Hints and Tips for Software Project

Lecture 7B

Summary

- Good idea to hold off doing some things until you have finished doing certain other things.
- No reason to stop doing some things once you start doing certain other things.
- So I think the thing to do is never to lose the ability to re-implement from the original design or architecture. Always track the bugs you find to the earliest point in the process you can.
- Open new "phases", but don't close old phases.
- An important point of reviews is to make sure the chain of argument linking the requirements all the way through to code has no breaks in it.

Software Project "Hints and Tips"

1. Great clarity about requirements
2. Ship something ASAP that demonstrates value
3. Ship weekly; build and run daily
4. Open new "phases"; don't close old ones: recalculate from 1st principles
   - Understand and write those 1st principles
   - "Recalculating" has to keep working

Software project "hints and tips":

1. Great clarity about requirements: in particular, call out "non-goals". A very clear idea of who your immediate customer is.
   - helps you make design decisions as you go along
   - placing limits on yourself like this helps creativity.
2. Demonstrate value as quickly as possible; publish a "technical plan" showing requirements along some kind of timeline
   - Funding
   - Creativity
3. Don't go for long without shipping anything. Avoid long periods when you don't build and run your code.
   - Train yourself to feel "jumpy" if you go too long between compilations, builds, runs of the regression tests, your immediate customer accepting a new release
   - Automate your build and your regression testing
   - Fix regressions shown up by your regression test before you work on new code
4. Open new "phases" in your project, but never close old ones: always be able to recalculate your design and implementation from first principles. That means
   - You have to understand and write down those first principles
   - However you "calculate", it has to work time and time again.

References

- [CCTA98] "Managing Successful Projects using PRINCE2", Central Computer and Telecommunications Agency, The Stationery Office Ltd., 1998.
- [Beck00] "Extreme Programming Explained", Kent Beck, Addison-Wesley, 2000

Functional Specification

- A **Functional Specification** is part of a contract between the customer/sponsor (the person with the requirements and the money), and the software supplier.
  - Abstractness
    - Don't "give away" design details
    - In terms of requirements
  - Completeness
    - All requirements covered
    - Propose new requirements?
  - Feasibility
    - No contradictions

The names I have chosen are arbitrary.

A Functional Specification is part of a contract between the customer/sponsor (the person with the requirements and the money), and the software supplier.
Concentrate here on role of the software designer: how you would review a draft F/S before you gave it to the customer?

A difficulty in a Functional Spec is to avoid putting in information about how you are thinking of implementing it that the customer doesn't really need to care about:

Every statement you put in a Functional Spec is a potential liability: don't needlessly "give away" freedom to make decision at implementation time.

So the first issue is abstractness.
But must address everything that the customer really needs. So the other issue is completeness.
It's also worth pointing out that F/S (or closely related documents) are often part of a Competitive bid: other potential suppliers may be submitting theirs in competition.

So Functional Specification also has to show we understood the requirements, both stated and implied.

Finally, what you say in a Functional Specification must be Feasibility: No contradictions: internal and with reality!
Until you have Functional Spec that meets these, there is no point in moving on to the next stage: what I've called the "Architectural Design".

But requirements will change or be clarified.

Plan to make changes to the F/S in an orderly way, making sure you ripple the changes through to things you base on the Functional Specification .

Reference

[RUP] "Rational Unified Process" http://www-306.ibm.com/software/awdtools/rup/

Architectural Design

- The **Architectural Design** is separate from functional specification: potentially many possible "Architectural Designs" for each functional specification, all different.

The "Architectural Design" is separate from Functional Specification: potentially many possible "Architectural Designs" for each functional specification, all different.

The objectives for Architectural Design:

- Abstract data model
- Divide into functional components
- Document interfaces and collaborations formally:
  - Message (member function) names
  - Type signatures
  - Protocols (often neglected)
  - Semantics of messages (as post-conditions on state of recipient?)
- Explain how each element of the functional specification is achieved

Implementation Design

- The **Implementation Design** is where you document (per component):
  - Implementation (how it done using available languages, operating systems, etc.)
  - Division of a single component into modules (e.g. source files)
  - Interfaces and collaborations
  - Chain of reasoning to Architecture, i.e. explain how conceptual class-design is refined into implementation class-design.

A total system will be divided into a handful of “components”, each with minimum coupling with other components, and maximum internal cohesion.

(A good number is seven.)

But components can be replicated or distributed across networked computers.

The “Implementation design” is where you document (per component):

- Implementation (how it done using available languages, operating systems, etc.)
- Division of a single component into modules (e.g. source files)
- Interfaces and collaborations
- Chain of reasoning to Architecture, i.e. explain how conceptual class-design is refined into implementation class-design.

It’s the “Construction” phase in RUP.

Check: complete, correct and feasible

Code Walkthrough

- A line-by-line walkthrough of the source code of a single module.
- Tempting not to do these, but very good at finding bugs!
- And finally, check:
  - It’s serviceable, maintainable and extendible

A line-by-line walkthrough of the source code of a single module.
Tempting not to do these, but very good at finding bugs!

Inspector checks such issues as:

- The choice of low-level algorithms
- Trade-offs, e.g. speed versus code size versus data size versus maintainability
- “Structural” correctness proofs, e.g. termination of loops and recursion, initialisation
- Preconditions and guarding:
  - what are preconditions for each function?
  - how do all callers of each function make sure they observe these?
- Chain of reasoning to Implementation Design
  - formal checks of refinement

And finally, check: it’s serviceable/maintainable/extendible

Testing versus other Validation Techniques

- Success to begin with!
  - Can't fix those last few bugs
  - After you ship, customers find lots of bugs
  - Developer has to debug every customer bug even known problems
  - Can't develop the next release
- No amount of testing in the world will do anything to prevent these.

The best that testing can do is give hints about where the design process was faulty.

May appear to be successful - if tests are superset of what customers does. Problems come:

- At the end of the project: can't fix last few bugs
  - Even trivial bugs need lots of code to be changed
  - Because every fix causes unexpected regressions in unrelated areas
  - So it gets longer between clean test runs.
- Shortly after you ship your product, things go wrong
  - Because customers do things you didn't test, and because the architecture doesn't embody the correct principles.
  - Because debugging approach only works "in the lab", developer has to debug every customer bug personally - even if it's a known problem.
- Customers have to wait longer and longer to get attention; the developer never gets a chance to work on the next release.
- When you try to develop the next release of your product, turns out very hard
  - Because code can't be extended at reasonable cost or quickly
  - Because nobody really understands the architecture
  - Because the code isn't portable to any other platform (with timing differences).

No amount of testing in the world will do anything to prevent these.

Conclusion

- No substitute for coming up with a design (or architecture) that is
  - Based on an analysis and correct understanding of the fundamental principles of the problem
  - Comprehensible by people of all abilities
  - Firmly and reproducibly coupled to the code
- Given that "humans are out of their depth writing software", best approach is lots of Formal Peer Reviews!

No substitute for coming up with a design (or architecture) that is

- Based on an analysis and correct understanding of the fundamental principles of the problem
- Comprehensible by people of all abilities
- Firmly and reproducibly coupled to the code

Given the current state of software engineering tools, And given that the above is a lot to do with people recognising that humans are out of their depth writing software), the best framework for achieving these characteristics is constant Reviews.