Administrivia

- Wednesday:
  - Groups and one-sentence idea(s) due at class
  - One per group
  - If you have a small group, still submit so that you will be kept together. (better to find 6 people)

- We assign teams and you start on Monday

Requirements and Specification

CS169
Lecture 4

Requirements Engineering

- The hardest single part of building a software system is deciding what to build
  - Cripples the process if done wrong
  - Costly to rectify later

- Must be adopted to the software process
  - Elaborate requirements/specs for waterfall
  - Partial set of user stories for iterative processes

Determining Stakeholders and Needs

- Must determine stakeholders
  - anyone who benefits from the system developed
  - E.g., who’s client and who’s user?

- Try to understand what their needs are

- Reconcile different needs/points of view

Techniques

- Interviewing
- User stories
- Strawmen
- Prototypes

Interviewing

- One path is obvious
  - Sit down with client/user and ask questions
  - Listen to what they say, and don’t say

- A less obvious path
  - Master-apprentice relationship
  - Have them teach you what they do
  - Go to workplace and watch them do the task

- In all types of interviews, get details
  - Ask for copies of reports, logs, emails on process
  - These may support, fill in, or contradict what the user said
**Extreme Programming - User Stories**

- Recall: client writes user stories
  - Using client vocabulary
- Describe usage scenarios of software
  - Title, short description
- Each user story has acceptance tests
  - Clarify the story
  - Will tell you when the customer thinks story is done

**User Story Example**

- **Title**: Delete account
- **Actors**: trusted operator
- **Preconditions**: account must be empty
- **Trigger**: operator selects menu option
- **Scenario**: ..., confirmation dialog box, ...
- **Exceptions**: account not empty, unknown user
- **Priority**: important, but not for first release
- **Frequency of use**: rare

**Disadvantages of Talking**

- Interviews are useful, but
  "I know you believe you understood what you think I said, but I am not sure you realize that what you heard is not what I meant!"
- Users/clients may not
  - Have the vocabulary to tell you what they need
  - Know enough about computer science to understand what is possible
  - Or impossible
- Good idea to gather requirements in other ways, too

**Strawmen**

- Sketch the product for the user/client
  - Storyboards
  - Flowcharts
  - HTML mock-ups
  - Illustrate major events/interfaces/actions
- Anything to convey ideas without writing code!

**Rapid Prototyping**

- Write a prototype
  - Major functionality, superficially implemented
  - Falls down on moderate-to-extreme examples
    - No investment in scaling, error handling, etc.
- Show prototype to users/clients
  - Users have a real system - more reliable feedback
  - Refine requirements
  - But, significant investment
Pitfalls of Rapid Prototyping

- Needs to be done quickly
  - Remember, this is just the requirements phase!
  - Danger of spending too long refining prototype
- The prototype becomes the product
  - Prototype deliberately not thoroughly thought-out
  - Product will inherit the sub-optimal architecture
- Prototype serves as the spec
  - Prototype is incomplete, maybe even contradictory
- When done well, extremely useful

Summary of Requirements

- Find out what users/clients need
  - Not necessarily what they say they want
- Use
  - Interviews
  - User stories
  - Strawmen
  - Rapid prototyping
  - As appropriate . . .

Specifications

- Describe the functionality of the product
  - Precisely
  - Covering all circumstances
- Move from the finite to the infinite
  - Finite examples (requirements) to infinite set of possible computations
  - This is not easy

Views of Specifications

- Developer's
  - Specification must be detailed enough to implement
  - Unambiguous
  - Self-consistent
- Client's/user's
  - Specifications must be comprehensible
  - Usually means: not too technical
- Legal
  - Specification can be a contract
  - Should include acceptance criteria
    - If the software passes tests X, Y, and Z, it will be accepted

Informal Specifications

- Written in natural language
  - E.g., English
- Example
  “If sales for current month are below target sales, then report is to be printed, unless difference between target sales and actual sales is less than half of difference between target sales and actual sales in previous month, or if difference between target sales and actual sales for the current month is under 5%”

Problems with Informal Specs

- Informal specs of any size inevitably suffer from serious problems
  - Omissions
  - Something missing
  - Ambiguities
    - Something open to multiple interpretations
  - Contradictions
    - Spec says “do A” and “do not do A”

These problems will be faithfully implemented in the software unless found in the spec
Informal Specifications Revisited

“If sales for current month are below target sales, then report is to be printed, unless difference between target sales and actual sales is less than half of difference between target sales and actual sales in previous month, or if difference between target sales and actual sales for the current month is under 5%.”

Specification implies, but does not say, that there are monthly sales targets. Are there separate monthly targets, or is the monthly target e.g., 1/3 of the quarterly target?

5% of the target sales, or of the actual sales?

Comments on Informal Specification

• Informal specification is universally reviled
  - By academics
  - By “how to” authors
  - By respectable pundits

• Informal specification is also widely practiced
  - Why?

Why Do People Use Informal Specs?

• The common language is natural language
  - Customers can’t read formal specs
  - Neither can most programmers
  - Or most managers
  - A least-common denominator effect takes hold

• Truly formal specs are very time-consuming
  - And hard to understand
  - And overkill for most projects
Semi-Formal Specs

- Best current practice is "semi-formal" specs
  - Allows more precision than natural language where desired
- Usually a boxes-and-arrows notation
  - Must pay attention to:
    - What boxes mean
    - What arrows mean
    - Different in different systems!

Example 1: Dataflow Diagrams

- Old ideas
  - Several competing models from the '70s
  - Called "Structured Systems Analysis"
  - We present one
    - Others are similar
- 9 step procedure
  - With refinements of each step
- Example: automating a software store
  - Only show key steps

Example: Data Flow Diagrams

- Show logical data flow
  - "what happens, not how it happens"
Step 2: Decide What to Computerize

- Spec so far says nothing about how steps are done
  - could be a person looking up info on a card file
- Cost/benefit analysis could help decide this

Step 3: Refine Data Flows

- Further specify data items for each data flow
- Refine each flow stepwise
  - order identification
  - customer details
  - package details
- Refine further
  - this creates the “data dictionary”

Step 4: Refine Logic of Processes

- Have process give educational discount
  - explain what this means
  - 10% on up to 4 packages, 15% on 5 or more
- Translate into decision tree
  - makes it easy to see what is missing

```
give educational discount

Educational Institution
  <= 4 packages: 10%
  > 4 packages: 15%

Other: 0%
```

Step 6: Define Physical Resources

- For each file figure out
  - names
  - Organization
  - Indexed
  - Database tables
  - storage medium

Step 7: Determine Input/Output Specs

- Specify
  - UI
    - Screens/Windows/Buttons/Menus
    - Interactions
  - output files created
  - printed reports

Steps 8 & 9: Perform Sizing & Hardware Requirements

- Look at
  - volume of input (daily/hourly)
  - size / frequency of reports
  - size / number of records moving between CPU & storage
  - size of files
  - back-up requirements
  - input needs
  - output devices
  - is existing hardware adequate?
Entity-Relationship Diagrams

- Semi-formal technique
- Focuses on data rather than actions - In contrast to dataflow diagrams
- Really comes out of database world - Has crossed over into object-oriented analysis

1 to many relationships

many to many relationships

Finite State Machines

- Formal method
- State transitions - Set of states - Rules for moving between states - Designated start and final states

![Finite State Machine Diagram]

Advantages

- More precise than DFDs
- Easy to write down, validate, & convert into code - Some CASE tools directly generate code from FSMs
- Makes maintenance easy -> changes spec & regenerate

Disadvantages

- Lack of structure leads huge numbers of states - Fixed by Harel's Statecharts
- Does not deal with timing issues - Could use Petri nets

Z ("zed") Notation

- Formal specification language - Most successful one - Real skill required: set theory, functions & discrete math
- Z specifications consists of 4 sections - Given sets, data types, and constants - Sets that get defined in detail - State definition - Variable declarations & predicates that constrain values - Initial state - Operations

Example of Formal Specification

- Title: delete account named A by user U - Precondition: A 3 Accounts and U 3 Trusted and Balance(A) = 0 - Postcondition:
  
  if ConfirmDeleteDialogBox(A, U) then
  Accounts@after = Accounts - {A}
  else Accounts@after = Accounts

Notes: no need to consider the case when precondition is not true
Comparison of Specification Techniques

- **Formal methods**
  - powerful and precise
  - difficult to learn & use
  - could be useful where cost/safety/reliability is important
  - computers can assist in the spec./code reviews

- **Informal methods**
  - little power
  - easy to learn and use

Testing/Validating the Specification

- **Specification inspection**
  - inspectors use a checklist of items to look for
  - Are requirements clear? Can they be misinterpreted?
  - Are all the cases handled? Are there inconsistencies?
  - Is the requirement testable?
  - Is the requirement traceable to project objectives?
  - Can also trace items back to requirements doc
  - record faults found & rate of finding faults

  - Can also measure convergence
  - Have changes in the spec dropped below some threshold?

Summary

- **Specification document**
  - explicitly defines functionality & constraints

- **Ranges from informal to formal**
  - informal
    - e.g., NL
    - very ambiguous, but easy to understand
  - semi-formal
    - e.g., DFD, E-RDs
    - easy to understand, but more precise
  - formal
    - e.g., FSMs, Z
    - very precise & powerful, but hard to learn
    - useful where safety or reliability is a concern