Specifications

John Jannotti

/course/cs0320/www/lectures/

Mar 7, 2017
Interesting Reading

- http://www.jeffknupp.com/blog/2012/02/07/coding-backwards/
Recall the Software Lifecycle

1. Requirements
2. Specifications
3. Design
4. Coding
5. Testing
6. Maintenance
What Specifications Are

- Requirements describe what your users want to accomplish.
- Specifications describe precisely what your software will do.
  - Must work out inconsistencies in requirements.
  - Must specify exact operations that support requirements.
  - Could be handed off to other programmers.
  - Can exist at several levels of detail.
- A form of documentation (APIs for programmers)
What Specification Are NOT

- Specifications do not describe implementation details.
- “Search results will return in less than 50ms”
  - Not “A KdTree will be used to provide fast responses”
- “The dialog box will have an OK and Cancel Button”
  - Not “The Cancel button will kill the rendering thread”
  - But the spec *should* describe the user’s perception.
- This is true even when providing detailed specifications.
  - A class should describe its exact external interface.
  - But not its internal design (except for “internal” use).
From Storyboards to Unit Tests

- Specifications form a bridge
  - From somewhat imprecise Requirements
  - To the ability to write tests
  - And set precise acceptance criteria

- Specifications force consensus

- But it’s not a one-way process
  - Finding consensus on contradictory requirements?
  - Sometimes writings specs forces a requirements change.
Elements of a Specification

- Interfaces
  - User – Graphical, Web, Command
  - Software – to network APIs, file formats, etc.
  - Hardware

- Functional capabilities

- Performance Levels – “Big O” and/or constants, Memory/Disk

- Safety – Levels of undo, warning for unrecoverable ops, real safety.

- Reliability – MTBF, redundancy, failover behavior.

- Security/Privacy – Encryption, Data handling policies.

- Compatibility & Limitations – File formats, OS/platform.
Term Project Specifications

- Try to write a user manual.
  - Include “screenshots”
  - Describe detailed workflows.
  - Describe outside interfaces

- Write your “acceptance” criteria.
Our specification were not perfect (not even intended to be).

What *did* we specify?
- Input format
- Query format
- Output format
- We “specified” a KD-Tree, which is really design.

What should *you* have specified?
- Performance - query times, memory requirements.
- Reliability - handling bad input.
- Resolve ambiguities - break ties.
- Limitations - ?
Case Study: Java Collections Framework

- Java’s early collection support had problems
  - Inflexible synchronization support.
  - Just Vector, Hashtable, and arrays.
  - Long, inconsistent method names.
  - Inflexible (non-existent) extension mechanism.

- Requirements:
  - Fix those things
We’re specifying software components.
Careful to differentiate spec from internal design.
  ▶ It can get a little muddy (ArrayList).
  ▶ But even “Array” has ambiguity to resolve. Is remove(0) O(1)?
Fix those problems, bit by bit, ideas?
  ▶ Key: extension mechanism
  ▶ How does it solve the rest?
JCF Specifications

- Introduce *Interfaces*
  - Acts as a specification for implementors.
  - Allows for new implementations
  - Implicitly creates consistency
  - Collections.synchronizedMap()

- Take a page from JCF - Use interfaces to spec your work.
The List interface places additional stipulations, beyond those specified in the Collection interface, on the contracts of the iterator, add, remove, equals, and hashCode methods.

- Returns an iterator over the elements in this list *in proper sequence*.
- Appends the specified element to the *end* of this list.

Note that this does not say *how* order is maintained.

No coincidence, it does not specify performance either.
Which allows two very different implementations.

- **ArrayList**
  - “The size, isEmpty, get, set, iterator, and listIterator operations run in constant time.”
  - “The add operation runs in amortized constant time.”
  - Poor spec: Is that add(E e) or add(int index, E e)?

- **LinkedList**
  - “All of the operations perform as could be expected for a doubly-linked list.”
  - What a lazy spec! (it’s also wrong, for size())

- **And many more, actually.**
  - Guava’s ImmutableList is a nice one.
  - ImmutableList.of(”red”, ”green”, ”blue”, ”yellow”);
  - Java 9: List.of(”red”, ”green”, ”blue”, ”yellow”);
Case Study: Foodler’s Android app

- We outsourced the Android work after building iOS
- Several iterations
  - Our lazy sepc: “Like the iOS app, but with Android conventions”
  - Proposed wireframes
  - Iterated three times
- At this level, we didn’t discuss
  - Color schemes, imagery (best to think of “flexibility” as the spec)
  - API to our servers (this is spec work, but iOS captured that)
Things to look for

- Wireframe technique. Avoid pointless questions.
  - Can hide problems though. The “look” of on/off was a hard decision, made late.
- “Lorem Ipsum” is the textual equivalent.
- Annotation technique: Numbered notes.
- Showing interactions, animations are hard in a document.
  - Notes can describe the screen changes
  - Connecting lines can help, used for dialogs.
  - Different states shown (Home: Shopping Bag)
  - Scrolling behaviour described in prose.
  - Tab behaviour is described and “shown”, but only barely.
  - Hard to show all default form filling behavior.
  - Swipe to delete is illustrated with floating hand.