ADVANCED SOFTWARE DESIGN

LECTURE 2

ANALYSIS

Dave Clarke
AT THE END OF THE LECTURE ...

- You will know about
  - several software development processes
  - risk management
  - requirements analysis and how to write use cases
QUESTIONS

• How do you develop software?

• How have you seen software developed?
  • Internship, open source project, via work, etc.
SOFTWARE DEVELOPMENT PROCESSES
SOFTWARE DEVELOPMENT ACTIVITIES

- requirements elicitation
- analysis
- system design
- object design
- implementation
- testing
- maintenance
AD HOC CODING
“METHODOLOGY”

• Does not scale to large sized projects
• Does not scale to large development teams.
THE WATERFALL MODEL
WATERFALL MODEL

- requirements elicitation
- analysis
- system design
- object design
- implementation
- testing
- maintenance
CRITIQUE OF WATERFALL MODEL

• Responds poorly to changes and problems.

• Substantial upfront documentation.

• Assumes a fixed specification – may not be what customer wants.

• Fixes come very late – costlier to fix later time.
COST OF FIXING BUGS

Source: Applied Software Measurement, Capers Jones, 1996
ITERATIVE & AGILE DEVELOPMENT PROCESSES
FACTS OF LIFE

• Requirements change, changes break existing design.

• Coding up a design suggests flaws in design.

• Testing identifies flaws in code – which could be design flaws.

• Maintenance requires not only fixes but new features.
PHILOSOPHY

• Embrace change.

• Don’t do too much, too soon.

• Individuals and interactions over processes and tools.

• Working software over comprehensive documentation.

• Customer collaboration over contract negotiation.

• Responding to change over following a plan.
Early coding, early testing of partial system in repeating cycles.

Development begins before all requirements are defined in detail.

Feedback is used to clarify evolving specification.
BENEFITS

• Early rather than late mitigation of high risks.
• Early visible progress
• Managed complexity – the team is not overwhelmed by “analysis paralysis” or very long and complex steps.
• Early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders.
• Feedback can also improve development process itself.
BENEFITS

• Early rather than late mitigation of high risks.

• Early visible progress

• Managed complexity – the team is not overwhelmed by “analysis paralysis” or very long and complex steps.

• Early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders.

• Feedback can also improve development process itself.

Less project failure, better productivity, and lower defect rates.
AGILE MANIFESTO

- Customer satisfaction by rapid delivery of useful software
- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Close, daily cooperation between business people and developers
- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity—the art of maximizing the amount of work not done—is essential
- Self-organizing teams
- Regular adaptation to changing circumstances
EXTREME PROGRAMMING
EXTREME PROGRAMMING (IN A NUTSHELL)

• Customer driven – frequent communication with customer

• Pair programming

• Extensive code review

• Unit testing

• Avoiding implementing features until they are needed

• Expect changes in requirements

• Constant refactoring to keep code supple (and well designed)
Intermission
MANAGING RISKS
RISKS

- Personnel shortfalls
- Unrealistic schedules/budgets
- Developing wrong software
- Developing wrong UI
- Gold plating
- Continuous stream of requirements changes
- Shortfalls in external components
- Shortfalls in externally performed tasks (outsourcing)
- Real-time performance shortfalls
- Straining computer science capabilities
RISK-DRIVEN PLANNING

Address high risk requirements early.

• Identify risk – locate risks before they become problems

• Risk analysis – evaluate (impact, probability, timeframe), classify and priorities

• Planning – choose risk mitigation actions

• Tracking and control – monitor risks, reassess and change plans
PRIORITISATION

• Trade-offs – cost-benefit analysis

• Importance to customer – cost to implement – risk involved in building it

• “Must be included”, “must be excluded”, “nice to have”

• Conflicts – different stakeholders have different views.

• Impact of adding feature later rather than now
RISK-DRIVEN PLANNING FOR YOU!

• For each requirement, consider:
  • architectural significance
  • (high) business value – stated as important in the spec
  • (high) risk

• Rank them in terms of how they will affect your success.
• Address such requirements early.
REQUIREMENTS AND USE CASES
Requirements are capabilities and conditions to which the system –and more broadly, the project– must conform.

Requirements analysis is about describing problems.
Some notion that there is a *problem* that needs solving

- dissatisfaction with the current state of affairs
- a new business opportunity
- a potential saving of cost, time, resource usage, etc.
REQUIREMENTS ANALYST: AN AGENT FOR CHANGE

The requirements analyst must identify the problem/opportunity

• Which problem needs to be solved?
  Identify problem **Boundaries**

• Where is the problem?
  Understand the **Context/Problem Domain**

• Whose problem is it?
  Identify **Stakeholders**

• Why does it need solving?
  Identify the stakeholders’ **Goals**
REQUIREMENTS ANALYST

• How might a software system help? Collect some **Scenarios**

• When does it need solving? Identify **Development Constraints**

• What might prevent us solving it? Identify **Feasibility** and **Risk**

Also become an expert in the problem domain – the **intelligent ignoramus**
KEY SKILL

Separate **problem** from **solution** descriptions.

Requirements is about the problem.

Design is about the solution.

*Problem description will never be perfect (or fixed).*

*Perfecting it will not be cost effective.*
FINDING REQUIREMENTS: FURPS+

• **Functional** – features, capability, security

• **Usability** – human factors, help, documentation

• **Reliability** – frequency of failure, recoverability, predictability

• **Performance** – response times, throughput, accuracy, availability, resource usage

• **Supportability** – adaptability, maintainability, internationalisation, configurability
FURPS+: THE REST

• **Implementation** – resource limitations, languages and tools, hardware, ...

• **Interface** – constraints imposed by interfacing with external systems

• **Operations** – system management in its operational setting

• **Packaging** – for example, a physical box, or app store

• **Legal** – licensing and so forth.
FUNCTIONAL VS NON-FUNCTIONAL REQUIREMENTS

YOU'RE 10 MINUTES LATE, JIMMY!

IF ONLY JIMMY'D LEARNED ABOUT NONFUNCTIONAL REQUIREMENTS...

THIS THING'S FALLING APART! LOOK, THAT WHEEL'S COMING OFF.

JEEZ, LADY, IT'S GOT FOUR WHEELS AND IT GOES. WHAT MORE DO YOU WANT?

AND WHERE WERE YOU LAST SATURDAY WHEN I WANTED TO GO TO BECKY'S BIRTHDAY PARTY?
FUNCTIONAL VS NON-FUNCTIONAL REQUIREMENTS

functional = behavioural

non-functional = quality: usability, reliability, performance, supportability

non-functional requirements strongly influence software architecture
REQUIREMENTS ELICITATION
ELICITATION TECHNIQUES

- **Background Reading** – company reports, organisational charts, etc.

- **Hard data collection** – forms, invoices, reports used for decision making, etc.

- **Interviews** – open-ended discussion with future system users

- **Questionnaires** – can quickly get information from large groups

- **Group Techniques** – focus groups, brainstorming

- **Participant Observation** – watch people in action

Rather than base requirements on pre-defined concepts, use users’ own concepts
REQUIREMENTS ELICITATION

PROBLEMS

• **Thin spread of domain knowledge** – not written down, distributed across many sources, conflicting.

• **Tacit knowledge** – people find it hard to describe the knowledge they regularly use

• **Limited observability** – problem owners having problem coping with existing system; presence of observer may change problem

• **Bias** – people may not tell you all you need to know
DOCUMENTING REQUIREMENTS
DOCUMENTING AND ORGANISING REQUIREMENTS

• **Use-case model** – a set of typical scenarios of using a system – functional.

• **Supplementary specification** – everything not in the use cases – primarily non-functional requirements.

• **Glossary**.

• **Data dictionary** (requirements on data such as validation).

• **Vision** – high level summary of the business use case for the project.

• **Business/domain rules** – requirements and policies that transcend one software project and are required by business or domain – tax laws.
USE CASES

• Text stories describing what the software does – scenarios.

• Visually represented using UML Use Case Diagrams.
ACTORS, SCENARIOS, USE CASES

• **Actors** – something with behaviour, including system itself
  - Primary actor – has goals fulfilled through using the services of the system
  - Supporting actor – provides a service for the system
  - Offstage actor – has an interest in the behaviour of the system (tax office)

• **Scenarios** – a specific sequence of actions and interactions between actors and the system – a *use case instance*

• **Use case** – collection of related success and failure scenarios that describe an actor using a system to support a goal
USE CASE FORMATS

• **Brief** – terse, one paragraph summary, usually of the main success scenario (one from two slides ago)

• **Casual** – informal paragraph format. Multiple paragraphs that cover various scenarios.

• **Fully dressed** – all steps and variations are written in detail and there are supporting sections such as preconditions and success guarantees.
Process sale: a customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.
Fully Dressed Example: Process Sale in a POS System

Fully dressed use cases show more detail and are structured; they are useful order to obtain a deep understanding of the goals, tasks, and requirements.

Use Case UC1: Process Sale
Primary Actor: Cashier

Stakeholders and Interests:
- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
- Customer: Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns.
- Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update to accounting and inventory.
- Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
- Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.

Preconditions: Cashier is identified and authenticated.

Success Guarantee (Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario (or Basic Flow):
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.
   *Cashier repeats steps 3-4 until indicates done.*
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):
*a. At any time, System fails:
   To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.
   1. Cashier restarts System, logs in, and requests recovery of prior state.
   2. System reconstructs prior state.
   2a. System detects anomalies preventing recovery:
      1. System signals error to the Cashier, records the error, and enters a clean state.
      2. Cashier starts a new sale.*
USE CASE DIAGRAMS
USE CASE DIAGRAMS ARE NOT REQUIREMENTS.

USE CASES ARE TEXT.
PUTTING INTO PRACTICE
EXAMPLE PROCESS: INITIAL PHASE

Early requirements analysis: **identify** a few *brief* use cases to get a quick sense of subject and scope – is project even feasible?

After many uses cases have been identified (80%) and written in a brief format, **select** a few (10%) *(which?)* and write out in more detail.

This involves **planning!** *How much can be done in this iteration?*

Develop domain model, elaborate into classes.

For each use case, develop appropriate models of interaction between actors involved.

Implement (in practice, completely for given use cases) – *not* a prototype.
EXAMPLE PROCESS: ITERATION

Each iteration is closer to 3 weeks in duration than 3 months

**Use** results of previous to feedback on requirements and design.

- Unclear, unfeasible, conflicting requirements
- Design does not work as well as expected
- Re-prioritise tasks
- Address externalities – shifts in requirements, etc.

Subsequent iterations:

- **Plan** what is to be done in iteration
- more requirements, more detailed requirements, extend design, implement
WHAT WE COVERED

• Now you know about
  • several software development processes
  • risk management
  • requirements analysis and how to write use cases