Adaptive Software Engineering
G22.3033-007

Session 6 - Main Theme
Planning and Managing Requirements

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

- Requirements Development Methodology
- Specifying Requirements
- Eliciting Accurate Requirements
- Documenting Business Requirements
- Defining User Requirements
- Validating Requirements
- Achieving Requirements Traceability
- Managing Changing Requirements
- Reviews, Walkthroughs, and Inspections
- Agile Requirements Engineering
- Summary
  - Course Project (Part 3)
  - Readings
Summary of Previous Session

- Session 4 Review
  - GRASP Patterns
  - Sample OCL Problem
  - Using UML to Model Application of XML
- Software Engineering Tools Primer
  - Build Tools (e.g., Ant)
  - Continuous Build Process Frameworks (e.g., CruiseControl)
  - Unit Testing Frameworks (e.g., jUnit)
  - Refactoring Browsers (e.g., IntelliJ’s IDEA)
  - Selecting Appropriate Tools
- Summary
  - Individual Assignment #4 & Team Assignment #1
  - Course Project (Part 2) - Ongoing
  - Readings

Review

Ensuring Coverage, Low Coupling, and High Cohesion in UML Models
Low Coupling

- Coupling: it is a measure of how strongly one element is connected to, has knowledge of, or relies upon other elements.
- A class with high coupling depends on many other classes (libraries, tools).
- Problems because of a design with high coupling:
  - Changes in related classes force local changes.
  - Harder to understand in isolation; need to understand other classes.
  - Harder to reuse because it requires additional presence of other classes.
- Problem: How to support low dependency, low change impact and increased reuse?
- Solution: Assign a “responsibility” so that coupling remains low.

Low Coupling

- Assume we need to create a Payment instance and associate it with the Sale.
- What class should be responsible for this?
- By Creator, Register is a candidate.
Low Coupling

- Register could then send an addPayment message to Sale, passing along the new Payment as a parameter.
- The assignment of responsibilities couples the Register class to knowledge of the Payment class.

Low Coupling

- An alternative solution is to create Payment and associate it with the Sale.
- No coupling between Register and Payment.
Low Coupling

• Some of the places where coupling occurs:
  – Attributes: X has an attribute that refers to a Y instance.
  – Methods: e.g. a parameter or a local variable of type Y is found in a method of X.
  – Subclasses: X is a subclass of Y.
  – Types: X implements interface Y.

• There is no specific measurement for coupling, but in general, classes that are generic and simple to reuse have low coupling.

• There will always be some coupling among objects, otherwise, there would be no collaboration.

High Cohesion

• Cohesion: it is a measure of how strongly related and focused the responsibilities of an element are.

• A class with low cohesion does many unrelated activities or does too much work.

• Problems because of a design with low cohesion:
  – Hard to understand.
  – Hard to reuse.
  – Hard to maintain.
  – Delicate, affected by change.

• Problem: How to keep complexity manageable?

• Solution: Assign a “responsibility” so that cohesion remains high.
High Cohesion

- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
  - By Creator, Register is a candidate.
  - Register may become bloated if it is assigned more and more system operations.

```
: Register

makePayment()

: Sale

create()

addPayment(p)
```

High Cohesion

- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
- This design supports high cohesion and low coupling.

```
: Register

makePayment()

: Sale

create()

: Payment
```
High Cohesion

• Scenarios that illustrate varying degrees of functional cohesion

1. Very low cohesion: class responsible for many things in many different areas.
   – e.g.: a class responsible for interfacing with a database and remote procedure calls.

2. Low cohesion: class responsible for complex task in a functional area.
   – e.g.: a class responsible for interacting with a relational database.

3. High cohesion: class has moderate responsibility in one functional area and it collaborates with other classes to fulfill a task.
   – e.g.: a class responsible for one section of interfacing with a database.

• Rule of thumb: a class with high cohesion has a relative low number of methods, with highly related functionality, and doesn’t do much work. It collaborates and delegates.
Part I

Requirements - An Introduction

New Features and Functions Start With Requirements Management
Getting Past the Obvious

Why is Developing Software Hard?

“The hardest single part of building a software system is deciding what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.”


*Computer* (April, 1987)
Obligatory SE Horror Statistics

From the *Standish Group Report* (1998):

- 26% of IT projects are successful
- 46% are challenged
- 28% fail
- average cost overrun is 89%
- average schedule overrun is 122%
- 45% of the functions provided in newly developed systems are *never used*.

Some Typical Problems

- **Specifications**
  - requirements not always identified
  - requirements not always verifiable
  - Allocation to components incomplete
  - Requirements traceability informal
- **Poor integration of disciplines**
- **Lessons are not learned from the past**
  - The systems engineers are permanently reinventing the wheel
Sources of Failure

Five of the top reasons:

- Incomplete requirements
- Lack of user involvement
- Unrealistic customer expectations
- Changing requirements and specifications
- Capabilities provided no longer needed.

Requirement Pitfalls

- Assume requirements are stated by the user.
- Customers are confused with users, and visa-versa.
- Dialog (communication) between users and developers is weak.
- Process lacking for managing changes to requirements.
- Developers allowed to fill in the gaps.
- Rush to get through requirements – “Let’s get to the good stuff...”
Why can’t we get requirements right?

• Mistaking “stated requirements” with the “real requirements.”
• Ambiguous requirements which cannot be validated in the final product.
• Inability to transfer domain expertise.
• Getting requirements from the wrong source(s).

Why can’t we get requirements right?

• The user doesn’t always know what they want!
• Not change, but the inability to effectively manage change.
• Commitment from all parties to be responsible for the success of a project.
• Politics
• Summary : It involves people.
Greatest Obstacle to Successful Business Solution Delivery: the Business—IT Gap

Sobering Statistics:
- 7 out of 10 projects fail
- Of those that fail, 7 out of 10 failures are due to problems in the early stages of deciding on the business requirements, project scope and planning.

How to ‘Bridge the Gap’?

Successful Solutions depend upon a precise translation from Customer Needs to Business Requirements to Software Engineering Capabilities

What causes 72% of all major IT projects to fail?

The industry has found three major obstacles:

1. Translation of Business Strategy and Vision into Solution Requirements
2. Coping with Product Choices, Marketing Hype and ever changing Technologies
3. Increasing Complexity of Enterprise Environments
Overcoming communication breakdowns that occur during Solution Design

Voice of the Customer
- Why doesn't it do?
- How do they expect us to use this?
- Look what is possible!

Voice of the Business
- We need a solution to a business problem. How can technology give us a competitive advantage?
- Where's our requirements? We're designing in a vacuum!

Voice of the Technologist

... What happens without translation?

- Premature commitments to solutions are often made
- Business objectives are not communicated well
- Vendors make unsupportable claims and projections regarding their capabilities
- Solution planning takes too long - people get stuck
- Non-optimal solution decisions are made
- Often developers are left to decide for themselves
- Business results are not realized
Envisioning has always been a necessary and difficult job

“Customers never know what they want…

… until they see what they get”

Part II

Requirements Engineering
Requirements Gathering
(e.g., Borland CaliberRM Definition Tool)

Business Analyst

- Requirement Engineering
- Requirements management
  - Versioning
  - Tracing
  - Impact analysis
- Automatic Notification

After Requirements, You Design
What is Requirements Engineering About?

Effectively generating High Quality Requirements:
- correct
- consistent
- complete
- modifiable
- traceable
- verifiable
- non-ambiguous
- understandable
- annotated

Requirements Engineering

Requirements Development
- Elicitation
- Analysis
- Modeling & Specification
- Verification & Validation

Requirements Management
- Change Control
- Version Control
- Tracing & Impact Analysis
- Status Tracking

Best Practices, Methods, Tools and Processes for Requirements Engineering

Definitions of Requirements Engineering

- “A systematic process of developing requirements through an iterative, cooperative process of analyzing the problem, documenting the resulting observations in a variety of representation formats and checking the accuracy of the understanding gained.” [Loucopoulos, 95]
- “Requirements engineering is the branch of software engineering that is concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families.” [Zave, 97]
Use Cases and Requirements

- The use cases are not all of the requirements – because some things don’t fit into scenarios
  - Business Rules (structural facts, computation rules, action restricting rules, action triggering rules)
  - Non-functional Requirements: security, reliability, performance, usability, flexibility (all of the “-ilities”)

- Links to use cases
  - The Business Rules and the Non-functional Requirements are written in text, and they have links to the use cases where they are pertinent
  - Some folks store these items in a spreadsheet or a repository tool (such as Telelogic DOORS or Rational RequisitePro)

Traditional RE

- 44% 8% of errors are due to incorrect requirements
- Cost of fixing an error increases exponentially with time
- Phase during which the most critical decisions are made
- Daniel Berry: “Devoting 25% of the system development budget on requirements reduces the cost overrun from ~ 80% to ~ 5%, “
Project Success Factors (Chaos report)

- User Involvement 15.9%
- Executive Management Support 13.9%
- Clear Statement of Requirements 13.0%
- Proper Planning 9.6%
- Realistic Expectations 8.2%
- Smaller Project Milestones 7.7%
- Competent Staff 7.2%
- Ownership 5.3%
- Clear Vision & Objectives 2.9%
- Hard-Working, Focused Staff 2.4%
- Other 13.9%

Project Challenged Factors (Chaos report)

- Lack of User Input 12.8%
- Incomplete Requirements & Specifications 12.3%
- Lack of Executive Support 7.5%
- Changing Requirements & Specifications 11.8%
- Technology Incompetence 7.0%
- Lack of Resources 6.4%
- Unrealistic Expectations 5.9%
- Unclear Objectives 5.3%
- Unrealistic Time Frames 4.3%
- New Technology 3.7%
- Other 23.0%
Project Impaired Factors (Chaos report)

- Incomplete Requirements: 13.1%
- Lack of User Involvement: 12.4%
- Lack of Resources: 10.6%
- Unrealistic Expectations: 9.9%
- Lack of Executive Support: 9.3%
- Changing Requirements & Specifications: 8.7%
- Lack of Planning: 8.1%
- Didn't Need It Any Longer: 7.5%
- Lack of IT Management: 6.2%
- Technology Illiteracy: 4.3%
- Other: 9.9%

Types of Requirements

- Requirements
  - Functional requirements
  - Non-functional requirements
    - Product requirements
    - Process requirements
    - External requirements
      - Efficiency requirements
      - Reliability requirements
      - Portability requirements
      - Interoperability requirements
      - Ethical requirements
      - Usability requirements
      - Delivery requirements
      - Implementation requirements
      - Standards requirements
      - Legislative requirements
      - Performance requirements
      - Space requirements
      - Privacy requirements
      - Safety requirements
Requirements Engineering

Set of Activities (Process):

• Requirements elicitation and discovery
• Modeling/Analysis
• Specification
• Verification & Validation

Discovering Requirements

“The plan is nothing; the planning is everything”
• Dwight Eisenhower
• Allied supreme commander during World War II
• 34th President of United States (1953–61)

“The discovery is nothing: the discovering (the exploring) is everything.”
Sources of Requirements

Domain Expertise

- Domains are subjects that you will need to know something about if you are to build a product.
- Subject domain expertise helps you understand how and why your customers use the kinds of systems that you're developing.
  – ("Work with a User to Think Like a User.")
- Warning – Domain knowledge acquisition is a nonlinear process!
Domain Expertise

Ignoring implementation details, what’s your confidence level for developing a system to:

– Play solitaire
– Play poker
– Play bridge

How about:

– Paging device
– Pulse monitor
– Pacemaker

Eliciting Requirements

• *Conscious Requirements* : uppermost in the user’s mind – symptomatic of something the user wants to improve.

• *Unconscious Requirements* : not mentioned since the user assumes everyone else has the same knowledge.

• *Undreamed of Requirements* : things the user will ask for once they realize what is possible
Quality Requirements

• **Correct** – only user representative can determine
• **Feasible** – get reality check on what can or cannot be done technically or within given cost constraints.
• **Necessary** – trace each requirement back to its origin
• **Unambiguous** – one interpretation
• **Verifiable** – how to you know if the requirement was implemented properly?
• **Prioritized** – function of value provided to the customer

(from “Writing Quality Requirements” – Karl Wiegers)

Writing Example #1

“The product shall provide status messages at regular intervals not less than every 60 seconds.”
Writing Example #1

“The product shall provide status messages at regular intervals not less than every 60 seconds.”

- Incomplete – What are the status messages and how are they supposed to be displayed?
- Ambiguous – What part of the product? Regular interval?
- Not verifiable

Alternative #1

   1.1. The Background Task Manager shall display status messages in a designated area of the user interface at intervals of 60 plus or minus 10 seconds.
   1.2. If background task processing is progressing normally, the percentage of the background task processing that has been completed shall be displayed.
   1.3. A message shall be displayed when the background task is completed.
   1.4. An error message shall be displayed if the background task has stalled.
Writing Example #2

“The product shall switch between displaying and hiding non-printing characters instantaneously.”

• Not Feasible – computers cannot do anything instantaneously.
• Incomplete – conditions which trigger state switch
• Ambiguous – “non-printing character”
Alternative #2

“The user shall be able to toggle between displaying and hiding all HTML markup tags in the document being edited with the activation of a specific triggering condition.”

• Note that “triggering condition” is left for design

The Specification Trap

The Landing Pilot is the Non-Landing Pilot until the ‘decision altitude’ call, when the Handling Non-Landing Pilot hands the handling to the Non-Handling Landing Pilot, unless the latter calls ‘go-around,’ in which case the Handling Non-Landing Pilot continues handling and the Non-Handling Landing Pilot continues non-handling until the next call of ‘land,’ or ‘go-around’ as appropriate. In view of recent confusions over these rules, it was deemed necessary to restate them clearly.

• British Airways memorandum, quoted in Pilot Magazine, December 1996
The Specification Trap

“Some Things Are Better Done than Described”
• Pragmatic Programmer Tip #57

“In the Development of Every Product There Comes a Time When it is Necessary to Shoot the Engineers and Start Production.”
• Anonymous (Taylor Instrument - 1982)

Part III

Requirements Development Methodology
The Planning Spectrum

PROJECT NEGOTIATIONS

- Managing project definition at the beginning of the project
- Using project definition to manage requirements creep
- Estimating techniques
- Tools for assisting estimation process
- Tradeoffs between schedule, budget, staff, quality
- Tools for rational negotiation
- What to do when rational communications are impossible
Managing Project Definition:
What does “success” mean?

- Many projects succeed or fail at the very beginning, before any technical work is done.
- Fundamental requirement: identifying who has the right to declare “success” — owner, shareholder, etc, etc.
- Fundamental elements of “success”
  - finishing on time
  - staying within budget
  - delivering the required functionality
  - providing “good enough” level of quality
  - getting the next round of VC funding, or launching the IPO
- The combination of these constraints may prove impossible to achieve — so the pragmatic aspect of success often depends on agreement as to which areas can be compromised or satisfied.
- Biggest risk: lack of realistic triage at beginning of project

Using Project Definition to Manage Requirements Creep

- Typical behavior in projects: new requirements are added at the rate of 1% per month
- Requirements “creep” and requirements “churn” are a major element of project management risk.
- But if you don’t have a formal document describing the requirements, it’s hard to identify creep or churn.
- Assuming that you do have such a document, you need to use it to negotiate schedule/budget/staff modifications if the requirements change or increase.
- Biggest risk of all: an ambiguous spec is usually a sign of unresolved conflict between diverse political camps in the user community. Related risk: techies assume that it’s their fault they can’t understand ambiguous spec
Estimating Techniques

- Fundamental truth: it’s almost impossible to estimate a project if you don’t have metrics from previous projects.
- Consequence: most of what’s described as “estimating” is either “guessing” or “negotiating”
- Political reality: estimates are produced by people who have little prior estimating experience, and who have a vested interest in believing their optimistic predictions
- A radical suggestion: create a separate estimating group whose work is judged and rewarded by the accuracy of its estimates, not the political acceptability of estimates
- Main technical suggestion: break the project down into small, independent “inch pebbles” and get several estimates
- For complex projects, get a commercial estimating tool

RUP Inception Phase

- Most projects require a short initial step in which the following kinds of questions are explored:
  - What is the vision and business case for this project?
  - Feasible?
  - Buy and/or build?
  - Rough estimate of cost: Is it $10K or $100K or in the millions?
  - Should we proceed or stop?
Sample Inception Artifacts

- These artifacts are only partially completed in this phase.
- They will be iteratively refined in subsequent iterations.

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vision and Business Case</td>
<td>Describes the high-level goals and constraints, the business case, and provides an executive summary.</td>
</tr>
<tr>
<td>Use-Case Model</td>
<td>Describes the functional requirements, and related non-functional requirements.</td>
</tr>
<tr>
<td>Supplementary Specification</td>
<td>Describes other requirements.</td>
</tr>
<tr>
<td>Glossary</td>
<td>Key domain terminology.</td>
</tr>
<tr>
<td>Risk List &amp; Risk Management Plan</td>
<td>Describes the business, technical, resource, schedule risks, and ideas for their mitigation or response.</td>
</tr>
<tr>
<td>Prototypes and proof-of-concepts</td>
<td>To clarify the vision, and validate technical ideas.</td>
</tr>
<tr>
<td>Iteration Plan</td>
<td>Describes what to do in the first elaboration iteration.</td>
</tr>
<tr>
<td>Phase Plan &amp; Software Development Plan</td>
<td>Low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources.</td>
</tr>
<tr>
<td>Development Case</td>
<td>A description of the customized UP steps and artifacts for this project. In the UP, one always customizes it for the project.</td>
</tr>
</tbody>
</table>

Requirements Categories in UP

- **Requirements** are capabilities and conditions to which the system—and more broadly, the project—must conform.
- The UP promotes a set of best practices, one of which is *manage requirements*.
  - In the context of inevitably changing and unclear stakeholder's wishes—"a systematic approach to finding, documenting, organizing, and tracking the changing requirements of a system"
Types of Requirements

• In the UP, requirements are categorized according to the FURPS+ model:
  – **Functional**—features, capabilities, security.
  – **Usability**—human factors, help, documentation.
  – **Reliability**—frequency of failure, recoverability, predictability.
  – **Performance**—response times, throughput, accuracy, availability, resource usage.
  – **Supportability**—adaptability, maintainability, internationalization, configurability.

Types of Requirements

• The "+" in FURPS+ indicates ancillary and sub-factors, such as:
  – **Implementation**—resource limitations, languages and tools, hardware, ...
  – **Interface**—constraints imposed by interfacing with external systems.
  – **Operations**—system management in its operational setting.
  – **Packaging**
  – **Legal**—licensing and so forth.
Case Study: Use Cases in a Typical Inception Phase

- Not all use cases are written in their fully dressed form during inception.
- The Use-Case Model at this phase of the case study could be detailed as follows:

<table>
<thead>
<tr>
<th>Fully Dressed</th>
<th>Casual</th>
<th>Brief</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Sale</td>
<td>Process Rental</td>
<td>Cash In</td>
</tr>
<tr>
<td>Handle Returns</td>
<td>Analyze Sales Activity</td>
<td>Cash Out</td>
</tr>
<tr>
<td></td>
<td>Manage Security</td>
<td>Manage Users</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Start Up</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shut Down</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Manage System Tables</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Other Requirements

- **Supplementary Specification**
  - Captures other kinds of requirements, such as documentation, packaging, supportability, licensing, and so forth.

- **Glossary**
  - Captures terms and definitions; it can also play the role of a data dictionary.

- **Vision**
  - Defines the stakeholders' view of the product to be developed, specified in terms of the stakeholders' key needs and features. Containing an outline of the envisioned core requirements, it provides the contractual basis for the more detailed technical requirements.
More recent approaches: XP/ASD

- Extreme Programming
- AgileSoftware Development
- ASD considers software development as a co-operative game of invention and communication involving developers and customers where work products of the team should be measured for “sufficiency with respect to communicating with the target group”. [Alistair Cockburn, Agile Software Development, 2002]

Extreme Programming (XP)

- One of a group of “agile methods”.
- Created to handle problem domains whose requirements change:
  - Customers may not have a firm idea of what the required system should do. Indeed the system's functionality may be expected to change every few months.
  - In many software environments, dynamically changing requirements is the only constant.
Extreme Programming (XP)

• XP emphasises team work.
• The method encourages four values:
  – communication;
  – simplicity;
  – feedback; and
  – courage.
• Note: I’m using XP as an example of agile methods. Some of XP’s effects on project management are shared by DSDM, SCRUM etc.

The XP Lifecycle

• Extreme Analysis - based on development of “user stories”, that is lightweight Use Cases. Stories are broken down into tasks using Planning Games.
• Extreme Testing - write unit tests before classes (compare Microsoft approach).
• Extreme Code - write code to pass tests cases - the simplest thing that will work. Pair programming - continuous code review and integration, e.g. hourly builds.
• Extreme Design - Refactor code, remove duplicate code, generally clean-up the code. If more design is required use CRC (Class-Responsibilities-Collaborators) cards, and group design sessions for major problems.
• Extreme Scheduling - customers write stories while developers estimate costs. (from Susan Eisenbach’s BCS talk)
XP Practices

- **The Planning Game** – determine the scope of the next release, based on business priorities and technical estimates.
- **Small Releases** – every release should be as small as possible, containing the most valuable business requirements.
- **Metaphor** – guide all development with a simple, shared story of how the whole system works.
- **Simple Design** – design for the current functionality, not future needs, and make the design as simple as possible for that functionality (YAGNI).
- **Refactoring** – ongoing redesign of software to improve its responsiveness to change.
- **Testing** – developers write the unit tests before the code; unit tests are run “continuously”; customers write the functional tests.

XP Practices (2)

- **Pair Programming** – an extreme form of inspections; two people sit in front of the same terminal to write test cases and code.
- **Collective Ownership** – anyone can change any code anywhere in the system at any time.
- **Continuous Integration** – integrate and build the system many times a day, every time a task is finished.
- **40-hour Week** – work no more than 40 hours as a rule; never work overtime two weeks in a row.
- **On-site Customer** – real, live user on the team full time.
- **Coding Standards** – must be adhered to rigorously because all communication is through the code.
On-Site Customer

- Customer available on site to clarify stories and to make critical business decisions.
- Developers don’t make assumptions
- Developers don’t have to wait for decisions
- Face to face communication minimizes the chances of misunderstanding

XP in seven sentences

1. Based on small, very interacting teams of people working in pairs
2. Testing is practiced since the very beginning
3. System integration is performed daily
4. Use cases driven, with specific techniques to estimate time and cost of the project
5. Programs are continuously refactored
6. Written documentation besides code is kept to minimum
7. Write the simplest system that can work!
**XP challenges assumptions (1)**

- XP says that analogies between software engineering and other engineering are false:
  - software customers’ requirements change more frequently;
  - our products can be changed more easily;
  - the ratio of design cost: build cost is much higher;
  - if we consider coding as “design” and compile as “build”:
    - the “build” task is so quick and cheap it should be considered instant and free,
    - almost all software development is “design”.

**XP challenges assumptions (2)**

- The design meets *known existing requirements*, not all possible future functionality.
- Beck (1999): “If you believe that the future is uncertain, and you believe that you can cheaply change your mind, then putting in functionality on speculation is crazy. Put in what you need when you need it.”
How XP works (1)

- As with RAD and DSDM etc. programmers meet and communicate with customers regularly, and the software gets released incrementally.
- Programmers always work in pairs (considered more productive).

How XP works (2)

- Testing is the start point, not the end:
  - For each user *story*, the customer first writes an acceptance test.
  - For each unit the programmer writes a set of unit tests.
  - Then each unit in a story is coded.
  - When a unit is ready, its tests are run automatically.
- Customers are *allowed* to suggest improvements.
- Redesigns are common - what they call *refactoring* - and handled easily.
What effect does XP have on project management? (1)

• Project scope
  – XP embraces changes in requirements. There is little value in the traditional project scope that forms a contract for what functionality will be provided for a fixed cost/effort.
  – Instead, the customer writes on cards a set of user stories that describe what they want the new application to do for them.

What effect does XP have on project management? (2)

• Task and project estimating
  – The time (duration) to complete each user story is written on the relevant card. If the story will take more than 3 weeks, it must be split into two or more smaller stories.
  – These stories are prioritised in a release plan; the plan sets out which stories will be produced in each iteration. XP projects may have fixed time (scope varies) or fixed scope (time varies), but not both.
What effect does XP have on project management? (3)

• Quality management
  – For each user story the customer must write an acceptance test. These are produced before any software is written to satisfy the story. Acceptance tests are normally automated so they can be used as regression tests each time the software is changed.
  – Programmers always work in pairs to ensure learning, adherence to standards, and avoidance of errors.

What effect does XP have on project management? (4)

• Progress monitoring
  – Progress monitoring can be as simple as keeping a track of which story cards have been marked as completed (passed its acceptance test).
  – The XP concept project velocity (the speed at which the developers are producing work) prevents unwarranted optimism, e.g.:

  X    • “The first task took too long, so I expect the other tasks will be done more quickly than expected.”
What effect does XP have on project management? (5)

• Change control
  • XP is intended for environments where requirements are expected to change several times before the project is over.
  • As new user stories are programmed, changes to existing work may be needed.
    – XP developers put a lot of effort into improving code and keeping it as simple as possible (refactoring).
    – Constant refactoring necessitates regression testing of all affected modules. This reinforces the need for automated testing (see Quality Management above).

What effect does XP have on project management? (6)

• Risk management
  – Traditional software development focuses on reducing risk from software errors (verification) by having copious plans and procedures.
  – XP focuses on reducing risk by ensuring customers' requirements are met (validation).
  – Barry Boehm (2002) suggests that the best way to decide whether XP is suitable for a given project is to assess risk exposure.
Overall XP process

Planning

– User stories are written for scheduling, requirements, and functional tests
– The Planning Game creates the schedule
– The Project Velocity must be estimated
– Iterations of 2 - 4 weeks
– An Iteration Planning Meeting starts each iteration
– A Stand-up Meeting at the beginning of each day
– Fix XP when it breaks
The Planning Game

• Extreme Programming delivers, and steers by Business Value

  Business Value belongs to the customer!

• Four variables:
  – Scope
  – Time
  – Resources (very non-linear)
  – Quality (fixed at maximum)

The Contract between Customer and Developers

– The customer can set 3 of the 4 variables (actually, quality and resources are already fixed)
– Developers set the 4th variable
– Usually, the customer defines stories (scope) and sets the time
– Developers say how many stories they can do in that time
– The customer chooses the stories to develop

The customer cannot set all 4 variables!!
Customer Bill of Rights

1. You have the right to an overall plan, to know what can be accomplished, when at what cost
2. You have the right to get the most value out of every programming week
3. You have the right to see progress in a running system proven to work by passing repeatable tests that you specify
4. You have the right to change your mind, to substitute functionality, and to change priorities without paying exorbitant costs
5. You have the right to be informed of schedule changes, in time to choose how to reduce scope to restore the original date. You can cancel at any time and be left with a useful working system reflecting investments to date

Developer Bill of Rights

1. You have the right to know what is needed, with clear declarations of priority
2. You have the right to produce quality work at all times
3. You have the right to ask for and receive help from peers, superiors, and customers
4. You have the right to make and update your own estimates
5. You have the right to accept your responsibilities instead of having them assigned to you
Agile RE

- We cannot determine in advance what the final product looks like.
- The requirements evolve over time.
- Minimum of documentation
- Minimum of RE process
- Traditional RE will produce documents that are out of date while they are still being developed.

SW developer vs. user

“… Software developers by themselves aren’t very good at designing software that users like. But software developers can be very good at helping users discover what they like, because users by themselves aren’t very good at designing software they like either.”

(Steve McConnell)
Revisited: Definition of RE

- Requirements Engineering is concerned about finding out what the system is all about.
- Depending on the project, it also includes documentation and traceability.

Agile Requirements

- Agile Methodologies (XP, Scrum, FDD,..) are designed to address many of the requirement challenges we have discussed:
  - Access to domain expertise
  - Short, frequent release cycles - reviews
  - Quick prototype solutions (spikes)
  - Test driven design – encourages requirement discovery.
  - Embrace change
Agile Requirement Tips

1. Expect to gather requirements throughout the entire project - *Embrace Change*.
2. Explain the techniques to all stakeholders
3. Get buy-in from your management and customer.
4. Use the terminology of your users, avoid geek-speak – “Keep the Customer Satisfied”
5. Take a breadth-first approach.
6. Existing documents are a good source of requirements – beware of N.I.H. syndrome
7. Build models and prototypes
8. Get into your application domain!

AM enhances other software processes.

<table>
<thead>
<tr>
<th>Agile Modeling (AM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Software Process (XP, UP, DSDM, ...)</td>
</tr>
<tr>
<td>Your Process</td>
</tr>
</tbody>
</table>
Agile Model-Driven Development Lifecycle

Steps of Test-Driven Development (TDD)
Comparisons of TDD and AMDD

• TDD shortens the programming feedback loop whereas AMDD shortens the modeling feedback loop.

• TDD provides detailed specification (tests) whereas AMDD can provide traditional specifications (data models).

• TDD “speaks” to programmers whereas AMDD speaks to data professionals

Part IV

Requirements Gathering Guidelines
"If you don't know how well you are doing, then you know you are not doing very well"

This is not a discussion on how to create software requirements, but how to know you have created good requirements.

The goal of this section is to give you some tips to help you be more successful.

The problem with doing Requirements is that it is difficult to measure their quality, so you can't get feedback.

As a result:
You don't know how well you are doing
You don't know when to stop
Steps to Create Requirements

1. You must be motivated, i.e. want to do it.
2. You must know how to do it.
3. You must create CORRECT requirements.
4. You must create COMPATIBLE requirements.

We will discuss 3 and 4.

1: Correctness

Definition: Software Requirements tell what the system is to do, not how to do it. ‘Correct' is too broad a term to be useful.

Instead we will look at:
Agreement with the system requirements
Complete
Clear
Feasible
Agree with System Requirements

There is really no such thing as software requirements. There are only system requirements, some of which are implemented with software in a computer.

TIP: Is there a non-trivial system, above the software? If so, Review the Software specification with the System Engineers.

What if No System?

TIP: If there is no larger system, create “Use Cases”, or CONOPS to capture functionality.
TIP: Make sure you know who all the stakeholders are. There are more than you think there are.
TIP: Remember that there are Requirements that the customer doesn’t tell you.
Complete

Completeness means that nothing is missing.

TIP: Use a Requirements Development tool or the suggested format in IEEE 830, to remind you of what should be in the specification.

TIP: Wrong information is better than no information because it motivates people to correct it. So be complete by entering wrong information.

Clear

Not ambiguous and Inter-consistent

TIP  You are the worst person to determine if your specification is ambiguous. Get some one else to read and explain it.

TIP  Never put the same information in two places. Instead use references, or better yet, hypertext format.
Feasible

Vendors love a non-feasible project because they can make more money working on it.

TIP: Create a working model or simulation, if only on paper.
TIP: It never hurts to call the first version, a Feasibility model. There is no upside risk and less downside risk.

2. Compatibility

Definition: Software Requirements must be compatible, that is interrelate within the other parts of the software development project.

Compatibility implies:
• Verifiable
• Traceable
• Modifiable
• Ranked for importance
Verifiable

A Requirement that cannot be tested, is not a Requirement.

TIP: Take a tip from the Agile/XP/Scrum people: create requirements that are in fact the tests that an acceptable system should be able to pass.

---

Traceable

Traceability is required for Change Management

TIP: Use a software tool or at least number each paragraph in the requirements specification. Then carry that number forward through the remainder of the development. Never change these numbers.
**Modifiable**

“Software projects change as rapidly as any project ever conceived by the human mind.”

“Requirements change 2% per month”

T. Capers Jones

TIP: Remember software has dis-economy of scale. Design and build the system in parts (GUI, DBM, etc.)

---

**Continued**

TIP: Because the requirements will change, design and build the system, a portion at a time: Incremental or Spiral development.

TIP: Guarantee there is some success within every budget year.

TIP: This is a big help when requirements change; It is always easier to say ‘later’ than to say ‘no’.
Ranked for Importance

TIP: Decide what is the most important aspect of the system:
• Get to market fast
• Ease of use
• Run in real time
• Be error-free to .999?

Continued

TIP: Attach an importance number to every requirement, if only 1...5, or H-M-L. Build just the most important subset of requirements first.
TIP: Remember that risky requirements are, by definition, important.
Ranked for stability

TIP: Attach a stability number to every requirement, if only 1...5, or H-M-L. Isolate the volatile parts in the system and don't spend too much time polishing them.

Other Things to Remember

TIP: Do not put too much into the Requirements specification, i.e. Project information or Design information.
TIP: Remember that the Requirements specification will be used to size the project.
TIP: Alternatives that are almost equally good, will generate the most argument.
Summary

If you do not know that you have good requirements, you will not have good requirements.

Make sure they are correct by following the steps here.
Make sure they are compatible by following the steps here.

Resources


Standards Resources

IEEE/EIA 12207.0 “Industry Implementation of International Standard ISO/IEC 12207”
IEEE Std 1220 “Application and Management of the Systems Engineering Process”
IEEE Std 830 “Recommended Practice for Software Requirements Specifications”
IEEE Std 1362 “Concept of Operations Document”
IEEE Std 1540 “Risk Management”
IEEE Std 1012 “Software Verification and Validation”
IEEE Std 730 “Software Quality Assurance Plans”
IEEE Std 828 “Software Configuration Management Plans”

Part V

Going Forward
There is an Urgent Need for More Reliable, Concept-driven, Early Requirements Processes

- SDLC Process model frameworks have matured and converged significantly on effective practices in system development phases, however, they are still weakest on the front-end.

```
RUP Phases: Inception Elaboration Construction Transition
- ???
- ...

MSF Phases: Envisioning Planning Developing Stabilizing

|| Less mature methods / support | More mature methods / support |
|--------------------------------|

Do Agile Methods and Approaches Fill the Gap?

- Agile approaches recognize that continuous, effective communication among all stakeholders is absolutely central
  - they stress developing an attitude of mutual collaboration and invention
  - represent a significant step forward in the maturity and “realism” of software development methods

- But, Agile approaches also don’t offer mature, reliable techniques or assets with which to ensure that this type communication and joint decision making is fostered.

- Bottom line: NO
  - approaches of all kinds recognize the need to address the “envisioning” phase
  - there is definitely room for improvement for reliably facilitating envisioning
How is the ‘intended’ system identified in the first place

Addressing the creative or inventive side of solution development requires exploration of:

• ‘What is it possible to be?’
• “What technology enablers should we assess and adopt” (in other words the ‘system how’)
• “How to encourage and support the discovery processes that occur when ‘solutions are in search of problems’?
• “How to deal with emerging technologies?” (how to realize their business value and gain competitive advantage)

“Software development is a game of invention and communication.”
Alistair Cockburn

Importance of Modeling the Problem Situation

• Why model problems?
  – Understanding of cause and effect
  – Surfacing assumptions
  – Determining scope
  – Understanding systemic nature of a world

“A problem well stated is a problem half solved”
Companies need to align IT projects to their Business Strategy

Key stakeholders include:
• End-users who will have to live with the solution
• The Enterprise that is hoping to realize a business ROI
• IT departments or other technical organizations responsible for implementing the solution
• Technology vendors whose technology provides underpinnings and components of the solution

Stakeholders often have conflicting objectives.

Emerging Technologies and Component Solutions are Revolutionizing ‘Business Capabilities’

• System development is no longer “green-field” development, but is now often more about:
  – configuring and integrating components,
  – innovatively leverage pre-built solution capabilities
  – creatively using new technologies.

• Methods are needed to help companies move from:
  – strategy to capability design
  – technology assessment to selection
  – Technology selection to solution roadmaps

• Tools are important
  – for decision making with consensus building, solution envisioning & design, strategic business case to project planning and project execution
Seeking a repeatable means to mediate strategic communication and requirements

Candidate Approaches (illustrative, partial list):
• Business Service Patterns
• Business Centric Methodology
• Business Capabilities Design
• Capability Cases (note, members of the organizing committee will present an approach based on these)
• Agile Enterprise Architecture Models
• …

Part VI

Distributed Requirements Engineering
Distributed RE

- More and more distributed teams
- Cost savings
- Time savings
- Can “get in” experts quickly
- HW cannot always travel
- Possible with new communication media:
  - Video conferencing
  - Groupware tools

Advantages of Distributed Development: Creativity & Quality

- Ocker, Hiltz, Turoff, Fjermestad:
  - 41 groups of 5 participants each
  - Face-2-face vs. distributed, asynchronous
  - Distributed teams produced:
    - marginally more creative designs
    - higher quality designs
Advantages of Distributed Development: Quality

• Damian:
  – 15 groups of 4 people
  – Face-2-face vs. 4 distributed settings
  – Used facilitation
  – One of the distributed settings produced the best results

Advantages of Distributed Development: Productivity

• Aoyama:
  – A distributed concurrent development process replaced a sequential centralized process for a large-scale telecoms software project.
  – Drastic reduction of the development cycle-time
Key Challenges in Distributed RE

- Communication
- Coordination
- Time shift
- Cultural issues
- Misunderstandings
- Knowing people
- Common repository

Example

An embedded system product at Lucent Technologies was developed in two locations: Germany and the UK. This project revealed many problems resulting from the distributed development process. Unit testing and development was plagued by incomplete specifications. Implementation diverged from the design document, and the design document was not updated. Bug reports were generated from outdated design documents that identified ill-functioning components. While the components were working as implemented, the tests generated at other sites used design documentation that was never updated after designs changed.
Spiral Model of the RE Process

Requirements Elicitation

“The most difficult part of requirements gathering is not the act of recording what the users want; it is the exploratory, developmental activity of helping users figure out what they want.”

(Steve McConnell)
The requirements elicitation process

Establish objectives

- Business goals
- Problem to be solved
- System constraints

Understand background

- Organisational structure
- Application domain
- Existing systems

Organise knowledge

- Stakeholder identification
- Goal prioritisation
- Domain knowledge filtering

Collect requirements

- Stakeholder requirements
- Domain requirements
- Organisational requirements

Interviews

- Interviewing is more than asking questions. It is a structured technique that helps the requirements engineer or analyst to discuss the system with different stakeholders and build up an understanding of their requirements.

- Types of interview:
  - Open/unstructured interviews: There is no predefined agenda and the requirements engineer discusses, in an open-ended way, what stakeholders want from the system.
  - Closed/structured interviews: The requirements engineer looks for answers to a pre-defined set of questions.
Preparing the questions

• Context-free questions:
  – Process
  – Product
  – Meta-questions

• Open- / closed-ended questions

Context-free process questions

• Relate to the process, rather than the product
• Examples:
  – Who is the client? Who is the user?
  – What is a highly successful solution really worth?
  – What is the real reason for wanting to solve this problem?
  – How much time do we have for this project?
  – What is the trade-off between time and value?
  – Where else can the solution to this problem be obtained?
Context-free product questions

• Relate to the product, rather than the process
• Examples:
  – What problems will the system solve?
  – What problems could this system create?
  – What environment is this system likely to encounter?
  – What kind of precision is required or desired in the product?

Meta-questions

• These are questions about the questions asked
• Examples:
  – Am I asking you too many questions?
  – Do my questions seem relevant?
  – Are you the right person to answer these questions? Are your answers official?
  – Is there anyone else who I should be talking to?
  – Is there anything else I should be asking you?
  – Is there anything you want to ask me?
  – Can I return or call you with more questions later?
Open-ended questions

• Encourages unconstrained answers that provide a large amount of information
• Can be used when you don’t have a good understanding of the problem or domain
• Examples:
  – What problems do you currently experience in your work?
  – What kind of things would you like to change in the way you work?
  – What type of reports would you like to see?

Closed-ended questions

• Used when you want a precise answer to a question
• Examples:
  – How often should the XYZ report be generated?
  – What fields do you think should be protected?
  – What kind of error should be generated if the user types in bad data?
Scenarios / Use Cases

- Scenarios are stories which explain how a system might be used. They should include:
  - a description of the system state before entering the scenario
  - normal flow of events in the scenario
  - alternative flows of events in the scenario
  - exceptions to the normal flow of events
  - information about concurrent activities
  - a description of the system state at the end of the scenario

- Scenarios are examples of interaction sessions which describe how a user or other system interacts with a system
- Discovering scenarios exposes possible system interactions and reveals system facilities which may be required

Types of prototyping

- Throw-away prototyping
  - prototype only serves the elicitation of the user reaction
  - creation of prototype must be rapid, otherwise too expensive

- Evolutionary prototyping
  - prototype altered to incorporate design changes
  - eventually becomes the final product

- Incremental prototyping
  - product built as separate components / modules
  - each component prototyped and tested, then added to the final system
Requirements analysis and negotiation

Requirements analysis

- Necessity checking
- Consistency and completeness checking
- Feasibility checking
- Unnecessary requirements
- Conflicting and incomplete requirements

Requirements discussion

Requirements prioritisation

Requirements agreement

Requirements negotiation

Measurable metrics

<table>
<thead>
<tr>
<th>Property</th>
<th>Metric</th>
</tr>
</thead>
</table>
| Performance | 1. Processed transactions per second  
2. Response time to user input |
| Reliability | 1. Rate of occurrence of failure  
2. Mean time to failure |
| Availability | Probability of failure on demand |
| Size | Kbytes |
| Usability | 1. Time taken to learn 80% of the facilities  
2. Number of errors made by users in a given time period |
| Robustness | Time to restart after system failure |
| Portability | Number of target systems |
Verification vs. validation

• Requirements verification works with raw requirements as elicited from the system stakeholders
  – “Have we got the requirements right?” is the key question to be answered at this stage
• Requirements validation works with a final draft of the requirements document
  – “Have we got the right requirements?” is the key question to be answered at this stage

What to look out for

• Incomplete lists, typically ending with "etc.", "and/or", and "TBD".
• Vague words and phrases, such as "generally", "normally", "to the greatest extent", and "where practicable".
• Imprecise verbs, such as "supported", "handled", "processed", or "rejected".
• Implied certainty, such as "always", "never", "all", or "every".
• Passive voice, such as "the counter is set." (by whom?)
• Every pronoun, particularly "it" or "its" should have an explicit and unmistakable reference.
What to look out for

• Comparatives/superlatives, such as "earliest", "latest", "highest". Words ending in "est" or "er" should be suspect.
• Words whose meanings are subject to different interpretations between the customer and contractor such as:
  – Instantaneous
  – Simultaneous
  – Achievable
  – Complete
  – Finish
  – Degraded
  – A minimum number of
  – Nominal/normal/average
  – Peak/minimum/steady state
  – As required/specified/indicated
  – Coincident/adjacent/synchronous with

What to look out for

• Non-quantifiable measures, such as
  – Flexible
  – Modular
  – Efficient
  – Adequate
  – Accomplish
  – Possible (possibly/correct(ly))
  – Minimum required/acceptable/reasonable
  – Better/higher/faster/less/slower/infrequent

Requirements Management

• The principal concerns of requirements management are:
  – Managing changes to agreed requirements
  – Managing the relationships between requirements
  – Managing the dependencies between the requirements document and other documents produced in the systems engineering process
• Requirements cannot be managed effectively without requirements traceability.
  – A requirement is traceable if you can discover who suggested the requirement, why the requirement exists, what requirements are related to it and how that requirement relates to other information such as systems designs, implementations and user documentation.

Types of traceability

• Requirements-sources traceability
  – Links the requirement and the people or documents which specified the requirement
• Requirements-rationale traceability
  – Links the requirement with a description of why that requirement has been specified.
• Requirements-requirements traceability
  – Links requirements with other requirements which are, in some way, dependent on them. This should be a two-way link (dependants and is dependent on).
Types of traceability

- **Requirements architecture traceability**
  - Links requirements with the sub-systems where these requirements are implemented. This is particularly important where sub-systems are being developed by different sub-contractors.

- **Requirements design traceability**
  - Links requirements with specific hardware or software components in the system which are used to implement the requirement.

- **Requirements interface traceability**
  - Links requirements with the interfaces of external systems which are used in the provision of the requirements.

A requirements management system
Part VII

Conclusion

Conclusions

• Distributed RE is possible!
• Use well-established RE practices
• Communicate!!
• Have ONE common repository
Questions?

“Everything should be made as simple as possible, but not simpler.”
- Albert Einstein

Course Assignments

- Individual Assignments
  - Reports based on case studies or exercises

- Project-Related Assignments
  - All assignments (other than the individual assessments) will correspond to milestones in the team project.
  - As the course progresses, students will be applying various methodologies to a project of their choice. The project and related software system should relate to a real-world scenario chosen by each team. The project will consist of inter-related deliverables which are due on a (bi-) weekly basis.
  - There will be only one submission per team per deliverable and all teams must demonstrate their projects to the course instructor.
  - A sample project description and additional details will be available under handouts on the course Web site.
Course Project

- **Project Logistics**
  - Teams will pick their own projects, within certain constraints: for instance, all projects should involve multiple distributed subsystems (e.g., web-based electronic services projects including client, application server, and database tiers). Students will need to come up to speed on whatever programming languages and/or software technologies they choose for their projects - which will not necessarily be covered in class.
  - Students will be required to form themselves into "pairs" of exactly two (2) members each; if there is an odd number of students in the class, then one (1) team of three (3) members will be permitted. There may not be any "pairs" of only one member! The instructor and TA(s) will then assist the pairs in forming "teams", ideally each consisting of two (2) "pairs", possibly three (3) pairs if necessary due to enrollment, but students are encouraged to form their own 2-pair teams in advance. If some students drop the course, any remaining pair or team members may be arbitrarily reassigned to other pairs/teams at the discretion of the instructor (but are strongly encouraged to reform pairs/teams on their own). Students will develop and test their project code together with the other member of their programming pair.

Readings

- **Readings**
  - [http://www.agilemodeling.com/essays/agileRequirements.htm](http://www.agilemodeling.com/essays/agileRequirements.htm)
  - Slides and Handouts posted on the course web site
  - Documentation provided with software engineering tools

- **Project Frameworks Setup (ongoing)**
  - As per references provided on the course Web site

- **Individual Assignment**
  - See Handouts: “Team Project Part #3”
Next Session:
Introduction to Software Analysis and Design

- Roles of Analysis and Design
- Traditional Data and Process Modeling Approaches
- Performing Requirements Analysis
- Object-Oriented Modeling
- Selecting and Combining Approaches
- Creating a Data Model
- Homework #5
- Project #4