CMPE 276
Software Engineering

Lecture 3
Project Planning
Requirements and Specification

Project Track: Details

- Tuesday, October 18: assign the projects and form the teams.
- Thursday, October 20th (tentative): informal requirement presentation for feedback.
- Tuesday, October 25th:
  - 20 minute presentation to class about the project
  - Paper report on the project
  - ...

Projects so far:

- Extension to WebSocket protocol
- Servlet configuration and management
- Shared paper repository on the web
- Location-aware search
- Collaboration using HTTP in user space (non-HTML documents)

Other ideas:
- Earth mapping tool
- Lego Mindstorms-Palm pilot protocol
Every project must have a customer!

Grading

- Project: 60% of the grade
  - 25% document
  - 25% project
  - 10% presentations (each presentation by a different team member)
- This is a real project
  - We want a usable outcome
  - No "make believe"
- Evaluations:
  - The project is evaluated by me, and by other teams
  - Team member evaluation by the team

Team Dynamics

- People problems trump technical issues
  - Often the reason for failure
- Team members have different goals
  - Do a great project
  - Get a good grade in the class
  - Survive the class
- Spend time discussing your goals
  - And work throughout the semester to make and keep commitment to shared goals
- Set the tone early
  - Be the group you want to be on the first day
  - Team culture is like cement
  - Sets quickly and hard to alter later
Team Problems

- Team members don’t meet commitments
  - Promise one thing
  - Do something else

- Terrible for teams of any size
  - Makes planning impossible
  - Can’t give the unreliable one anything significant

- Everyone needs to take responsibility
  - But recognize also that everyone made mistakes

Tools track:

- First homework (assigned next class):
  - UML
  - Statecharts
  - Applied to an example
  - Usage report (strong/weak points, etc)

- First tool:
  - Most likely, Visio (see link on web page)

Project Planning in Four Easy Parts

- Break project into tasks
  - Requires a good design with good interfaces
  - Allows tasks to be correctly enumerated
  - Allows places for parallel development to be identified
  - Apm: interfaces how to be right or unexpected dependencies will be discovered later

- Realism in estimating task length

- Observability of completion
  - Tasks are clearly done or not done

- Prioritization

Plan from Design

- Start with the design

- Break project into tasks
  - Invisible units of work for one person

- Rules of thumb:
  - Nothing less than a day is a task
  - Anything more than a week is at least two tasks
  - Larger tasks harder to estimate
  - Need to think about how to break it into logical pieces

Task Dependency Graph

Estimate Time

Scale estimated time by a fudge factor
- Allows for the inevitable unexpected problems
Critical Path

- Each task requires resources
  - Particular people
  - Money
  - Perhaps special hardware, etc.
- Make sure these will be available
  - E.g., one person isn’t expected to be doing two tasks at the same point in the schedule

Resources

Measuring Progress

- Checking off tasks gives illusion of progress
- Real progress only if task completion is observable
  - Bad
    - Task 1: 10% of feature, task 2: 20% of feature
    - What does 10% mean?
  - Good
    - Task 1: All menus implemented and respond correctly to mouse click.

Move from Working System to Working System

- Get the first build up as soon as possible
- After that, always maintain a working system
- System grows as tasks are checked off
- Move from build to build

Why?

- Can observe true progress
  - If nothing runs, hard to know if we are close to running
- Makes changes in plan easier
  - Each build provides a natural point for changes
- Allows priorities
  - Put most critical features in first build
  - If schedule slips, just don’t get to lower-priority builds late in the schedule

Builds
Summary

- Tasks are unit of work
  - But tasks need to make sense
  - Realistic duration, know when they are done
- Group tasks into builds
  - Guaranteed we aren’t completing lots of tasks without checking that everything works together!
  - Another form of false progress

Determining the Requirements

Techniques

- Interviewing
- 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
  - How they teach you what they do
  - Get to workplace and watch them do the task
- In all types of interviews, get details
  - Ask for copies of reports, logs, email in process
  - These may support, fill in, or contradict what the user said

Requirements as Anthropology

“Take the attitude that nothing any person does is done for no reason; if you think it’s for no reason, you don’t yet understand the point of view from which it makes sense. Take the attitude that nothing any person does is unique to them, it always represents an important class of customers whose needs will not be met if you don’t figure out what’s going on.”

- p. 63, Contextual Design, by Beyer & Holtzblatt

Disadvantages of Talking

- Interviews are useful, but

- 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 mockups
  - 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
  - Surveys
- Rapid prototyping
  - As appropriate...

Specifications

- Describe the functionality of the product
  - Precisely
  - Covering all circumstances
- Move from the finite to the infinite
  - Finite example (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/User’s
  - Specification 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%"

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%"

What are sales? Orders received but not yet paid for? Only orders paid for?
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%"

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 flow
  - "what happens, not how it happens"
Step 1: Draw the DFD

1st refinement (infinite # of implementations)

CUSTOMER DATA

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>order</th>
<th>process orders</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>credit status</td>
</tr>
</tbody>
</table>

CUSTOMER DATA

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>invoice</th>
<th>process orders</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>credit status</td>
</tr>
</tbody>
</table>

Step 1: 2nd refinement of DFD

CUSTOMER DATA

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>order</th>
<th>process orders</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>credit status</td>
</tr>
</tbody>
</table>

Step 2: Decide What to Computerize

- Specify what the system does
- 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
  - 10% on up to 4 packages, 15% on 5 or more
  - Translate into decision tree
  - Makes it easy to see what is happening

- Other: 0%
- > 4 packages: 15%

Educational institution

- <= 4 packages: 10%
Step 6: Define Physical Resources

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

(This is very '70s...)

Step 7: Determine Input/Output Specs

- Specify
  - UI
    - Screens
    - 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 fixed
  - backup requirements
  - input needs
  - output devices
  - is existing hardware adequate?

Entity-Relationship Diagrams

- Semi-formal technique
- Focuses on data rather than actions
- Really comes out of database world
- Has moved over into object-oriented analysis

Finite State Machines

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

Finite State Machines

- Write out state transition tables
**Finite State Machines**

- **Advantages**
  - more precise than DFAs
  - 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
  - RED 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 store
  - operations

---

**A File System**

Using Z
Jan Dvorsky & Jan Woodock

---

**File operations**
- read: used to read a piece of data from a file
- write: used to write a piece of data to a file
- add: used to add a new piece of data to a file
- delete: used to delete a piece of data from a file

---

**File system operations**
- create: used to create a new file
- destroy: used to destroy an existing file
- open: used to make a file available for the reading and writing of data
- close: used to make a file unavailable for reading and writing
Files

- (Key, Data)
- File
  - contents: Key → Data

Initialisation

- FileInit
- FILE

read

- READ
  - FILE
  - k?: Key
  - d?: Data
  - k? ∈ data.contents
  - d? = contents<k

write

- WRITE
  - FILE
  - k?: Key
  - d?: Data
  - k? ∈ data.contents
  - contents' = contents @ (k? → d?)

add

- ADD
  - FILE
  - k?: Key
  - d?: Data

delete

- DELETE
  - FILE

read

write

add

delete
Key errors

Report := keyNotUse | keyNotInUse | okay

File

if k? : Key
rl := Report

KeyNotInUse

KeyError

k? ∈ don’t contents
rl := keyNotInUse

KeyNotUse

KeyError

k? ∈ don’t contents
rl := keyNotUse

Success

Success
rl := Report
rl := okay

Read ⊆ (Readk ⊆ Success) ⊆ KeyNotInUse
Write ⊆ (Writek ⊆ Success) ⊆ KeyNotInUse
Add ⊆ (Addk ⊆ Success) ⊆ KeyInUse
Delete ⊆ (Deletek ⊆ Success) ⊆ KeyNotInUse

Otherwise

contents, contents’ : Key → Data
k? : Key
d1 : Data
rl := Report

( k? ∈ don’t contents ∧
d1 = contents ∧
contents’ = contents ∧
rl = okay )
∧
( k? ∈ don’t contents ∧
contents’ = contents ∧
rl = keyNotInUse )

File system

[Name]

System
file : Name → File
open : P’Name
open ∈ don’t file

Initialisation

SystemInit
System’
file’ = Ø
Promotion

Promote
System
File
n? : Name
n? e open
file! = 0File
file! = file ? (n? -> 0File)
open! = open

File operations

KeyRead, E | AFile | Read \& Promote
KeyWrite, E | AFile | Write \& Promote
KeyAdd, E | AFile | Add \& Promote
KeyDelete, E | AFile | Delete \& Promote

File access

FileAccess
System
n? : Name
n? e dom file
file! = file

Open
FileAccess
n? e open
open! = open \{ (n?)

Closing a file

Close
Open
FileAccess
n? e open
open! = open \{ (n?)

File management

FileManage
System
n? : Name
open! = open

Create
FileManage
n? e dom file
file! = file \{ (n? -> 0File)

Destroy

Destroy
More reports

Report := keyInUse | keyNotFound | okay | fileExists | fileDoesNotExist | fileNotOpen | fileNotOpen

File errors

FileStream

FileError

n1 : Name
FileOpen

FileError

FileOpen

n1 : Name
FileOpen

File errors

FileStream

FileError

n1 : Name
FileOpen

File system operations

KeyRead 2 KeyReadok  v FileNotOpen  v FileDoesNotExist
KeyWrite 2 KeyWriteok  v FileNotOpen  v FileDoesNotExist
KeyAdd 2 KeyAddok  v FileNotOpen  v FileDoesNotExist
KeyDelete 2 KeyDeleteok  v FileNotOpen  v FileDoesNotExist

Formal analysis

• consistency of requirements
• operation preconditions

Initialisation theorem

System ' Systeminit
Proof

\[ \varphi \in \text{Name} \quad \varphi = \text{open} \quad (\text{Spot}) \]
\[ \text{open' : Name} \quad (\text{pre}) \]
\[ \varphi \in \text{Name} \rightarrow \text{File} \quad \text{open'} \subseteq \text{open} \]
\[ \equiv \text{file' : Name} \rightarrow \text{File} \quad \text{open'} : \text{Name} \]
\[ \text{file' = \text{open file'} \rightarrow \text{file'} = \varphi} \]
\[ \equiv \text{System' + System'} \]

Precondition

\[ \text{KeyRead} \equiv \text{KeyRead} \land \text{FileDoesNotExist} \land \text{FileOpen} \]
\[ \text{pre KeyRead} = \]
\[ \text{pre KeyRead} \equiv \text{pre KeyRead} \land \text{pre FileDoesNotExist} \land \text{pre FileOpen} \]

pre FileDoesNotExist

System
\[ n? : \text{Name} \]
\[ \equiv \text{Report} \]
\[ n? \notin \text{Name} \land \]
\[ n? \notin \text{open file'} \land \]
\[ n? \notin \text{FileDoesNotExist} \]

pre KeyRead

\[ \text{KeyRead} \equiv \exists \text{File} \land \text{Read} \land \text{Promote} \]
\[ \text{pre KeyRead} = \exists \text{Local} \land \text{pre Read} \land \text{pre Promote} \]
\[ \text{pre KeyRead} \equiv \text{true} \]

Comparison of Specification Techniques

- Formal methods
  - powerful and precise
  - difficult to learn & use
  - could be useful where cost/safety/reliability is important

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyRead</td>
<td>n? \in open</td>
</tr>
<tr>
<td>FileDoesNotExist</td>
<td>n? \notin (close file') \land open</td>
</tr>
<tr>
<td>KeyRead</td>
<td>n? \in open file' \land open</td>
</tr>
<tr>
<td>KeyRead</td>
<td>true</td>
</tr>
</tbody>
</table>
Testing the Specification

- Specification inspection
  - Inspectors use a checklist of items to look for
  - Especially check for handling of all cases
  - Can also trace them back to requirements doc
  - Record faults found & rate of finding faults

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

Metrics for Specification Phase

- Five standard metrics
  - Size
  - Cost
  - Duration
  - Effort
  - Quality

- The usefulness of these is a bit suspect
  - At best, depends on skilled user

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., FSM, Z
    - Very precise & powerful, but hard to learn
    - Useful when safety or reliability is a concern