

Expressing Business Rules (in Fit)

Rick Mugridge

Rimu Research Ltd

&

Department of Computer Science

University of Auckland

New Zealand

rick@mugridge.com

Contents

1. **What are business rules and why express them?**
2. **Storytests in Fit**
3. **Creating storytests**
4. **Domain Driven Design**
5. **Storytest smells and design**
6. **Other topics**
7. **Getting Fit**

1. What and Why?

- **How can we communicate and discuss requirements?**
- **How do we manage feedback and change?**
- **Major risk:**
 - **Not building what's needed**
 - **Written requirements aren't easy: "What does this mean?"**
 - **It can be hard to design a solution to a (business) problem without getting feedback**
- **Major risks:**
 - **The wrong system**
 - **Lack of focus, with wasted effort**
 - **A buggy system**
 - **A system that's hard to change – becomes legacy**

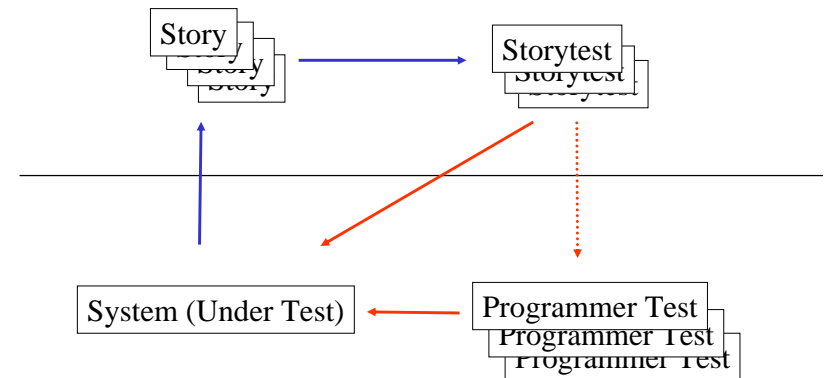
Key Points

- **Communicate with storytests**
 - **Concrete examples of business rules**
 - **Early feedback to enable understanding and design**
 - **Need to be expressive for the Customer**
 - **Drive development with storytests**
 - **Fast feedback to enable development of ideas**
 - **Automate testing**
 - **Fast feedback to find errors and enable refactoring**
 - **Regression testing as system evolves**
-
- **Pay back**
 - **Quality is cheaper**
 - **Big trap**
 - **Storytests become hard to change and become legacy**

Business Rules & Storytests

- We use storytests to express various sorts of business rules (eg, for a conference management system):
 - **Business objects**
 - Eg, Attendee, session, track, proposal, reviewer
 - **Business processes**
 - Eg, Select a session to attend, submit a proposal, review a proposal
 - **Business calculations**
 - Eg, Calculate cost of attendance, combine reviewer ratings for proposal
 - **Business constraints**
 - Eg, Can't attend two sessions at same time, can't submit a change to a proposal after the deadline, can't review own proposal, can review a proposal only once, can't view reviews for own proposal

Agile approach: SDD



Storytest Driven Development: Joshua Kerievsky, IndustrialXP

The Value of Concrete Examples

- Most Customers find it difficult or impossible to express business rules and system requirements in the abstract
- The following are often too abstract, or too imprecise for Customers:
 - Requirements documents
 - Use cases
 - UML diagrams
 - Formal specifications in Z, etc
- Trying to do this all upfront just makes it impossible!
 - Few Customers know completely where the real business value lies when they start
 - They need to evolve their understanding with feedback

The Value of Concrete Examples

- The development of examples enables the Customer to express and refine how a system could be used
 - Getting clearer about the underlying business rules that exist (analysis)
 - Getting clearer about how a system can be used to provide real business value (synthesis)
- Evolution and feedback are essential, at small and large scales
 - At the story level:
 - Develop very simple examples and refine them to add complexity to get the storytests
 - At the iteration level:
 - Change and add to the existing storytests to make the system richer

Storytests & Customers

- Storytests need to be owned by Customers
- So storytests need to:
 - Minimise formal notation
 - Encourage the expression of business rules as examples
 - precise, concise, concrete and realistic
 - Encourage understanding, discussion and communication within the whole team and with those outside
 - Be based in business design
 - Evolve as understanding of business needs change
- Storytests become a significant and valuable business resource

Storytests & Developers

- Storytests need to be understood, automated and implemented by Developers
- So storytests need to be:
 - Clear and reasonably complete, helping developers to understand the business goals and context of the system
 - Connected easily to the System Under Test for testing
 - Run independently of each other
 - Organised in test suites to run automatically in a build
 - Run quickly with good feedback
- Storytests drive the development
 - Storytests are often added in response to developers' questions during implementation

2. Storytests in Fit

- Fit was developed by Ward Cunningham in 2002, in Java
 - Versions for C#, C++, Python, Ruby, ...
- Uses one or more tables to represent a storytest and to report the results
- Fixtures mediate between the tables and the System Under Test
- Tables (and supporting text, etc) from:
 - HTML, FitNesse, spreadsheet, MS Word, etc
- FitNesse (Micah and Bob Martin) incorporates Fit with a wiki
- FitLibrary (Rick) extends Fit with various fixtures and runners
 - DoFixture, etc

Fit

The screenshot displays three windows from a Fit test run:

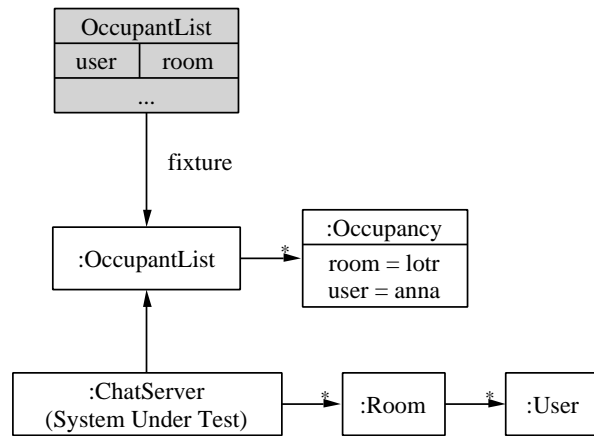
- TestDiscount - Moz...**: A table with test steps:

connect user	anna
user	anna creates lotr room
user	anna enters lotr room
users in room	lotr
name	anna
disconnect user	anna
check	occupant count lotr 0
- TestArrayWrong - Mozilla**: A table titled "DiscountGroupArrayList":

futureValue	maxCwling	minPurchase	discountPercent
low	0.00	0.00	0
low	0.00	2000.00	3
medium	500.00	600.00	3
high missing	2000.00	2000.00	10
medium	0.00	500.00	5
high surplus	2000.0	2000.0	10.0
- Invoice Table - Mozilla**: A table with invoice details:

#	Part Number	Description	Dispatch Price	Total
1	CAT 98142-00-GH	L3 Switch 32x1000T	6804.00	6804.00
2	CAT 99000-01-PH	Macronetic switch	2317.00	2317.00
Total:				9121.00
- UsersAndRooms**: A diagram showing relationships between users and rooms. Nodes include lotr, shrek, luke, anna, and madelin. Arrows indicate connections between these nodes.

Fit Fixtures Mediate Between Tables and SUT



Fit

- Defining *business processes* (state changes)
 - DoFixture, ActionFixture
- Defining *calculations and constraints*
 - ColumnFixture & CalculateFixture
- Defining *business objects* within storytests
 - SetUpFixture, ObjectFixture
- Defining given and expected *lists* and queries within storytests
 - RowFixture, SetFixture, ArrayFixture, etc
- We'll look at some simple examples

Calculation Rules & Constraints

- First row identifies what's being calculated
- Second row identifies the names of the givens and calculated () columns
- Each remaining row individually specifies the expected result for the givens of that row

months	reliable	balance	allowCredit()	creditLimit()
14	true	5000.00	true	1000.00
0	true	0.00	false	0.00
24	false	0.00	false	0.00
18	true	6000.00	false	0.00
12	true	5500.00	true expected	1000.00 expected
			false actual	0.0 actual

Business Processes

- Storytests usually develop from what a system should do
- So let's start with business processes
- We express this independently as a storytest in 3 parts:
 - Initial context (or state)
 - Given 2 connected users
 - They're in a chat room
 - Change to the system
 - A user disconnects
 - Final state
 - She is not in that room anymore

TestAutoLeaveOnDisconnect

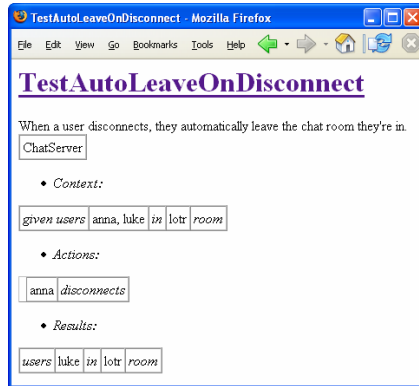
When a user disconnects, they automatically leave the chat room they're in.

ChatServer

- Context:
 - given users anna, luke in lotr room
- Actions:
 - anna disconnects
- Results:
 - users luke in lotr room

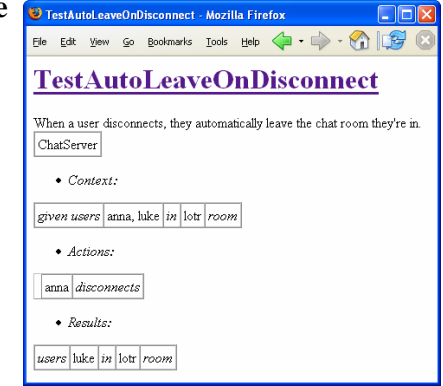
Business Processes

- The first table defines what the storytest is about (and starts the System Under Test in testing)
- Here, the other tables consist of a single row
- In a row:
- The first cell and every second one after that names the action
 - These are shown in *italics*
 - Eg, *given users in room*
- The other cells contain the data for the action
 - Eg, *anna, luke; lotr*



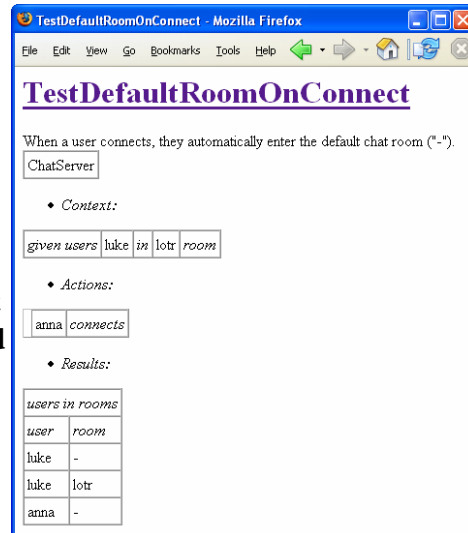
Business Processes: Context

- Notice that we define the *context* (or initial state, or *givens*) as succinctly as possible
- This introduces the business objects
- We don't carry out a sequence of actions to get to that state
- The language for defining the context is just as important as defining the changes



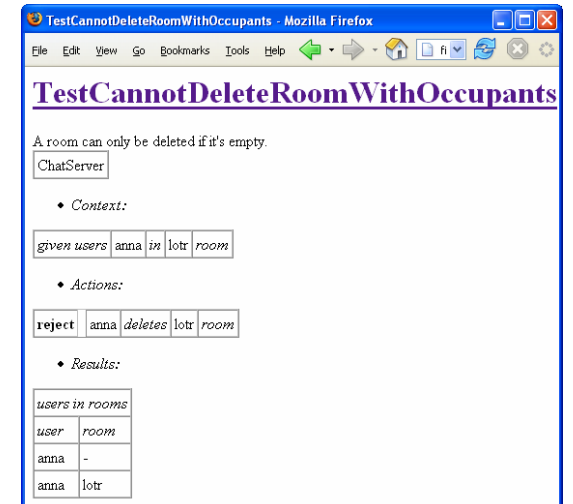
More Associations: Lists

- In this storytest, we're interested in the associations between all users and rooms, so we use a *list* table
- Consider the table with the action *users in rooms*
 - This defines how the rest of the table is interpreted
- The second row labels the data for list elements
- The next 3 rows show the associations between users and rooms



Action is rejected

- We can express that an action in a storytest is explicitly **rejected**
- This may be due to an exception in the SUT



Abstracting Calculation Rules

- After writing several related business process rules, it's tempting to alter one and change it slightly
- Copy and paste is often a sign of a missed abstraction!
- Ask, "What is the difference between these storytests?"
- The difference may be:
 - The given data varies slightly, so the results also differ
 - There is an underlying calculation rule
 - As the given data varies slightly, an action may be accepted or rejected
 - There is an underlying business constraint
 - Several business process steps are tangled and need to be separated
- Eg, fair charging

3. Developing Storytests

- It can be hard to start because you need to:
 - Pick a first example for a story
 - Start in the "middle" with a simple case
 - Start working out the language of tables
- So evolve a storytest (or related set of storytests)
 - Start simply and get underway – you can change anything
 - Simplify by ignoring some issues
 - Leave out the hard ones and the trivial ones
 - Try out examples, refining and expanding:
 - The example
 - The language; how it's expressed
 - Aim towards being concise and precise

Developing Storytests

- Who are the different types of users
 - Persona
- Role play use of the system
 - From a user perspective and/or an domain object perspective
- Tim Lister:
 - "All interesting requirements are invented"
 - "How can he know what he wants when he doesn't know what he can get?"
 - "Incrementalism is a great risk mitigation strategy"

4. Domain Driven Design

- *Domain Driven Design*, Eric Evans, Addison Wesley, 2004
- Development of a *ubiquitous language* for communication between customers and developers
 - Through a conversation of customers, users and testers/developers
- Language used in the implementation – names of classes, etc
- Design issues that come up during implementation are discussed in the whole team in terms of the domain
 - As businesspeople better understand the opportunities for supporting the business
 - As they better understand the business rules as they emerge
- The language is expected to evolve, and the whole team is atuned to a changing domain, undergoing clarification

Domain Driven Design

- **But there's a gap between the general discussions and the code**
- **Storytests can fill that gap:**
 - **They are written in terms of the business domain: business objects, associations, calculations and constraints**
 - **They help the refinement and evolution of the *ubiquitous language***
 - **The whole team can contribute:**
 - **Business knowledge**
 - **A testing perspective**
 - **An OO modelling perspective can inform**
 - **Abstraction skills: refactoring, untangling**
 - **Asking the right questions**

5. Table Design

- **Motivations:**
- **Business rules: Requirements specification**
 - **Communicate clearly in business terms**
 - **Aid understanding and clarification**
- **Storytests will change**
 - **Avoid redundancy**
 - **Keep untangled (add rather than multiply)**
- **Automated testing**
 - **Direct interpretation for testing**
 - **Independent tests, organised in suites**
 - **As part of build**
 - **Testing at multiple levels and test points**

Table Design

- **Realistic examples help everyone's understanding**
- **Express intent**
 - **Concise and precise**
- **Avoid irrelevant data**
 - **And especially premature commitment**
- **Avoid redundancy**
 - **Share common setup tables and actions**
- **Untangle storytests to express business rules independently**
- **Avoid magic numbers and other obscurities**
- **Good paths usually before bad paths**
 - **Testers more likely to be left with bad paths**

6. Other Topics

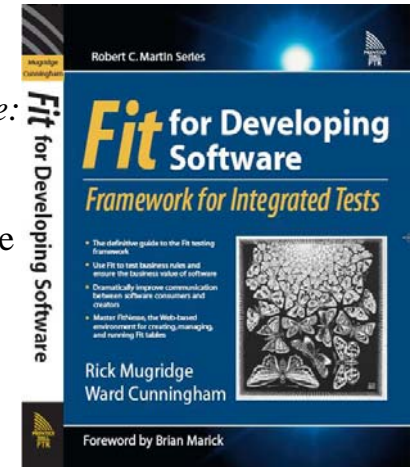
- **Handling legacy systems**
- **Fixtures testing at multiple test points**
- **Compared to Programmer Tests (unit tests)**
- **Compared to Acceptance Testing**
 - **end-to-end**
- **UI and Usability**
- **Storytests for manual processes**

Legacy

- **Trap:** write storytests in specific terms of the current system
- **Instead:** focus on storytests for expressing business rules
 - Fixtures map those storytests in testing the legacy system
 - Helps get domain model clear, which supports refactoring the legacy system
 - Tests inevitably leads to changes to the SUT to make it testable
 - In all respects, these changes may seem painful to begin with but they are beneficial beyond testing
- For help with the implementation issues, see Michael Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2005

7. Getting Fit

- Readings
 - *Fit for Developing Software: Framework for Integrated Tests*, Rick Mugridge and Ward Cunningham, Prentice Hall, 2005
 - *User Stories Applied*, Mike Cohn, Addison Wesley, 2004
 - *Domain Driven Design*, Eric Evans, Addison Wesley, 2004 (for programmers)



Getting Fit

- **Fit**
 - <http://fit.c2.com>
- **FitNesse**
 - www.fitnessse.org
- **FitLibrary**
 - <http://fitlibrary.sourceforge.net>
 - <http://sourceforge.net/projects/fitlibrary>
- **Brian Marick's website**
 - www.testing.com

Getting Fit

- **Email groups:**
 - **yahoo**
 - **fitnessse, agile-testing, industrialxp**
 - **sourceforge**
 - **fit-users, fit-dev, fitlibrary-user**
- **Workshop**
 - **W11, Hands-on Domain-Driven Acceptance Testing, Geoff Bache, Rick Mugridge and Brian Swan, Tuesday 14:00 – 17:30**