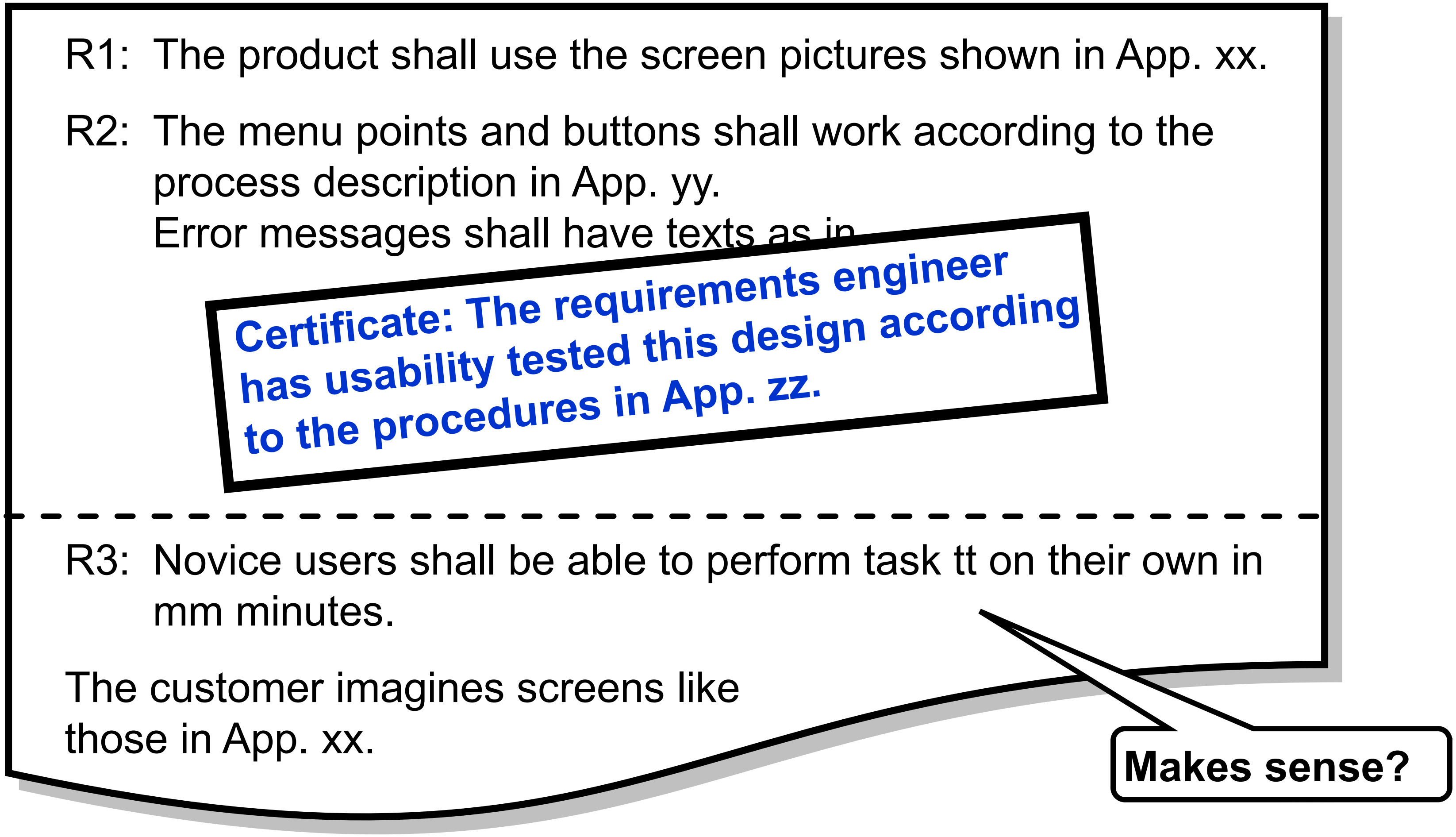
```
reqT> val s = Status.start
s: reqT.Status = Status(ELICITED)

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

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



up and down the salmon ladder

```
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
res1: reqT.Model =
Model(
  Feature("x") has Status(SPECIFIED),
  Feature("y") has Status(SPECIFIED)
)

reqT> m.up("x")
res2: reqT.Model = Model(
  Feature("x") has Status(SPECIFIED),
  Feature("y") has Status(ELICITED)
)
```

Fig 3.5A Screens & prototypes



R1: The product shall use the screen pictures shown in App. xx.

R2: The menu points and buttons shall work according to the process description in App. yy.
Error messages shall have texts as in ...

Certificate: The requirements engineer has usability tested this design according to the procedures in App. zz.

R3: Novice users shall be able to perform task tt on their own in mm minutes.

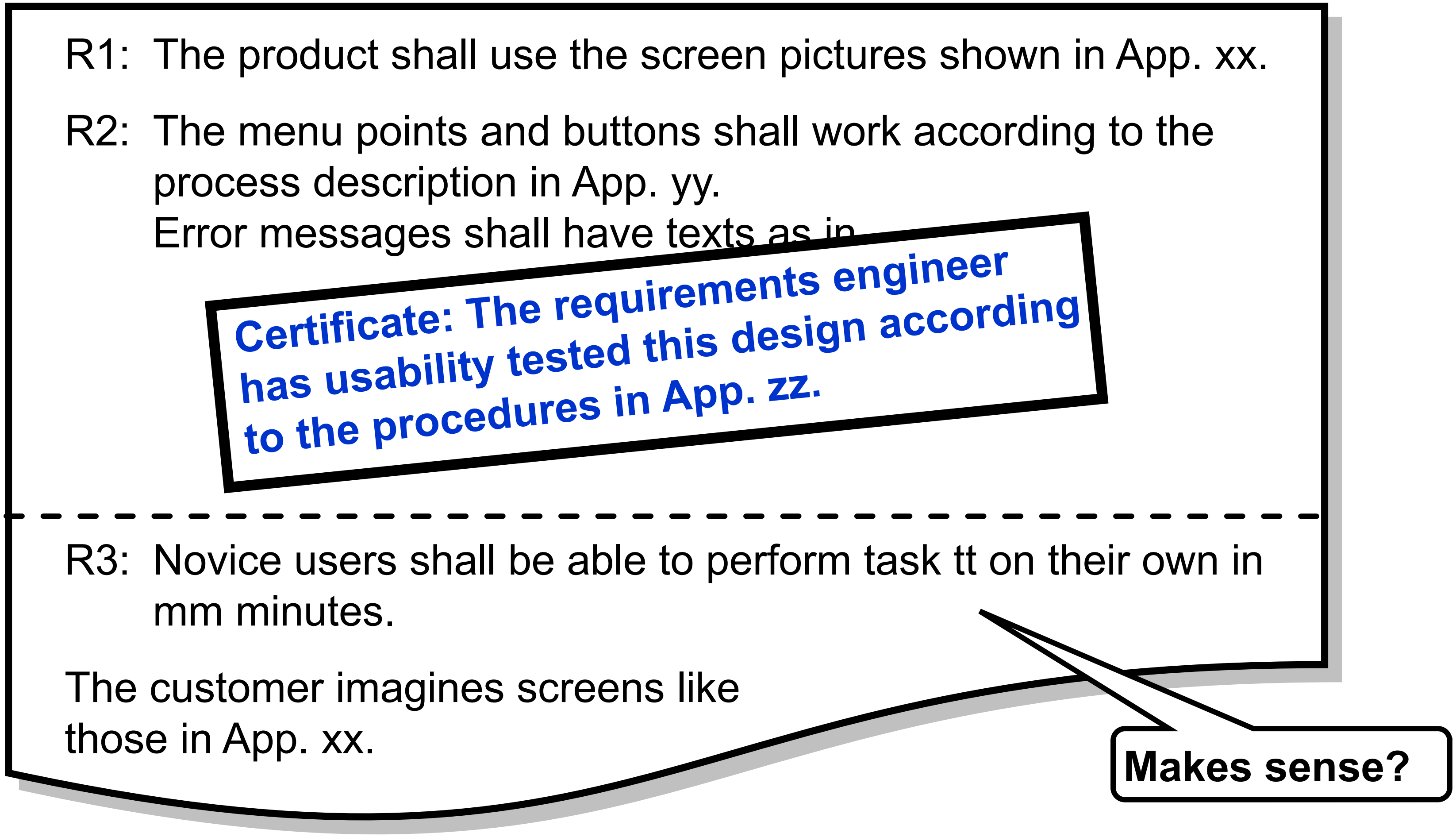
The customer imagines screens like those in App. xx.

Makes sense?

From: Soren Lauesen: Software Requirements

Fig 3.5A Screens & prototypes

```
Design("screen1") has Image("screen1.png")
```



R1: The product shall use the screen pictures shown in App. xx.

R2: The menu points and buttons shall work according to the process description in App. yy.
Error messages shall have texts as in

Certificate: The requirements engineer has usability tested this design according to the procedures in App. zz.

R3: Novice users shall be able to perform task tt on their own in mm minutes.

The customer imagines screens like those in App. xx.

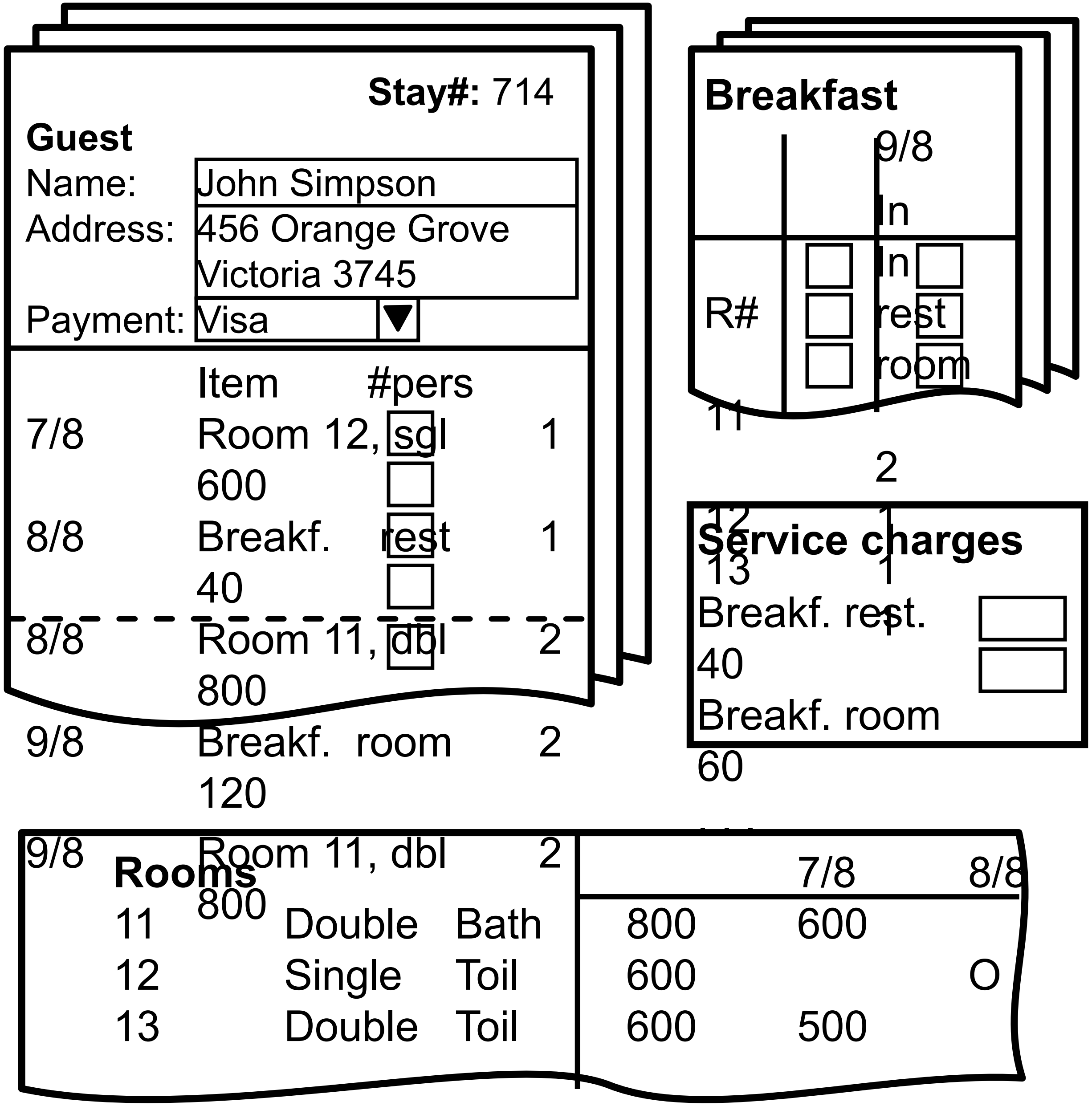
Makes sense?

From: Soren Lauesen: Software Requirements

Fig 2.5 Virtual Windows

R1: The product shall store data corresponding to the following virtual windows:

R2: The final screens shall look like the virtual windows ??



Stay#: 714

Guest

Name: John Simpson

Address: 456 Orange Grove
Victoria 3745

Payment: Visa ▼

	Item	#pers	
7/8	Room 12, sgl	1	600
8/8	Breakf. rest	1	40
8/8	Room 11, dbl	2	800
9/8	Breakf. room	2	120
9/8	Room 11, dbl	2	800

Breakfast 9/8

R#	In	In rest	room
11		2	
12		1	
13		1	1

Service charges

Breakf. rest. 40 □

Breakf. room 60 □

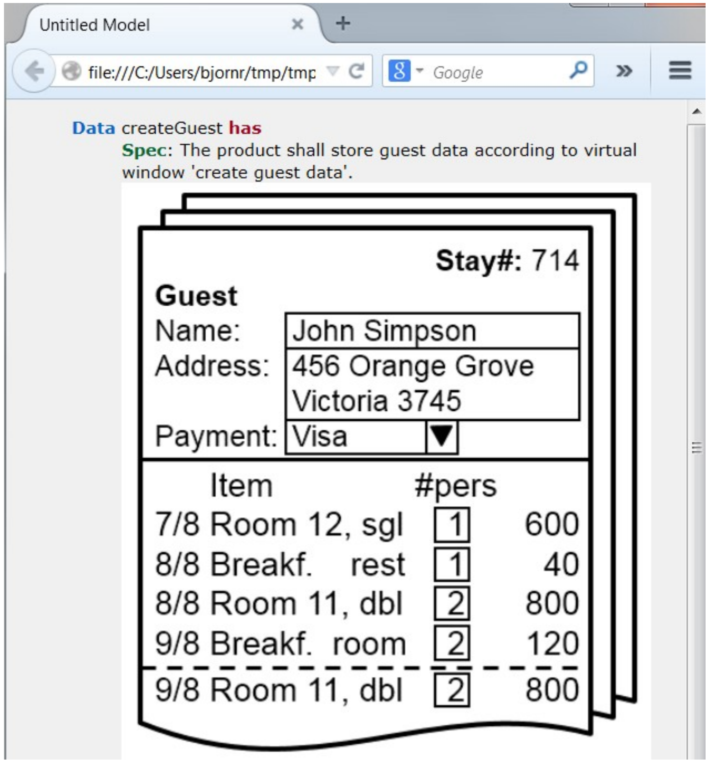
Rooms

				7/8	8/8
11	Double	Bath	800	600	
12	Single	Toil	600		O
13	Double	Toil	600	500	

From: Soren Lauesen:
Software Requirements
© Pearson / Addison-Wesley 2002

reqT Virtual Window example

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



Data createGuest **has**
Spec: The product shall store guest data according to virtual window 'create guest data'.

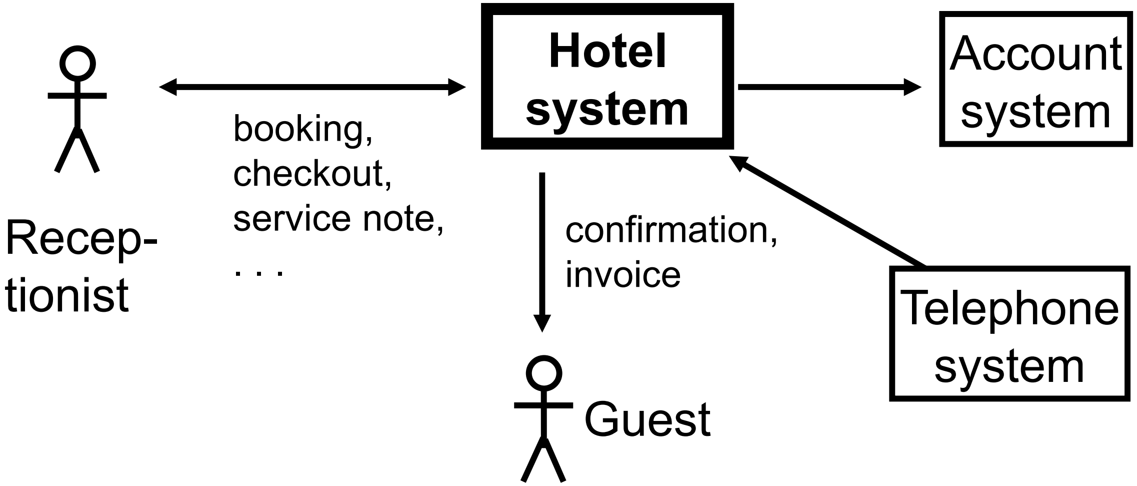
Stay#: 714

Guest
Name: John Simpson
Address: 456 Orange Grove
Victoria 3745
Payment: Visa ▼

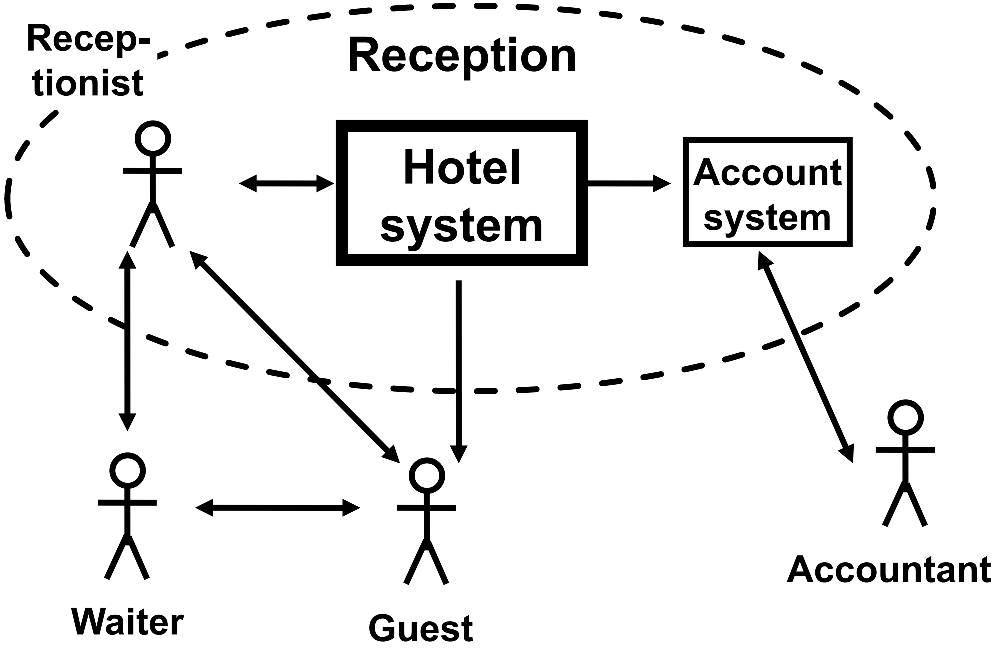
Item	#pers	
7/8 Room 12, sgl	1	600
8/8 Breakf. rest	1	40
8/8 Room 11, dbl	2	800
9/8 Breakf. room	2	120
9/8 Room 11, dbl	2	800

Fig 3.2 Context diagram

R1:
The product shall have the following interfaces:



R2 ??:
The reception domain communicates with the surroundings in this way:



From: Soren Lauesen: Software Requirements

reqT Context Diagram Example

```
Model(
  Product("HotelApp") has (
    Interface("receptionUI") has
Actor("Receptionist"),
    Interface("guestUI") has Actor("Guest"),
    Interface("phoneAPI") has System("Telephony"),
    Interface("accountAPI") has
System("Accounting")),
  Data("InterfaceIO") has (
    Interface("receptionUI") has (
      Input("booking"), Input("checkOut"),
      Output("serviceNote")),
    Interface("guestUI") has (
      Output("confirmation"), Output("invoice"))))
```

Fig 3.6A Task descriptions

Work area: 1. Reception
Service guests - small and large issues. Normally standing. Frequent interrupts. Often alone, e.g. during night.
Users: Reception experience, IT novice.

R1: The product shall support tasks 1.1 to 1.5

Task: 1.1 Booking
Purpose: Reserve room for a guest.

Task: 1.2 Checkin
Purpose: Give guest a room. Mark it as occupied. Start account.
Trigger/ Precondition: A guest arrives
Frequency: Average 0.5 checkins/room/day
Critical: Group tour with 50 guests.

Sub-tasks:
1. Find room
2. Record guest as checked in
3. Deliver key

Missing sub-task?

Variants:
1a. Guest has booked in advance
1b. No suitable room
2a. Guest recorded at booking
2b. Regular customer

Task: 1.3 Checkout
Purpose: Release room, invoice guest.
. . .

From: Soren Lauesen: Software Requirements

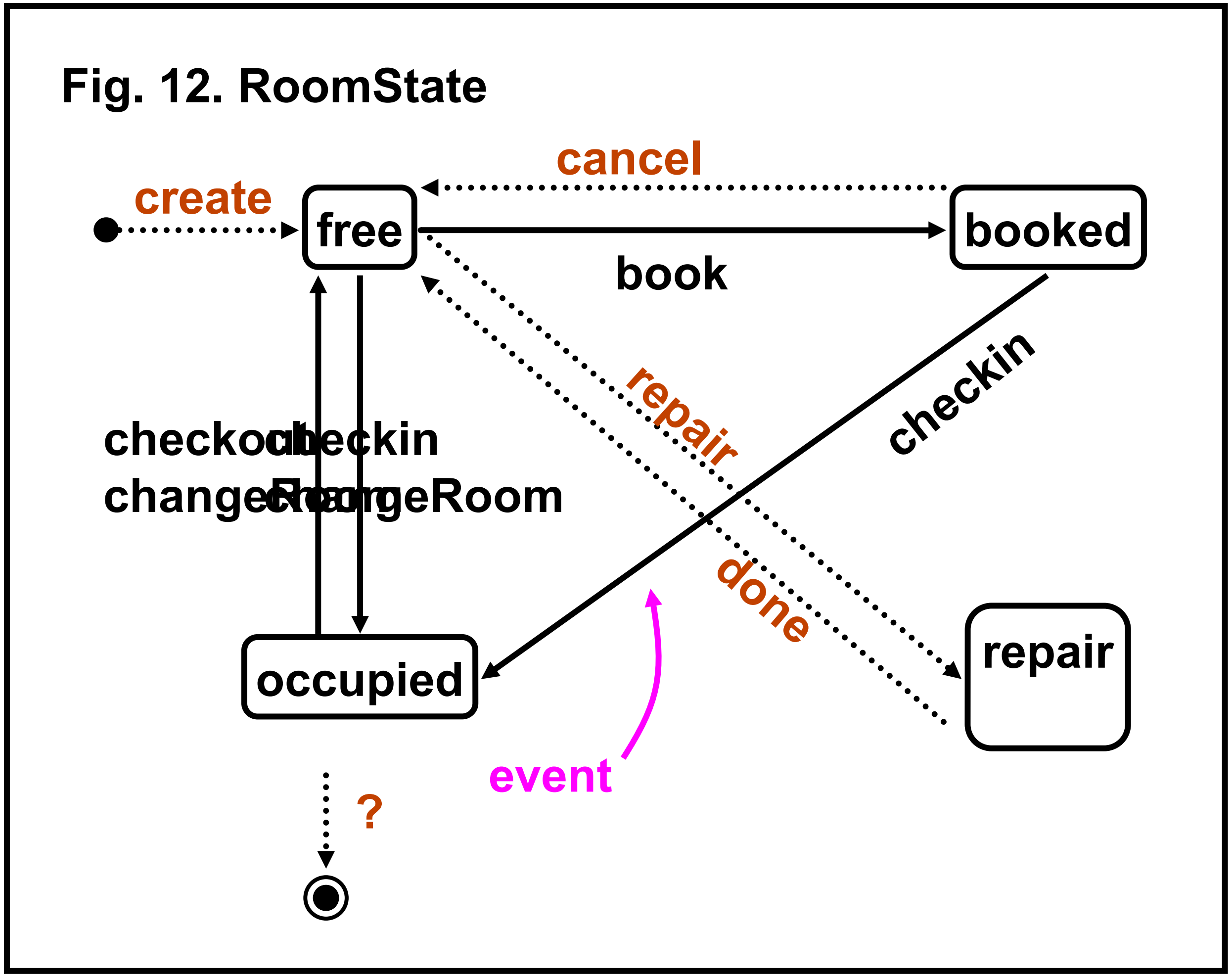
reqT Task description example

```
Model(
  Task("receptionWork") has (
    Task("booking"),
    Task("checkIn") has (
      Why("Guest wants room."),
      Frequency(3),
      Spec(
       "Give guest a room, mark it as occupied and start account.
       Frequency scale is median number of check-ins/room/week.
       Trigger: A guest arrives.
       Critical: Group tour with 50 guests."),
      Task("findRoom"),
      Task("recordGuest") has
        Spec("variants:
               a) Guest has booked in advance,
               b) No suitable room"),
      Task("deliverKey"))))
```

Fig 4.4 State diagrams

Rooms have a RoomState for each day in the planning period. The status shows whether the room is free, occupied, etc. that day.

R12: RoomState shall change as shown in Fig. 12.



From: Soren Lauesen:
Software Requirements
© Pearson / Addison-Wesley 2002

reqT State transition Model

```
Model(
  Section("roomState") has (
    Title("Room State Model"),
    State("free") has (
      Event("book")       precedes State("booked"),
      Event("checkin")    precedes State("occupied"),
      Event("changeRoom") precedes State("occupied"),
      Event("repair")     precedes State("repairing")),
    State("booked") has (
      Event("checkIn")    precedes State("occupied"),
      Event("cancel")     precedes State("free")),
    State("occupied") has (
      Event("checkout")   precedes State("free"),
      Event("changeRoom") precedes State("free")),
    State("repairing") has (
      Event("done")       precedes State("free"))))
```

Example: variability model

```
Model(
  Component("apperance") has (
    VariationPoint("color") has (
      Min(0), Max(2),
      Variant("blue"), Variant("red"), Variant("green")),
    VariationPoint("shape") has (
      Min(1), Max(1), Variant("round"), Variant("square")),
    VariationPoint("payment") has (
      Min(1), Max(2), Variant("cash"), Variant("credit")),
    VariationPoint("payment") requires Variant("cash"), /* mandatory */
    Variant("round") excludes Variant("red"),
    Variant("green") requires Variant("square")),
  Component("apperance") requires VariationPoint("shape"), /* mandatory */
  App("free") requires Component("apperance"),
  App("free") binds (
    VariationPoint("shape") binds Variant("round")),
  App("premium") requires Component("apperance"),
  App("premium") binds ( /* violating variability constraints */
    VariationPoint("color") binds (Variant("red"), Variant("green")),
    VariationPoint("shape") binds (Variant("round"), Variant("square")),
    VariationPoint("payment") binds Variant("cash")))
```

Constaint solving

```
val m = Model(
  Stakeholder("x") has Constraints(
    Var("x") > Var("y"),
    Seq(Var("x"),Var("y"))::{1 to 42}
  )
)
m.satisfy
```

Priorities and benefits

```
val m = Model(
  Stakeholder("modeler") has (
    Prio(1),
    Req("autoSave") has Benefit(25),
    Req("exportGraph") has Benefit(10),
    Req("exportTable") has Benefit(8),
    Req("autoCompletion") has Benefit(28)),
  Stakeholder("tester") has (
    Prio(2),
    Req("autoSave") has Benefit(3),
    Req("exportGraph") has Benefit(25),
    Req("exportTable") has Benefit(14),
    Req("autoCompletion") has Benefit(2)))
```

Some Model operations

```
// run in reqT:
m.collect { case s: Stakeholder => s }
m.collect { case Stakeholder(id) => id }
m.collect { case Benefit(b) => b }.sum
m.collect { case e: Entity => e.id }.
   foreach{s => println("hej "+s)}
m / Stakeholder("modeler").has / Prio
m.toHtml
m.toLatex.save("myModel.tex")
Vector(Feature("x"), Feature("y")).toModel
m.atoms
m.flat   // same as: m.atoms.toModel
m.contains(Stakeholder)
m.restrict(Stakeholder("modeler"))
m * Stakeholder("modeler")
m.transform { case Stakeholder(id) => User("Mrs. "+ id) }
```

Scenario("workInParallell")

```
// Kalle works on one model and Stina on another
val kalle = rndModel()
kalle.save("k.reqt")
val stina = rndModel()
stina.save("s.reqt")

// another day they want to load and merge
val k = Model.load("k.reqt")
val s = Model.load("s.reqt")
val merged =  k ++ s

// check if they are working on common ids (risk of clash):
kalle.ids.toSet.intersect(stina.ids.toSet)
kalle.ids.toSet & stina.ids.toSet  // same as intersect

//create latex fragment for input in main latex file
merged.toLatexBody.save("m.tex")
```

Some question for you

- How will you partition your req space?
- How will you synchronize your work?
- What entity id policy will you have?
- How will you manage versions?
- How will you build your document from requirements fragments?