Software Development Methodologies

Lecture 5 - Development Processes 2

SOFTENG 750 2013-04-08

Software Development Worst Practices

Worst Practices 1

Underestimating Required Effort

Estimates often too optimistic, not accounting for

- Changing requirements
- Proper design/refactoring and testing
- Technical problems, e.g. with 3rd-party/legacy assets, integration, etc.
- Human problems, e.g. miscommunication, staff turnover, downtime

Underestimating Testing & Release Management

- Unit testing done by devs not enough:
  integration testing, stress/load testing, acceptance testing
- Testing, packaging, deployment, and support requires a dedicated effort

Overdependence on "Experts"

- Often only some people have a good overview of the system
- High risk: What if they leave? Get ill?

Worst Practices 2

Assumptions instead of Requirements

False-consensus bias ("everyone thinks as I do")

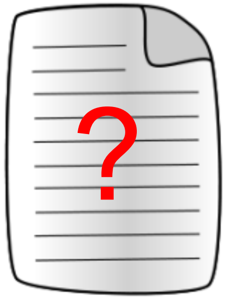
- Relying on assumptions rather than stakeholder requirements
- Not considering how users actually work ("don't fix the user")

Quantity over Quality

Code that just "works" often has hidden costs

- Lack of proper design / refactoring (high maintenance cost)
- Lack of reuse (more code to develop & maintain)
- Lack of exception handling / robustness (high testing/debugging cost)

Insufficient Documentation



Documentation is lower priority than code. It often gets forgotten.

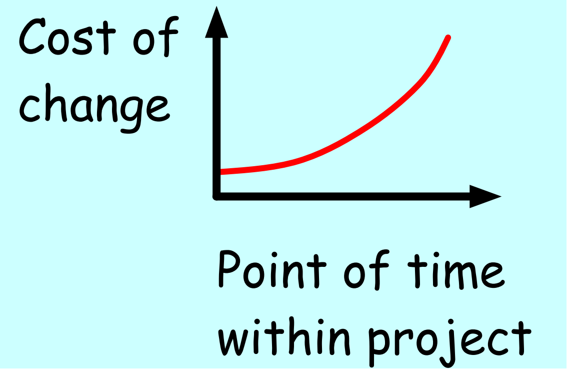
- Lack of understandability: more work for new devs
- Higher maintenance cost and risk

eXtreme Programming
Best Practices

XP

The 5 XP Values

1. Communication
   - Teamwork: consistent shared view of the system
   - Open office environment: developers, managers, customers
   - Verbal, informal, face-to-face conversation
2. Feedback
   - Find required changes ASAP to avoid cost
   - From the customer: through early prototypes & communication
   - From the devs: testing, code review, team estimates
3. Simplicity
   - Build the simplest thing that works for today
   - No work that might become unnecessary tomorrow
   - Simple design easier to communicate
4. Courage
   - To change and to scrap, “embrace change”
   - Better change now (cheaper)
   - Never ever give up!
5. Respect your teammates and your work



XP Practices

Fine-scale feedback

1. Pair Programming: in teams of two, driver and navigator
2. Planning Game: method for project planning with the customer
3. Test Driven Development: first write test cases, then program code
4. Whole Team: teamwork of customer, developer/manager

Shared understanding

5. Use an agreed Coding Standard
6. Collective Code Ownership: everybody responsible for all code
7. Simple Design
8. System Metaphor: consistent, intuitive naming of program parts

Continuous process

9. Continuous Integration: integrate work ASAP
10. Refactoring: improve design whenever possible
11. Small Releases

Programmer welfare

12. Sustainable Pace: no overtime – adjust timing or scope instead

Pair Programming

- **Driver** uses keyboard and mouse, low-level coding
- **Navigator**: reviews driver's work, reference lookup, planning & evaluating options, maintaining TODOs

Advantages:

- Quality generally better (esp. for complex tasks and junior devs)
- Training: very beneficial when pairing up junior and senior devs
- Preference: more job satisfaction and overall confidence
- Efficiency: generally faster than a single dev for a single task (but not necessarily)

Disadvantages:

- Initial Cost: time to learn & practice for a pair
- Efficiency Loss: not twice as efficient as a single developer alone
- Quality benefits can be limited for simple tasks and senior devs
- Preference: not everybody likes it

Refactoring

Improving the design of existing code safely.

- To improve quality attributes: adaptability, maintainability, understandability, reusability, testability

Advantages:

- Can reduce maintenance cost (typically larger than development cost)
- Can make development more efficient (adaptability, reusability, testability, understandability)

Disadvantages:

- Takes time that may be used to develop new features
- Common refactoring do not always improve quality
- Often requires experienced devs to make the right design decisions
- Risk of over-engineering
- Often not noticeable by customer
- Time-to-market sometimes more important than quality

Sustainable Pace (No Overtime)

- IT industry often scores badly here
- Overtime is often caused by incorrect cost estimates
- Overtime can be reduced by using a proper process

Advantages:

- Better morale (important for agile teams & customer relations)
- Lower employee turnover (attrition/churn)
  - Less risk of "brain drain"
  - Reduced cost & risk of hiring & training
- Less downtime

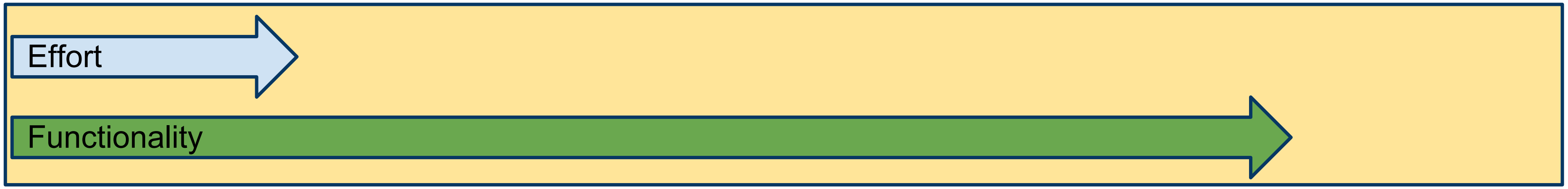
Disadvantages:

- Less flexibility: overtime can boost short-term efficiency
- Deadlines & fixed scope often require overtime
- Time-to-market sometimes more important than quality

More on Best Practices

Pareto Principle

80% of the functionality can be achieved in 20% of the time/effort.
(obviously a rule-of-thumb and not true for every project)



There is a mathematical basis to it: Power laws (Zipf's law)

- Is in theory self similar:
  20% of 20% = 4% of effort….
  ….should achieve 80% of 80% = 64% of functionality
  (but may break down for small projects)
- Consequence: good prototype with 4% of total effort

But conversely: 20% of the functionality takes 80% of the time/effort

- Deceptively fast progress at the beginning
- Many hard problems remaining at the end (bugs, tricky requirements)
- Can lead to overly optimistic time estimates

Software Sizing & Effort Estimation

Software Sizing

- Estimates the size / functional complexity of software
- Common metrics used:
  - Lines of Code (LOC):
    Simple and direct, but depends on technology and coding-style
  - Function Points (FP):
    Quantify functional user requirements (use cases, features) by assigning them points & summing up all points

Effort Estimation

- Effort = Size / Productivity
- Approaches: expert estimation, formal models (e.g. regression)
- Expert estimates are often over-optimistic and overconfident !!!

Specifying Requirements

- Most important artefacts in software development
  - Basis of software development contracts
  - Fulfillment determines the success of the software
- Standards for requirements specification, e.g. IEEE Std 830

a) **Functionality**. What is the software supposed to do?
b) **External interfaces**. How does the software interact with people, the system’s hardware, other hardware, and other software?
c) **Performance**. What is the speed, availability, response time, recovery time of various software functions, etc.?
d) **Attributes**. What are the portability, correctness, maintainability, security, etc. considerations?
e) **Design constraints** imposed on an implementation. Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

Filing Bug Reports

- Bug reports should be managed using a bug/issue tracking system
- Debugging efficiency relies heavily on the information available

1. Is there already a report for a bug? Search for keywords & tags.
   If yes, try to add useful information. Don't create duplicate reports.
2. How does the actual behavior differ from the expected one?
3. What steps need to be performed to reproduce the bug?
   If possible, provide a minimal test case that fails predictably.
4. Provide context information: software version, hardware used etc.
5. Attach as much supporting information as possible:
   screenshot (esp. for UI bug), system log, error message, ...

Managing Releases

- Appoint a release manager: responsible for managing the release
- Create a release branch: release manager decides what goes in
- Create a release plan: who, what, when, how?

Example release plan:

1. Settle on a release scope and release date with the stakeholders
2. Feature freeze: from now on only bugfixes/ improvements
3. Beta testing: build, package and deploy pre-release to beta testers
4. Code review and code freeze of reviewed code
5. Build, document, package and deploy release
6. Announce release to stakeholders

Today’s Summary

Software Engineering
The University of Auckland

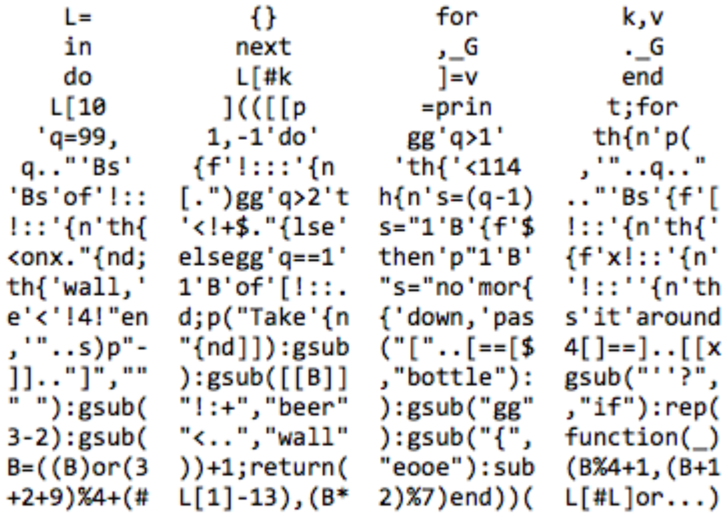
- **Worst Practices** are common and make our life as software engineers difficult:
  underestimation, assumptions, lack of quality, ...
- Many processes such as **XP** define **best practices**:
  pair programming, refactoring, sustainable pace, ...
- Many **other practices** are important for a successful software project: sizing & effort estimates, release management, ...

Further Reading:

- Don Wells. XP - A Gentle Introduction. http://www.extremeprogramming.org
- COSMIC International Software Sizing Standard. ISO/IEC 19761:2011. http://www.cosmicon.com/
- Recommended Practice for Software Requirements. IEEE Std 830-1998. http://www.math.uaa.alaska.edu/~afkjm/cs401/IEEE830.pdf

Quiz

1. In what situations would pair programming be of benefit? Why?
2. How would you decide whether some code should be refactored or not? Give reasons.
3. How should a good bug report look like?



```
     L=           {}           for           k,v
     in          next          ,_G           ._G
     do          L[#k          ]=v           end
    L[10        ](([[p        =prin         t;for
   'q=99,      1,-1'do'      gg'q>1'       th{n'p(
  q.."'Bs'    {f'!:::'{n    'th{'<114     ,'"..q.."
 'Bs'of'!::  [.")gg'q>2't  h{n's=(q-1)  .."'Bs'{f'[
 !::'{n'th{  '<!+$."{lse'  s="1'B'{f'$  !::'{n'th{'
 <onx."{nd;  elsegg'q==1'  then'p"1'B'  {f'x!::'{n'
 th{'wall,'  1'B'of'[!::.  "s="no'mor{  '!::''{n'th
 e'<'!4!"en  d;p("Take'{n  {'down,'pas  s'it'around
 ,'"..s)p"-  "{nd]]):gsub  ("["..[==[$  4[]==]..[[x
 ]].."]",""  ):gsub([[B]]  ,"bottle"):  gsub("''?",
 " "):gsub(  "!:+","beer"  ):gsub("gg"  ,"if"):rep(
 3-2):gsub(  "<..","wall"  ):gsub("{",  function(_)
 B=((B)or(3  ))+1;return(  "eooe"):sub  (B%4+1,(B+1
 +2+9)%4+(#  L[1]-13),(B*  2)%7)end))(  L[#L]or...)
```

Obfuscated Lua - http://www.corsix.org/content/obfuscated-lua

Prints out the lyrics for the song "99 beer bottles"