Alternative process models

- Incremental and iterative design
- Agile methods (eXtreme Programming)
- Cleanroom software engineering

Promising attacks on essence

Buy vs. build
Rapid prototyping to aid in the iterative elicitation of requirements
New metaphor for development: Software systems should be grown rather than built
Key resource is the great designer; these should be identified early and cultivated

Iterative requirements refinement

Hardest part of building a software entity:
- Deciding precisely what to build
- i.e., establishing the detailed technical requirements
Thus: Most important service of a software builder to clients is the iterative extraction and refinement of the product requirements
Customers never know what they really want
- Even “Make new system work like our old manual information-processing system” is too simple-minded.
- Dynamics of complex software system are difficult to imagine
- So in planning any software activity, it is necessary to allow for an extensive iteration between the client and the designer as part of the system definition

Incremental development

If conceptual structures that underlie a software entity are too complicated to be accurately specified in advance and too complex to be built flawlessly, then we must take a radically different approach
Hypothesis: Software should not be built; rather, it should be grown!
- Analogy with complex systems in nature, e.g., human brain
Candidate method: Mills’ incremental model of software development

Mills’ incremental model

System should first be made to run, even though it does nothing useful except to call the proper set of dummy sub-programs
Then, development proceeds bit by bit, fleshing out the dummy sub-programs into fully functioning modules
Increments can be planned for, and their development can be scheduled in a roll-out strategy
Approach necessitates a “top-down” design
Real applications and variants:
- Beck’s “extreme programming” model
- IBM’s “cleanroom” development method

Agile methods
Agile method: Extreme programming

Incremental method with short iterations
Designed to deliver software to customers when it is needed
Adapts to volatile requirements, even late in the development lifecycle
Involves close collaboration with customers
Testing early and often
Novel in its apparent inattention to specification and design

Extreme programming (XP)

Win for customer:
- get something useful early and often
- no stigma associated with changing requirements; it’s built into the process
- expectations curtailed early if too ambitious

Win for developer:
- forces frequent meetings with customer, increasing likelihood that “right system” will be built
- offloads some of the risk to the customer

User stories

Form of “requirements” in XP
Not synonym for use cases!
- more akin to scenarios
- written by customer
- named and recorded
- 2-3 sentences in length
Obviously tailored to “functional requirements”

Project velocity

Measure of how much work is getting done
Computed as the sum of the estimates of the user stories completed in previous iteration
When planning the next iteration, may not plan to exceed the velocity of the prior iteration
Allows programmers to recover and “clean up” after a difficult iteration
Velocity increases when programmers finish early: they may ask customers for more stories to work on

Release planning

Meeting with customers:
- estimate time required to complete each user story, including testing (time in terms of ideal weeks)
- customer ranks stories by importance
- each release may require multiple iterations to complete
- culminates in a release schedule
Release plans often revisited and adjusted
- Based on dramatic changes to project velocity
- Should hold a new release planning meeting about every 4-5 iterations
**Iteration planning**

Select requirements among those remaining for this release + any bugs that need to be fixed
- Typical iteration interval 1-3 weeks
- Customer chooses which reqts have priority
- Use project velocity to decide how many new requirements can be implemented

From this select set, devise programming tasks
- Programmers then "sign up" for tasks; estimate time
- Estimates should be 1, 2, or 3 ideal days

**Dealing with uncertainty in planning**

“Spike” solutions to control uncertainty in estimates
- prototype of a software design
- very simple program designed to explore a solution
- e.g., simple “time-of-day” client-server program to get sense of time required to go from CORBA IDL to fleshed out stub and skeleton
- harkens back to copilot tasks in surgical teams

**Design unit tests first**

Before writing code for new class:
- design a suite of unit tests
- add to an automated testing environment

Rationale:
- Good unit test suites hard to design; can’t be put off till the end of the project
- Encourages “collective ownership” of all code
- Enable aggressive refactoring and frequent integration

**Pair programming**

Two programmers, one terminal
Shown to increase productivity and quality of code
**Question:** Why might this be the case?

**Aggressive refactoring**

Obvious problem with XP is that reqts are implemented linearly with no foresight
- this is intentional, to prevent development of a reqt that is later dropped
- but it means we may choose a poor design early

Solution is to aggressively refactor code
- this means all code, not just your code
- cannot work without automated unit and integration testing

**Question**

What characteristics of a project would make XP beneficial?
Cleanroom Software Engineering

Cleanroom SE

Testing and corrective maintenance expensive!

Philosophy:
It is cost-effective to establish a fabrication approach that precludes the introduction of defects

Applied to software:
– Rather than build product and work to remove defects...
– Apply discipline to remove defects in specification and design; then fabricate in a clean manner

Process supported by empirical results

Errors and failure rates...

Study of several major projects (IBM) shows:
– significant variation between errors and failure rates
– majority of failures happen very infrequently

Fixing errors with low failure rates contributes little to the reliability of the system

In fact, the fix may introduce errors with higher failure rates!

Software reliability

Correlates with frequency of usage

Measured in mean time to failure (MTTF)

Thus, by randomly selecting a sample of tests that reflects the frequency of usage, we can test to certify reliability up to a desired MTTF

Usage based testing

Black-box approach
Relies on a usage profile, with frequencies of expected use
– E.g., in a mailer, message deletion will be more frequent than inspection of rich headers
– If we can reliably quantify expected usage...
– Then we can use statistical models to decide on the number of tests we need to devise to reach MTTF target

Relies heavily on fidelity of usage model

Cleanroom process

Incremental model
NO unit testing whatsoever:
– deemed to be too expensive
  • because they do not focus on finding defects that are likely to cause failures
– combination of software inspections and formal proofs of correctness used in lieu of testing
– programmer strictly prohibited from testing code

Usage testing to certify reliability
Question
What kinds of systems should benefit from cleanroom SE?

What is the “best” process?
Short answer: There is no best; different problems call for different processes
Examples:
- Small projects with volatile requirements ⇒ XP
- Large projects
  • lots of prior knowledge ⇒ waterfall model
  • with major technical obstacles ⇒ spiral model