CS 683 Emerging Technologies: Embracing Change
Spring Semester, 2001
Doc 13 Extreme Programming Overview
Contents

Extreme Programming (XP) 2
Four Variables of Software Development 6
The 12 Practices 9
What is Refactoring 11

References

The New Methodology, Martin Fowler, http://www.martinfowler.com/articles/newMethodology.html

Refactoring: Improving the Design of Existing Code, Martin Fowler, 1999

Extreme Programming Explained: Embrace Change, Beck, 2000

Copyright©, All rights reserved. 2001 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

Extreme Programming (XP)

XP is a lightweight way to produce software

Started March 6, 1996 with

- Project to write a payroll system for Chrysler
- Project done in Smalltalk
- Kent Beck brought in to help an existing project

Other lightweight processes:

- Scrum
- Crystal Light
- Adaptive Software Development
- Dynamic System Development Method
- dX

- Light methods are adaptive rather than predictive

- Light methods are people-oriented rather than process-oriented.

Risk

Software development

- Is a business
- Has risks

  Schedules slip
  Building the wrong product
  Market for product changes
  Buggy software
  Too many features that users don't want
  Struggle between management and developers

- Risks need to be managed

  If can't manage risks than it is hard to succeed

The Struggle between Management & Developer
The Developer Bill of Rights

- You have the right to know what is needed, via clear requirements, with clear declarations of priority
- You have the right to say how long each requirement will take you to implement, and to revise estimates given experience
- You have the right to accept your responsibilities instead of having them assigned to you
- You have the right to produce quality work at all times
- You have the right to peace, fun, and productive and enjoyable work

The Customer Bill of Rights

- You have the right to an overall plan, to know what can be accomplished, when and at what cost
- You have the right to see progress in a running system, proven to work by passing repeatable tests that you specify
- You have the right to change your mind, to substitute functionality, and to change priorities
- You have the right to be informed of schedule changes, in time to choose how to reduce scope to restore the original date. You can even cancel at any time and be left with a useful working system reflecting investment to date

Four Variables of Software Development

- Cost

  How much money to spend developing the product

  Too little money makes it hard to develop the product

  Too much money too soon causes problems

- Time

  The time allocated to develop the product

  Too little time makes development impossible

  Too much time hurts development
  - Market may change
  - Lose feedback from system in use

- Quality

  The least measurable variable

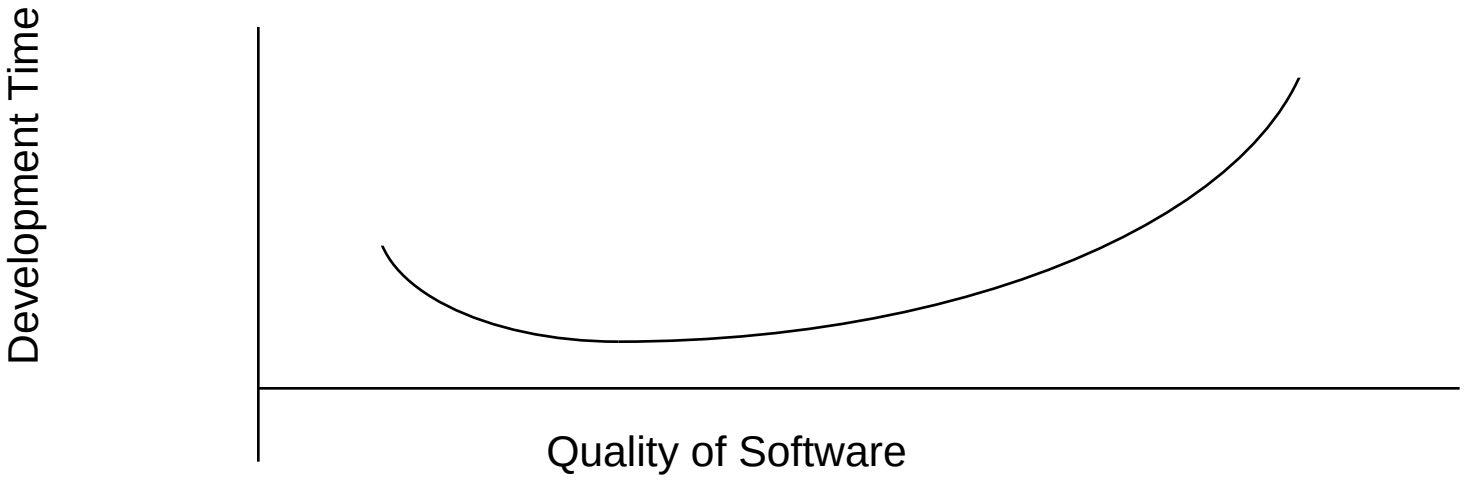
  The first to suffer in a crunch

  Poor quality increases development time and costs

- Scope

  What features will be in the product

Quality and Development Time



Interaction between Four Variables

The interaction between is non-linear

If a program takes one programmer a year to develop

Can we use 50 programmers to develop it in a week or

Use 250 programmers to develop it in a day?

The 12 Practices

- 40-hour work week
- Small Releases
- On-site Customer
- Pair Programming
- Continuous integration
- Testing
- Collective Ownership
- Coding Standards
- Simple design
- Metaphor
- Planning game
- Refactoring

The Planning Game - Greatly Simplified

Customer writes user stories for the product

User story is a short description of the behavior of the product

Customer ranks stories by importance

Developers estimate development time per story

Using customer ranks & developers estimates the stories for next iteration are selected

Development Simplified

Developers
- Break story into small tasks
- Select the task they wish to work on

To implement a task
- Write tests for the task
- Write the code for the task
- Refactor to clean up design
- Integrate task into code base

What is Refactoring

Refactoring is the design of existing code with adding more functionality

What does that mean?

Refactoring Example - Problem 3

This code stinks

```
at: anInteger put: anObject
   (smallKey ~= largeKey)
      ifTrue:
         [(anInteger < smallKey)
            ifTrue: [self atLeftTree: anInteger put: anObject]
            ifFalse: [(smallKey = anInteger)
               ifTrue: [smallValue := anObject]
               ifFalse: [(anInteger < largeKey)
                  ifTrue: [self atMiddleTree: anInteger put: anObject]
                  ifFalse: [(largeKey = anInteger)
                     ifTrue: [largeValue := anObject]
                     ifFalse: [(largeKey < anInteger)
                        ifTrue: [self atRightTree: anInteger put: anObject]]]]]]
      ifFalse:
         [self addNewKey: anInteger with: anObject].
```

Apply "Replace Nested Conditional with Guard Clauses" refactoring to get:

```
at: anInteger put: anObject
   smallKey ~= largeKey
      ifFalse: [^self addNewKey: anInteger with: anObject].
   anInteger < smallKey
      ifTrue: [^self atLeftTree: anInteger put: anObject].
   smallKey = anInteger ifTrue: [^smallValue := anObject]
   anInteger < largeKey
      ifTrue: [^self atMiddleTree: anInteger put: anObject]
   largeKey = anInteger ifTrue: [^largeValue := anObject]
   largeKey < anInteger
      ifTrue: [^self atRightTree: anInteger put: anObject]
```

More Refactorings

To get the code below I applied:

- Replace Conditional with Polymorphism
- Introduce Null Object

```
TernarySearchTree>>at: aKey
   ^root at: aKey

TernaryTreeNode>>at: aKey
   aKey < smallData key ifTrue:[^leftSubtree at: aKey].
   aKey = smallData key ifTrue:[^smallData value].
   aKey < largeData key ifTrue:[^middleSubtree at: aKey].
   aKey = largeData key ifTrue:[^largeData value].
   aKey > largeData key ifTrue:[^rightSubtree at: aKey]

LeafTreeNode>>at: aKey
   aKey = data key
      ifTrue:[^data value]
      ifFalse:[self error: 'Key not found']

NilTreeNode>>at: aKey
   self error: 'Key not found'
```