Software Engineering Project – DAT255 - lp4, 2013Chalmers University of Technology, 2013-05-20



Fredrik Einarsson  Niklas Johansson  René Niendorf   
Anders Nordin  Sofie Peters

Chalmers On The Go – the Complete Chalmers Experience

**This document…**

Software Development Document

Software Development Document

Table of Contents

Det gick inte att hitta några poster för innehållsförteckningen. **Markera de ord i dokumentet som du vill ta med i innehållsförteckningen och välj sedan ett rubrikformat under Format på fliken Start. Upprepa detta för varje rubrik som du vill ta med och infoga sedan innehållsförteckningen i dokumentet. Om du vill skapa en innehållsförteckning manuellt pekar du på ett format under Innehållsförteckning på fliken Dokumentelement och klicka sedan på nedpilen. Klicka på något av formaten under Manuell innehållsförteckning och anger sedan posterna manuellt.**

* Guidelines: Just enough documentation that it makes sense – documentation (not 20 page manual – just enough pages/words) – no page amount to fill – just motivate

**30% grade – ”final acceptance test by a customer”**

* Initial vision
  + How “good” it was
  + That we kept to the initial vision
* User stories
  + How “good” they were
  + How well they capture the vision
  + How they were realized in the application
  + Any non-functional requirements expressed as user stories “will be taken into account when determining ‘goodness’”
* General qualities
  + Performance
  + Usability
  + Stability
  + Makes sense for the domain

**10% grade – ”design decisions”**

* Design elements - UI, classes, packages, external dependencies (libraries, services (Google API)))
  + Well motivated by user stories
  + Well motivated by application needs

**15% grade – ”documentation and testing”**

* Documentation
  + Major design decision (external dependencies etc)
  + Tests for user stories
    - Through: user stories, source code, other artefacts

(30 % Development process – Post-mortem.

15 % Development and Code quality – The code itself)

Föresläsning 1, process, ev. endast för post-mortem rapport:







Föreläsning 2.

Agile Manifesto

**Individuals and interactions** over processes and tools

**Working software** over comprehensive documentation

**Customer collaboration** over contract negotiation

**Responding to change** over following a plan

*“That is, while there is value in the items on the right, we value the items on the left more.”*

Early, continous working delivery, welcome change in requirements, even late in development.

Developers and business ppl work together daily.

Face-to-face communication important – best.

Working software = primary measure of progress

Constant developing pace.

Simplicity, quiality.

The best architectures, requirements, and designs emerge from self-organizing teams.

Reflections regularly to improve the groups working methods.

Ppl primary drivers, focus on effectiveness

**Scrum**

**Committed:**

Product owner, Scrum master, Scrum Team

**Involved**:

Stakeholders, users

**Slide 31-49 – gå igenom, viktig info och mindre viktiga detaljer**

XP:



Customer tests



User stories:

One or more sentences in the everyday or business language

Captures what a user does or needs to do as part of his or her job function

Quick way of handling requirements without formalized requirement documents

Respond faster to rapidly changing real- world requirements

"As a <role>, I want <goal/desire> (so that <benefit>")

"As <who> <when> <where>, I <what> because <why>."

*“As a user, I want to search for my customers by their first and last names.”*

*“As a user closing the application, I want to be prompted to save if I have made any change in my data since the last save.”*

Beneftis with user stories:

Represent small chunks of business value that can be implemented in a period of days to weeks.

Needing very little maintenance.

Allowing projects to be broken into small increments.

Being suited to projects where the requirements are volatile or poorly understood. Iterations of discovery drive the refinement process.

Making it easier to estimate development effort.

Require close customer contact throughout the project so that the most valued parts of the software get implemented.

Ha med story map ev. i rapporten?

Föresläning 3.

Requirements

Functional (”the system should do X”) – decribes behavious or supports users/tasks/activities

Non-functional (”the system should be Y”) – qualities, not necessities, and also constraints (to limit what solutions are possible to apply)

Sammanfattning av req:

FURPS+

Functionality

Usability

Reliability

Performance

Supportability

+ Design requirements, implementation req, interface req, physical req.

F = Functional req

URPS+ = Non- functional req

URPS = qualities

+ = constratints

Non-functional just as important as functional – can give the user the sense of quality

Might be contradictory

User stories:

Card (formulera), converation (diskutera), Confirmation (konfirmera/testa/finn ett mål och se till att det uppnåtts)

Process:

Understanding the customer

Create roles - student 180 p, lärare etc.

Writing stories - for the different roles since they might be using differently

Estimate the stories – how difficultmany points

Make a release plan – what will the user be able to do on the respective releases

Points, velocity, triangulation (learning by compariing to earlier tasks with points)

Föreläsning 4, design choices:

Associtation, ex. ”A is a part of B – wing of airplane”, ”A is owned by B – plane of airline” etc.

Inheritance, composition and aggregation

GRASP:

General Responsibility Assignment Software Patterns

• •

•

Describes basic principles for object and responsibility design

as a collection of nine patterns

1. Creator – pattern to determine who reares instances of class A – B does if it…slide 67
2. Information Expert – how should responsability be distributeda mon objects? The class/object with enough information should be responsible
3. Low Coupling - How can we minimize the effects of change and support reuse? Distribute responsibility to minimize decencies between classes
4. Controller
5. High Cohesion – lower complexity by distrubtuing responsabiity to focused classes/objects
6. Polymorphism – move responsability to sub classes /types and call only the correct implementation
7. Pure Fabrication - Conceptual classes from the domain model can result in classes with low cohesion and high coupling. Invent new classes that are not part of the domain and assign these specific responsibilities.
8. Indirection - Remove direct coupling? Distribute responsibility to intermediary objects that manage the connection between two or more objects
9. Law of Demeter (Don’t talk to strangers) – only model relevant associations

**Protected variations**

How can classes, subsystems and systems be constructed to minimize the effect of instability and change on other parts?

Identify points where variation can happen or points that could be unstable

Distribute responsibility to create stable interfaces

**Variation points**

Separate

evolution points The first one should always be handled

The second one relates to future changes that may happen

can cost a lot of time and effort, and the evolution might never happen

Law of Demeter can be considered a part of Protected Variations

Variations happens in structure, and this can be hidden by a stable interface

Bra med Controllerklass

1. Product,Vision and Requirements

We will also consider how well the finished application captures the vision, and

How well the stories capture the vision, and

How these are realized in the application.

2. Design Decisions

•

•

•

Various design elements, such as user interface, classes and packages, as well as external dependencies (libraries and services) will be considered.

Should be motivated by user stories

All external dependencies should be explicitly motivated (except Android)

4. Documentation and Testing

•

• • • • • •

Major design decision, such as external dependencies, should be documented.

comments build and install instructions getting started “user stories” design document motivation for design decisions

Structure:

What, why

How

What is the product

Differences from original vision and motivate why differences

Code:

Github usage

Usage of design patterna (GRASP etc)

Comments – not only “this is” but also “why this is”

Overall:

Performance

Stability

Usability “Should be reasonable performance – not take 5 sek to react etc”

Content:

Motivate external dependencies (why Google? Why not something else? – Does not need a lot of motivation, enough with “we needed X, this gave X and is recommended by Y”)

Motivate any weaknesses/missing or removed functionality

What was the original idea

How good was the idea

How was the vision/idea translated into user stories

How was the vision/idea realized with the code

What differed from the original vision and why – motivate

Will the user uppleva the intended values

Only a few examples of user stories – not all of them (just to show what kind of stories we used and how they look)

A few unit tests – very nice if they can be connected to user stories, but not a requirement

Only tests for the final release, not for previous releases that are not current now

Nice with a release history – brief

What problems came up and how did we sole/hanterade/kringgick dem?

Change log

**Application development and feature decisions**

**Product backlog**

Product backlog over features (may be extracted from PivotalTracker)

**Design decisions**

Design decisions and description of use of external code (what, how and why)

**Change log**

”Log or record of changes made to a project, such as a [website](http://en.wikipedia.org/wiki/Website) or software project, usually including such records as bug fixes, new features, etc. Some [open source](http://en.wikipedia.org/wiki/Open_source) projects include a changelog as one of the top level files in their distribution.” (Wikipedia)

**Testing**

**Acceptance tests**

Acceptance tests have been run for the different user stories.

Those tested concern design, possibilities and restrictions on the map and application features.

**Test details**

User stories, examples:

* As a user who opens the program, I want to have a fixed view on the Chalmers map, always with the same starting coordinates.
* As a user, I should not be able to zoom out to see larger areas than that of Chalmers.
* As a user, I want to be able to search for buildings and the result should be all rooms.
* As a user, when I type in the search for an item, a dropdown menu with word-completed suggestions should appear.
* As a user I want to see where I am on campus with just one click.
* As a user, I want to be able to search for two separate locations, and get the path between them.
* As a user I want to be able to have a Checkbox Menu in which I can choose between lecture halls, computer rooms and group rooms.

**Unit tests**

All database methods in the DAO (Data Access Object) class have been unit tested, using the public Assert class.

**Test details**

* Insertion and getting in table 4 (buildings table)
  + insertIntoTable4 and getAllFromTable4 were tested together:
    - A building name (String) was inserted into table 4 via insertIntoTable4 and fetched with getAllFromTable4
* Insertion and getting in table 2 (room types table)
  + insertIntoTable2 and getAllFromTable2 were tested together:
    - Three room types (String) were inserted into table 2 via insertIntoTable2 and fetched with getAllFromTable2
* Insertion and getting in table 1 (coordinates and buildings table)
  + insertIntoTable1 and getClosestEntry were tested together:
    - A pair of coordinates (Double) and a building name (String) were inserted into table 1 via insertIntoTable1.
    - The coordinates (Double) were used to create an object (LatLng) containing latitude and longitude.
    - The object (LatLng) and the building name (String) served as input in getClosestEntry.
    - The result of getClosestEntry (LatLng) and the object (LatLng) containing the coordinates were compared and found to be equal.
    - Calculating the closest entry
  + insertIntoTable1 and getClosestEntry were tested together:
    - An object (LatLng) containing zero coordinates, the current coordinates, were created.
    - Five different coordinate pairs (Double) and a building name (String) were inserted into table 1 via insertIntoTable1.
    - The pair of coordinates (Double) closest to the zero coordinates, were in addition used to create an object (LatLng) containing latitude and longitude.
    - The zero coordinates object (LatLng) and the building name (String) served as input in getClosestEntry.
    - The result of getClosestEntry (LatLng) and the closest coordinate pair object (LatLng) were compared and found to be equal.
    - Insertion and getting in table 3 (room name, coordinates, room type, building and floor table)
  + insertIntoTable3 and getRoomCoordinates were tested together:
    - A pair of coordinates (Double) were used to create an object (LatLng) containing latitude and longitude.
    - A room type (String) was inserted into table 2 via insertIntoTable2.
    - A building name (String) was inserted into table 4 via insertIntoTable4.
    - The room name (String), the coordinates (Double, the room type (String), the building name (String) and a floor (String) were inserted into table 3 via insertIntoTable3.
    - The room name (String) served as input in getRoomCoordinates.
    - The result of getRoomCoordinates (LatLng) and the object (LatLng) containing the coordinates were compared and found to be equal.
    - Getting all rooms in a specific building
  + insertIntoTable2, insertIntoTable3, insertIntoTable4 and getAllRoomsInBuilding were tested together:
    - A room type (String) was inserted into table 2 via insertIntoTable2.
    - A real building name (String) was inserted into table 4 via insertIntoTable4
    - A false building name (String) was inserted the same way.
    - Room name1 (String), coordinate pair1 (Double), the room type (String), the true building name (String) and floor1 (String) were inserted into table 3 via insertIntoTable3.
    - Room name2 (String), coordinate pair1 (Double), the room type (String), the true building name (String) and floor2 (String) were inserted into table 3 via insertIntoTable3.
    - A false room name (String), coordinate pair1 (Double), the room type (String), the false building name (String) and floor1 (String) were inserted into table 3 via insertIntoTable3.
    - The true building name (String) served as input in getAllRoomsInBuilding.
    - The result of getAllRoomsInBuilding (ArrayList<String>) was tested using methods size and contains, and found to be satisfactory.
    - Getting all rooms with a specific type
  + insertIntoTable2, insertIntoTable3, insertIntoTable4 and getAllRoomsInBuilding were tested together:
    - A room type (String) was inserted into table 2 via insertIntoTable2.
    - A false room type (String) was inserted the same way.
    - A building name (String) was inserted into table 4 via insertIntoTable4
    - Room name1 (String), a coordinate pair1 (Double), the true room type (String), the building name (String) and floor1 (String) were inserted into table 3 via insertIntoTable3.
    - Room name2 (String), coordinate pair1 (Double), the true room type (String), the building name (String) and floor2 (String) were inserted into table 3 via insertIntoTable3.
    - A false room name (String), coordinate pair1 (Double), the false room type (String), the building name (String) and floor1 (String) were inserted into table 3 via insertIntoTable3.
    - The building name (String) served as input in getAllRoomsInBuilding.
    - The result of getAllRoomsInBuilding (ArrayList<String>) was tested using methods size and contains, and found to be satisfactory.
    - Getting suggestions
  + insertIntoTable2, insertIntoTable3, insertIntoTable4 and suggestions were tested together:
    - A room type (String) was inserted into table 2 via insertIntoTable2.
    - A building name (String) was inserted into table 4 via insertIntoTable4
    - A room name (String), a coordinate pair (Double), the room type (String), the building name (String) and a floor (String) were inserted into table 3 via insertIntoTable3.
    - Different strings of letters matching the strings in table 3 served as input in suggestions.
    - The result of suggestions (ArrayList<String>) was tested using methods for size and null, and found to be satisfactory.
    - Getting room names
  + insertIntoTable3 and getName were tested together:
    - A room name (String), a coordinate pair (Double), a room type (String), a building name (String) and a floor (String) were inserted into table 3 via insertIntoTable3.
    - The room name (String) served as input in getType.
    - The result of getName (String) and the room name were compared and found to be equal.
    - Getting room types
  + insertIntoTable3 and getType were tested together:
    - A room name (String), a coordinate pair (Double), a room type (String), a building name (String) and a floor (String) were inserted into table 3 via insertIntoTable3.
    - The room name (String) served as input in getType.
    - The result of getType (String) and the room type were compared and found to be equal.
    - Getting floor
  + insertIntoTable3 and getFloor were tested together:
    - A room name (String), a coordinate pair (Double), a room type (String), a building name (String) and a floor (String) were inserted into table 3 via insertIntoTable3.
    - The room name (String) served as input in getFloor.
    - The result of getFloor (String) and the floor were compared and found to be equal.

Under testing:

Verify and validate

Test case = input and expected output

Test suite = a set of test cases

Coverage = “quality measure” of tests – how many lines/expressions of code are tested/covered?

Requirements Analysis – acceptance test

Architectural design – system test

Subsyetm design – integration test

Detailed design – module test

Implementation – unit test (often with TDD)

Test to pass – simple test cases

Test to fail – evil test cases

Statement coverage (alla statements/tester gårs igenom) and branch coverage (alla decisions gårs igenom/alla möjliga vägar som kan tas) – find paths through the program (while white box-testing) and make sure to cover them

Acceptance testing – tests a user story (from a “custome” perspective) – the customer can accept or reject



Document What?

1. Requirements - Statements that identify attributes, capabilities, characteristics, or qualities of a system.This is the foundation for what shall be or has been implemented.
2. Architecture/Design - Overview of software. Includes relations to an environment and construction principles to be used in design of software components.
3. Technical - Documentation of code, algorithms, interfaces, and APIs.

Design decisions Vision Statement Operations documentation Project overview Requirements document Support documentation System Documentation User documentation

Burn down chart btw!

Agile development är målet



With high quality source code and a test suite to back it up you need a lot less system documentation.

Developers rarely trust the documentation, particularly detailed documentation because it's usually out of sync with the code

Ask whether you NEED the documentation, not whether you want it

Document stable things, not speculative things