Software Process and Requirements Analysis

Announcements
- HW9: Do commit the map if you have cropped it. Your repo will be fine

Announcements
- CHECK YOUR GRADES!
  - HW1-7, Exam1-2, Quiz1-10 in the Submitty/Rainbow Grades
- HW9 due on Tuesday, Dec 12th
- Final exam: Mon, Dec 18, 3-6pm in West Hall
- Final exam review on Tuesday
  - I’ll post back tests and review slides
  - Check Announcements page for TA office hours during week of Dec 11th. Send us email if you’d like to meet at different times.

Today’s Lecture Outline
- Usability, conclusion
- Software process, overview
- Requirements analysis
- Software processes

Usability
- Dimensions of usability (LES)
  - Learnability
  - Efficiency
  - Safety
  - Simplicity (not a dimension)
- Design principles for learnability
  - Facts: working memory, long-term memory
  - Consistency: internal, external, metaphorical
  - Use simple words, not tech jargon
  - Recognition, not recall

Usability
- Design principles for efficiency
  - Facts: Human motor processor. Helps derive Fitts’s law and Steering law
  - Make important targets big and nearby, or at the edge of screen
  - Avoid steering tasks, provide shortcuts
- Design principles for safety (error handling)
  - Avoid modes and mode errors
  - Use confirmation windows sparingly
Question: Which Menubar is More Efficient and Why?

- **Mac**: Menubar at the very top of the screen

- **Windows**: Menubar separated from the top of the screen by a window title bar

---

Question: Which Context Menu is More Efficient and Why?

- **Pie menu:**

- **Typical linear menu:**

---

**Simplicity**

Source: Alex Papadimoulis

---

**Simplicity, Google Back in 2003**

---

**Simplicity, Google Now**
Design Principles for Simplicity

- “Less is More!”
  - Omit extraneous information, graphics & features
- Good graphic design
  - Few well-chosen colors and fonts
  - Group with whitespace
- Use concise language
  - Choose labels carefully

Document Your System

- Write the user manual
  - Program and UI metaphors
  - Key functionality
  - Do not include: exhaustive list of all menus
- What is hard to do?
  - Who is your target audience?
    - Power users need a manual
    - Casual users might not
  - Piecemeal online help is no substitute

Outline

- Usability
  - Iterative Design
    - Design
      - Design principles
    - Implement
      - Low-fidelity prototypes
    - Evaluate
      - User testing

Low-fidelity Prototype

- Paper is a very fast and effective prototyping tool
- Sketch windows, menus, dialogs, widgets
- Crank out lots of designs and evaluate them
- Hand-sketching is OK — even preferable
  - Focus on behavior & interactions, not fonts & colors
  - Similar to design of your ADTs and classes
- Paper prototypes can even be executed!
  - Use pieces to represent windows, dialogs, menus
  - Simulate computer’s responses by moving pieces around and writing on them

User Testing

- Start with a prototype
- Write up a few representative tasks
  - Short but non-trivial
    - E.g., “add this meeting to calendar”, “type this letter and print it”
- Find a few representative users
  - 3 is often enough to find obvious problems
- Watch them do tasks with the prototype

How to Watch Users

- Brief the user first
  - “I’m testing the system, not testing you”
  - “If you have trouble, it’s the system’s fault”
  - “Feel free to quit at any time”
  - Ethical issues: informed consent
- Ask user to think aloud
- Be quiet!
  - Don’t help, don’t explain, don’t point out mistakes
  - Two exceptions: prod user to think aloud, and move on to the next task when stuck
- Take lots of notes
Watch for Critical Incidents

- Critical incidents: events that strongly affect task performance or satisfaction
- Usually negative
  - Errors
  - Repeated attempts
  - Curses
- Can also be positive
  - “Cool!”
  - “Oh, now I see.”

More Formal User Testing

- Empirical methods collect evidence about how users interact with UI
  - These methods quantify UI designs
  - E.g., lab experiments, field studies, surveys
  - Controlled experiments quantify UI usability
    - Hypothesis: e.g., Mac bar faster than Windows bar
    - Independent variable: e.g., y-coordinate of menu bar
    - Dependent variable: e.g., #errors, #tasks done
  - Statistical methods
  - Ethics!

Usability, Summary

- You are not the user
- Keep human capabilities and UI design principles in mind
- Iterate over your design
- Write documentation
- Make cheap, throw-away prototypes
- Evaluate prototypes with users

Software Process

- Software lifecycle activities:
  - Requirements analysis
  - Design
  - Implementation
  - Integration + Testing and verification
  - Deployment and maintenance
    - Maintenance is costly. The later a problem is found, the costlier it is to fix
- Software process puts these together
  - How do we combine these activities?
  - In what order?

Software Lifecycle

- Activities from inception to end-of-life
- Can take months or years
- Each activity has specific goals
  - Defines a clear set of steps
  - Produces an artifact (i.e., tangible item)
  - Allows for review
  - Specifies actions to perform in next activity

Activities and Their Artifacts

- Requirements analysis produces “requirements documents”
  - Use-case model, supplementary specifications
- Design produces “design models”
  - Class diagrams, interaction diagrams, ADT specs
- Implementation produces, well, … obviously code
  - + specs for classes and individual methods, AFs and RIs
- Readability of code is crucial!
- Testing produces
  - Test suites
Overview Example, Step 1: Requirements, Use-Case Model

- A Dice Game

- Use case: Play Dice Game

Main success scenario:
Player picks up the two dice and rolls them. If face value is 7 then they win… Else …

Example, Step 2: Design: Interaction Diagram

Dynamic object design

Example, Step 2: Design: Class Diagram

Static design

Example, Step 3: Code + specs

"/ a mutable class ... */
class DiceGame {
private Die die1 = new Die();
private Die die2 = new Die();

// Abstraction Function: ...
// Rep invariant: ...
// @modifies: ...
// @effects: 
public void play() {
die1.roll();
int fv1 = die1.getFaceValue();
... }
}

Analogies with Process in Other Engineering Disciplines

- Civil engineering
  - Requirements: architectural design
  - Design: engineering blueprints
  - Implementation: construction
  - Testing: inspections throughout construction and after completion

Main Differences with other Engineering Disciplines

- Requirements changes do not creep into construction
  - In contrast, in software, requirements change constantly even when implementation is well underway
  - Majority of engineering design and construction uses tried and true materials and techniques
  - Software constantly innovates — new languages, new frameworks, new uses
  - Construction projects are mostly on time
    - But it’s a myth that construction projects are never late.
    - E.g., The Big Dig
Today’s Lecture Outline

- Some catch up
- Software process, overview
- Requirements analysis
- Software processes

Requirements Analysis...

Requirements Analysis is Hard

- Major causes of project failure
  - Poor user input
  - Incomplete requirements
  - Changing requirements
- Essential tools
  - Classification of requirements
  - Use cases

Aside: Measures of Complexity

- Function points
  - Roughly, the number of interactions of user with the system
- Cyclomatic complexity
  - Roughly, the number of predicate nodes in CFG
- Lines of code

Classification of Requirements

- FURPS+ model
- The FURPS:
  - Functionality, Usability, Reliability, Performance, Supportability
- The +:
  - Design constraints, implementation requirements (e.g., must use Java), other
Requirements Analysis Artifacts

- Requirements analysis produces:
  - **Use-case model**
    - A set of use cases
    - Specifies the functional requirements (behavior, features) of the system
  - **Supplementary specification**
    - Specifies non-functional requirements (-ilities: usability, reliability, performance, supportability)

Use Cases

- Describe the interaction of the user with the system as TEXT stories
  - The most widely used approach to requirements analysis in modern software practice
  - Requirements are discovered and recorded through use cases
  - All other activities influenced by use cases!

Example Use Case

- **Point-of-sale (POS) system**
- **Process Sale**: A customer arrives at checkout with items to buy. 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.
  - The use case is a collection of scenarios: main success scenario + scenario variations

Another Example Use Case

- **RPI Campus Paths**
- **Shortest Path**: Campus Paths system displays the campus map. The user selects two buildings. The system draws the shortest path on the map.

Brief, Casual and Fully Dressed formats

- **Brief**: one-paragraph summary, usually for the main success scenario
- **Casual**: multiple paragraphs that cover various scenarios
- **Fully dressed**: all steps and variations are written down
  - Developed iteratively

Example Use Case

- **Handle Returns** use case
- **Main success scenario**: A customer arrives with items to return. The cashier uses the POS system to …
- **Alternative Scenarios**
  - If they paid by credit card, but reimbursement transactions to their credit card are rejected, pay by cash
  - If the system detects a failure in the external accounting system, …
Use Cases vs. Feature Lists

- Low-level feature lists --- common in the past:

<table>
<thead>
<tr>
<th>ID</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT1.9</td>
<td>System should accept entry of item identifiers.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FEAT2.4</td>
<td>System should log credit payments to the accounts receivable system.</td>
</tr>
</tbody>
</table>

- Are use cases the better way to discover and record requirements?

Today’s Lecture Outline

- Usability, conclusion
- Software process, overview
- Requirements analysis
- Software processes

Software Process

- Software lifecycle activities:
  - Requirements analysis
  - Design
  - Implementation
  - Testing
  - Deployment and maintenance

- Software process puts these activities together
- Software process forces attention to these activities and their artifacts

Some Software Processes

- Code-and-fix (ad-hoc): write some code, make up some inputs, debug
- Waterfall: 1st: requirements analysis, 2nd: design, 3rd: implementation, 4th: testing
- Iterative (Unified process, Agile, Scrum) repeat activities: (a small chunk of requirements, design, implementation, testing)
- Other

Benefits of Software Process

- A framework to work within
- A management tool
- Forces attention to important activities and their artifacts
  - Won’t forget requirements analysis or requirements documents, or design (including AFs and RIs), or testing, etc.

- Drawbacks?

Project with Little Early Attention to Software Process

[Graph showing Thawing, Productive Work, and Time]
**Project with Early Attention to Process**

- Threading
- Productive Work
- Process
- Begining of Project
- End of Project

**Let’s See Some Software Processes**

**Code-and-fix**

- Advantages
  - Little or no overhead
  - Dive in and see progress quickly
- Dangerous for most projects
  - Applicable for very small, very short-lived projects
  - May ignore important tasks and artifacts (design, testing)
  - Not clear when to start or stop an activity
  - Hard to review
  - Scales poorly to multiple people

**Waterfall**

- Requirements
- Design
- Implementation
- Integration and Testing

Long lifecycle, usually 6 months or more

**Advantages of Waterfall**

- Can work well for projects with very well understood requirements
- Tackles all planning upfront
- Orderly, easy to follow, sequential process
- Stages are well-defined, easy to perform reviews at each stage to determine if the product is ready to advance

**Waterfall Limitations**

- Requirements change
- Waterfall requires a lot of planning upfront
- Waterfall assumes requirements are clear and well-understood
- Rigid, sequential; does not embrace change
- Costly to “swim upstream” back to an earlier phase
- No sense of progress until the very end
- Integration occurs at the very end
  - Defies “integrate early and often” rule
  - Inflexible, no feedback until the very end
  - Product may not match customer’s need
- Reviews are massive affairs
Iterative Processes Work in Short Iterations

Requirements → Design
Implementation & Test & Integration & More Design
Final Integration & System Test → feedback

Short 2-4 weeks
Iterations are fixed in length, timeboxed
System grows incrementally

Advantages of Iterative Process

- Provides constant feedback, problems are visible early
- Accommodate, embrace change
- Appropriate at the beginning of the project when requirements are still fluid
- Developers can address the biggest risk first
  - As costs increase, risks decrease!

Disadvantages of Iterative Process

- A lot of planning and management
- Frequent change of task
- Requires customer and contract flexibility
- Developers must be able to assess risk
  - Must address most important issues

Staged Delivery

Requirements → Design
Stage 1: detailed design, code, debug, test, and deliver
Stage n: detailed design, code, debug, test, and deliver

Waterfall-like beginning
Then short release cycles
- Plan, design, implement, test, deliver
- Delivery possible at the end of any cycle

Staged Delivery Advantages

- Can ship at the end of any release
  - Looks like success to customers, even if not original goal
- Intermediate deliveries show progress, customers are happy, lead to feedback
- Problems are visible early
- Practical, widely used and successful

Next time

- Review
- Final on Monday, December 18th, 3-6PM in West Hall Aud