Extreme Programming

CS 236503
Dr. Oren Mishali

Outline

• Overview of XP
• Design strategy
• Agile testing
• Continuous Integration
XP Origins

C3 Project - 1996

1999

XP’s Values and Practices

Note: this scheme reflects the original XP version. Both values and practices have evolved in the second edition of XP
Release Planning Game

- Customers & Developers meet to create “User Stories”, divided into Iterations

In the following slides we’ll get a taste of XP. We will cover XP’s practices in more depth throughout the course.

#130

As a Product Owner
I want to add a picture to my profile
So that other users can see my beautiful face

Goal: 2 (Find content)
Engagement: Engage
Estimate: 5

Development Iteration

- Very short: 1-3 weeks
- User stories + Acceptance tests
- On-site customer

Pair Programming
Design-Coding-Testing

- Waterfall
  - Design => Coding => Testing
- XP
  - They take place all together

Code Refactoring
Simple Design

Don’t build this … if all you need is this.

YAGNI
• You Ain’t Gonna Need It

Test-Driven Development (TDD)
Extensive Test Suite

• Is developed and executed daily
• Provides confidence to make changes

Design Strategy

From an agile (XP) perspective
About XP’s design strategy

• The design strategy in XP is to always have the simplest design that runs the current test suite

• **Why simple design?**
  • A complicated design is harder to communicate
  • Using simple design, we quickly move to coding and get to know whether the design is right or wrong
  • Programmers tend to anticipate future problems, simple design tends not to guess the future

XP’s design strategy in a nutshell

• Start with a test
  • So we know when we are done
• Design and implement just to pass the test
  • What are the objects and their visible methods?
• Repeat
• See a chance to make the design simpler? *Refactor!*
XP’s design in action #1 – test driven design

• Pick up the first test case
• “If all we had to do was to implement this test case, then we would only need one object with two methods”
• Implement the object and the two methods
• Our whole design is one object (for about a minute)

XP’s design in action #2 – refactoring

• Pick up the next test case
• Instead of “hack in a solution”, we restructure the existing one object into two objects
• Run the first test case to make sure it works
• Implement the next test case
XP’s design in action #3 – team work

• After a day or two of working in this style, the system is big enough to allow two pairs working that way without stepping on each other
• Another day or two and the system can support the whole team developing that way

XP’s design metaphor – learning to drive

• In the XP view, design is not drawing a bunch of pictures and then implement the system to confirm to the pictures
  • That would be pointing the car
• Learning to drive points to a different design style
  • Get the car started
  • Point is a little this way, then a little that way

“A day without refactoring is like a day without sunshine”
– Kent Beck
What is “simple design”? 

• Is it a design with the fewest classes?  
  • This would lead to big and ineffective objects  
• Is it the design with the fewest methods?  
  • This would lead to big methods and duplication  
• Is it the design with the fewest lines of code?  
  • This would lead to code compression and a loss of communication  

Recall: the design strategy in XP is to always have the simplest design that runs the current test suite

XP’s simple design – the 4 constraints

1. The system (code and tests) must communicate everything you want to communicate  
2. The system must contain no duplicate code  
3. The system should have the fewest possible classes  
4. The system should have the fewest possible methods
Understanding the 4 design constraints

• The purpose of the design of the system is
  • First, communicate the intent of the programmers
  • Second, to provide a place for the logic of the system to live
• The 4 constraints provide a framework that satisfies these 2 requirements
  • Communicative design leads to objects and methods for important concepts, with meaningful names
  • Communication is important but you must eliminate duplicate logic

Here comes the 3 other constraints. Beck says that eliminating duplication “is the hardest part of design”

Do not be afraid to delete code

• Found a class or method that does nothing and communicates nothing? Delete it
• An alternative XP’s view to simple design
  • You have a system that runs the test cases
  • Delete everything that doesn’t have a purpose
  • You are left with the simplest design that could possibly work
About refactoring

• One of XP’s practices
• Programmers constantly look for ways of making the existing program simpler
• Refactoring – making the program simpler and more organized without changing the functionality (while still running all the tests)
• You refactor when the system asks you too – when you identify “code smells”
• Without a comprehensive test-suite you will not have the confidence to refactor

Refactoring

... is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Its heart is a series of small behavior preserving transformations. Each transformation (called a “refactoring”) does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it’s less likely to go wrong. The system is kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.
Sometimes big refactoring is needed

• Often, the team feels that the system needs overall reorganization
• Somebody then calls “Time out”
• The team get together for a day, and restructures the system as a whole using CRC cards, sketches, and refactoring

As a developer, when faced with big refactoring, take it in small steps. During regular development move a method here, a variable there. Eventually, all that will remain of the big refactoring is a little job.
Designing with “pictures” – the XP strategy

- **Small initial investment** – draw a few pictures at a time
- **Play to win** – don’t use pictures just because you need to
- **Rapid feedback** – quickly find out if the pictures are on target or not
- **Working with people’s instincts** – encourage pictures from those who work best with pictures
- **Embrace change and travel light** – don’t save pictures once they have had their effect on the code, since the decisions they represent will probably change tomorrow anyway

- But what about all those pretty picture of designs and analyses?
- Some people really do think better about their designs in terms of pictures instead of code
- How does a “visually oriented” person make contribution to the design?

Testing in XP

Extreme Programming

Extreme Programming explained

Embrace Change

Kent Beck
Testing is one of XP’s core practices

- Any program feature without an automated test – doesn’t exist
- Programmers write unit tests
- Customers write functional tests
- So their confidence in the operation of the program becomes part of the program itself

“We will write tests before we code, minute by minute. We will preserve these tests forever, and run them all together frequently. We will also derive tests from the customer’s perspective.”

No need to write a test for every single method you write, only production methods that could possibly break.

Testing makes the program more confident

- Every time a programmer writes some code, they think it is going to work
- So they take that confidence “out of the ether” and turn it into an artifact that goes into the program
- The same story works for the customer
- Every time they think of something concrete the program should do, they turn it into another piece of confidence that goes into the program
Nobody likes to do testing!

- If we want programmers and customers to write tests, we should make the process as painless as possible
- If it was possible to develop without tests, we would dump all the tests in a minute

Tests in XP are isolated and automatic

- **Isolated** – each test doesn’t interact with the others you write
- That way you avoid the problem that one test fails and causes a hundred other failures
- **Automatic** – the tests must be automatic and return a “thumbs up/thumbs down” indication of whether the system is behaving
What should be tested?

- It is impossible to test absolutely everything
- You should test things that might break
- Testing is a bet that pays off when your expectations are violated
- For example, when a test works that you didn’t expect to work
- Or a test breaks when you expected it to work

If you could you would only write those tests that pay off, but you can’t so you write tests that might pay off.

As you test, you reflect on that and improve your test writing skills

Cases where unit tests are written

- If the interface of a method is unclear, write a test before you write the method
- If the interface is clear but the implementation is a bit complicated, write a test before you write the method
- If you think of unusual circumstance in which the code should work, write a test to communicate the circumstance
- If you find a problem later, write a test that isolates the problem

Unit tests always run at 100%

If one of the unit tests is broken, no one on the team has a more important job than fixing the tests
Customer tests

- The customers write tests story-by-story
- They should ask themselves: “what would have to be checked before I would be confident that this story was done”?
- Each scenario they come up with turns into a test, in this case a functional test
- The functional tests don’t run at 100% all of the time
- As you get close to a release, the customer will need to categorize the failing functional tests. Some will be more important to fix than others.

Who writes the customer tests?

- Customers typically can’t write functional tests by themselves
- They need the help of someone who can first translate their data into tests, and can create tools that let the customer write, run, and maintain their own tests
- That’s why an XP team of any size carries at least one dedicated Tester
- The tester’s job is to translate the sometimes vague testing ideas of the customer into real, automatic, isolated tests
Continuous Integration

Continuous integration – an XP practice

- Code is integrated and tested after a few hours, a day of development at most
- One simple way to do so is to have a machine dedicated to integration
- When the machine is free, a pair with code to integrate sits down, loads the current release, loads their changes, checking for and resolving any collisions, and runs the tests until they pass 100% correct
- Integrating one set of changes at a time works well because it is obvious who should fix a test that fails – we should, since we must have broken it, since the last pair left the tests at 100%
Continuous Integration by Martin Fowler

• In the next slides we will provide highlights from an article about continuous integration written by Martin Fowler
• The article is available at https://www.martinfowler.com/articles/continuousIntegration.html

A basic Continuous Integration flow (1)

Let's assume I have to do something to a piece of software, it doesn't really matter what the task is, for the moment I'll assume it's small and can be done in a few hours.
A basic Continuous Integration flow (2)

I begin by taking a copy of the current integrated source onto my local development machine. I do this by using a source code management system by checking out a working copy from the mainline.

A basic Continuous Integration flow (3)

Now I take my working copy and do whatever I need to do to complete my task. This will consist of both altering the production code, and also adding or changing automated tests.

Continuous Integration assumes a high degree of tests which are automated into the software. Often these use a version of the popular XUnit testing frameworks.
A basic Continuous Integration flow (4)

Once I'm done (and usually at various points when I'm working) I carry out an automated build on my development machine. This takes the source code in my working copy, compiles and links it into an executable, and runs the automated tests. Only if it all builds and tests without errors is the overall build considered to be good.

A basic Continuous Integration flow (5)

With a good build, I can then think about committing my changes into the repository. The twist, of course, is that other people may, and usually have, made changes to the mainline before I get chance to commit. So first I update my working copy with their changes and rebuild.
A basic Continuous Integration flow (6)

If their changes clash with my changes, it will manifest as a failure either in the compilation or in the tests. In this case it's my responsibility to fix this and repeat until I can build a working copy that is properly synchronized with the mainline.

A basic Continuous Integration flow (7)

Once I have made my own build of a properly synchronized working copy I can then finally commit my changes into the mainline, which then updates the repository.

However my commit doesn't finish my work. At this point we build again, but this time on an integration machine based on the mainline code. Only when this build succeeds can we say that my changes are done. There is always a chance that I missed something on my machine and the repository wasn't properly updated. Only when my committed changes build successfully on the integration is my job done. This integration build can be executed manually by me, or done automatically by Cruise (a CI tool)
Additional reading

• More practices and details can be found in Fowler’s article at https://www.martinfowler.com/articles/continuousIntegration.html