xTreme Programming
(summary of Kent Beck’s XP book)
Contents

- The software development problem
- The XP solution
- The JUnit testing framework
The Software Development Problem
Risk Examples

- delivery schedule
- project cancelled
- high defect rate - system unusable
- business misunderstood or changed
- false feature rich
- staff turnover
Four control variables in SW development

- Cost
- Time
- Quality
- Scope

- External forces (customers, management) pick the values of any three of the variables
- Development team picks the value of the fourth variable
- Make the four variables visible
Interaction between the variables

- **Cost:** more money can help a little
  - Too much money creates more problems than it solves
- **Time:** more time can improve quality and increase scope
  - too much time will hurt it
- **Quality:** short-term gains by deliberately sacrificing quality
  - but the cost (human, business, technical) is enormous

- **less scope** => better quality (as long as the business problem is still solved)
The four values of XP

- Communication
- Simplicity
- Feedback
- Courage
Short-term vs. long term thinking (I)

- Communication:
  - effect of pair programming, unit testing, task estimation – programmers, customers and managers have to communicate
  - Monitored by a coach

- Simplicity: it is better to do a simple thing today and pay a little more tomorrow to change it, than to do a more complicated thing today that may never be used anyway
Short-term vs. long term thinking (II)

- Feedback:
  - when customers write new „stories“ (description of features, simplified use cases):
    - the programmers immediately estimate them;
    - customers and testers write functional tests for all the stories
  - must be enabled by putting the system in production as early as possible

- Courage: throwing parts of the code away and start over on the most promising design
Basic principles (derived from the four values)

- Rapid feedback
- Assume simplicity
- Incremental change
- Embracing change
- Quality work
Some more principles

- small initial investment
- play to win
- concrete experiments
- open, honest communication
- work with people‘s instincts, not against them
Basic activities in the XP development process

- Coding
- Testing
- Listening
- Designing
The Solution
The solution: overview

- Practices
- Management strategy
- Planning strategy
XP Practices (I)

- Planning game: determine the scope of the next release;
  - as reality overtakes the plan, update the plan

- Small releases: release new versions on a very short cycle after putting a simple system into production quickly

- Metaphor: guide development with a simple shared story of how the whole system works
XP Practices (II)

- Simple design: as simple as possible but not simpler (A. Einstein)

- **Testing**: continually write unit tests

- Refactoring: restructure the system to remove duplication (c.f. framelets, etc.)

- Pair programming: two programmers at one machine

- Collective ownership
XP Practices (III)

- Continuous integration: integrate the system many times a day, every time a task is complete.
- 40-hour week.
- On-site customer: include a real, live customer.
- Coding standards.
Practices support each other
Management Strategy: Overview

- decentralized decision making, based on
  - metrics
  - coaching
  - tracking
  - intervention

- using business basics: phased delivery, quick and concrete feedback, clear articulation of the business needs
Metrics

- don’t have too many metrics

- numbers are regarded as a way of gently and non-coercively communicating the need for change

- ratio between the estimated development time and calendar time is the basic measure for running the Planning Game
Coaching

- be available as a development partner
- see long-term refactoring goals
- explain the process to upper-level management

⇒ no lead programmer, system architect, etc.
  ⇒ Get everybody else to make good decisions
Tracking

- It is the job of the tracker to gather whatever metrics are being tracked at the moment and make sure the team is aware of what was actually measured (and reminded of what was predicted or desired).

- Running the Planning Game is a part of tracking. The tracker needs to know the rules of the game and be prepared to enforce them.
Intervention

- when problems cannot be solved by the emergent brilliance of the team, the manager has to step in, make decisions and see the consequences through to the end

- sample situations: changing the team‘s process, personnel changes, quitting a project
Planning Strategy: Overview

- bring the team together
- decide on scope and priorities
- estimate cost and schedule
- give everyone confidence that the system can be done
- provide a benchmark for feedback

- put the most valuable functionality into production asap
XP Summary
What makes XP hard?

It’s hard to ...

- do simple things
- admit you don’t know (eg, basics about computer/software science in the context of pair programming)
- collaborate
- break down emotional walls
XP & Kent Beck (I)

Kent Beck is afraid of:

- doing work that doesn’t matter
- having projects canceled
- making business decisions badly
- doing work without being proud of it
Kent Beck is not afraid of:

- coding
- changing his mind
- proceeding without knowing everything about the future
- relying on other people
- changing the analysis and design of a running system
- writing tests
Testing: the JUnit 3.x framework
(with UML-F diagrams)
The JUnit components (I)

- Adding new test cases: JUnit provides a standard interface for defining test cases and allows the reuse of common code among related test cases.
- Tests suites: Framework users can group test cases in test suites.
- Reporting test results: the framework keeps flexible how test results are reported. The possibilities include storing the results of the tests in a database for project control purposes, creating HTML files that report the test activities.
The JUnit components (II)

- Overview of JUnit design

For example, class `ComplexTest` defines test cases for complex numbers
The TestCase variation point (I)

- The *TestCase* design is based on the Template Method design pattern
  - The method `run()` controls the test execution

```java
public void run() {
    setUp();
    runTest();
    tearDown();
}
```
The initialization part is responsible for creating the test fixture.

The test itself uses the objects created by the initialization part and performs the actions required for the test.

Finally, the third part cleans up a test.
The TestSuite variation point

- **TestCases** are grouped into **TestSuites**
- variation of the Composite design pattern

- Black-box adaptation
The TestResult variation point (I)

- **Failures** are situations where the `assert()` method does not yield the expected result.

- **Errors** come from unexpected bugs in the code being tested or in the test cases themselves.

- The *TestResult* class is responsible for reporting the failures and errors in different ways.
The TestResult variation point (II)

- TestResult must provide four methods:
  - `startTest()` - initialization code
  - `addFailure()` - reports a failure
  - `addError()` - reports an error
  - `endTest()` - clean-up code
The TestResult variation point (III)
Adapting JUnit

- Cookbook recipes and UML-F diagrams for each of the JUnit variation points
  - Create a test case (*ComplexTest*)
  - Create a test suite (for the *ComplexTest* methods)
  - Create an HTML reporting mechanism
Adapting TestCase (I)

- **TestCase adaptation recipe:**
  - Subclass `TestCase`
  - Override `setUp()` (optional). The default implementation is empty
  - Override `runTest()`
  - Override `tearDown()` (optional). The default implementation is empty
Adapting TestCase (II)

- **TestCaseExample** illustrates the code that has to be added by the application developer

- White-box adaptation
Adapting TestCase (III)

- Four possible adaptation examples, considering the optional hook methods.
Adapting TestCase (IV)

- One aspect in the *TestCase* class cannot be captured in UML-F design diagrams
  - Method `runTest()` takes no parameters as input
  - Different test cases require different input parameters
  - The interface for these test methods has to be adapted to match `runTest()`
For adapting testExample1(). One inner subclass has to be defined for each test method. The inner subclass overrides runTest() so that the corresponding test method can be invoked with the appropriate parameters.
Adapting TestCase (VI)

```java
public class ComplexTest extends TestCase {
    private ComplexNumber fOneZero;
    private ComplexNumber fZeroOne;
    private ComplexNumber fMinusOneZero;
    private ComplexNumber fOneOne;

    protected void setUp() {
        fOneZero = new ComplexNumber(1, 0);
        fZeroOne = new ComplexNumber(0, 1);
        fMinusOneZero = new ComplexNumber(-1, 0);
        fOneOne = new ComplexNumber(1, 1);
    }

    public void testAdd() {
        //This test will fail !!!
        ComplexNumber result = fOneOne.add(fZeroOne);
        assert(fOneOne.equals(result));
    }

    public void testMultiply() {
        ComplexNumber result = fZeroOne.multiply(fZeroOne)
        assert(fMinusOneZero.equals(result));
    }
}
```
Adapting TestSuite (I)

Adaptation by overriding the `suite()` method

```
:TestCase

+run() «C-op()»
+addTest(Test) «C-add»
```

```
:TestSuite

fTests

+run() «C-op()»
```

```
«interface»
«C-Comp»

Test

+run() «C-op()-h»
«fixed»
```

```
«C-Leaf»

TestCase

... 
+run() «C-op()»
+suite()
```

```
«C-Composite»

TestSuite

+run() «C-op()-t»
```

Stefan Resmerita, WS2015
Adapting TestSuite (II)

- TestCase and TestSuite are related variation points

```java
public static TestSuite() {
    TestSuite suite = new TestSuite();

    suite.addTest(new ComplexTest("testing add") {
        protected void runTest() { this.testAdd(); }
    } );

    suite.addTest(new ComplexTest("testing multiply") {
        protected void runTest() { this.testMultiply(); }
    } );

    return suite;
}
```
Adapting TestResult

```
+startTest() «Strategy-algolInt()»
+addFailure() «Strategy-algolInt()»
+addError() «Strategy-algolInt()»
+endtTest() «Strategy-algolInt()»
```

```
create the HTML file
```

```
report the failure by appending a line to the HTML file
```

```
report the error by appending a line to the HTML file
```

```
print the number of tests executed so far and closes the file
```
Pattern-annotated diagrams

- Pattern-annotated diagram for the main JUnit classes
From JUnit 3.x to JUnit 4.x
(an annotation-based framework)
Simplified usage through annotations

- JUnit 4 requires Java 5 or newer
- No TestCase class
  - Use an ordinary class for a test case (don’t extend TestCase)
- Use annotations instead of special method names:
  - Instead of a setUp method, put @Before before some method
  - Instead of a tearDown method, put @After before some method
  - You can define multiple test methods with any names, just put @Test before each test method
- Useful assertion methods are defined in the static-only class Assert
Java annotations in a nutshell

- From J2SE 5 up
- Allow adding decorations to code (resemble Java tags, e.g., `@author`)
- Used for code documentation, compiler processing, code generation, runtime processing
- Small set of predefined annotations (e.g., `@Deprecated`)
- New annotations can be created by developers
import org.junit.*;
import static org.junit.Assert.*;

public class ComplexTest {
    private ComplexNumber fZeroOne;
    private ComplexNumber fMinusOneZero;
    private ComplexNumber fOneOne;
    @Before
    public void setUp() {
        fZeroOne = new ComplexNumber(0, 1);
        fMinusOneZero = new ComplexNumber(-1, 0);
        fOneOne = new ComplexNumber(1, 1);
    }
    @Test
    public void testAdd() {
        //This test will fail !!
        ComplexNumber result = fOneOne.add(fZeroOne);
        assertTrue(fOneOne.equals(result));
    }
    @Test
    public void testMultiply() {
        ComplexNumber result = fZeroOne.multiply(fZeroOne);
        assertTrue(fMinusOneZero.equals(result));
    }
}
Before and After

- **@BeforeClass**: one-time initialization, when the class is loaded
- **@AfterClass**: clean-up method, after all tests have been completed
- Multiple **@Before** and **@After** methods possible, no order assumed – for example:

```java
@Before
public void setUpZeroOne() { fZeroOne = new ComplexNumber(0, 1); }

@Before
public void setUpMinusOneZero() { fMinusOneZero = new ComplexNumber(-1, 0); }

@Before
public void setUpOneOne() { fOneOne = new ComplexNumber(1, 1); }
```
Additional features

- Execution time limit to avoid infinite loops
  - `@Test (timeout = 10)`

- Specification of expected exceptions
  - `@Test (expected = ArrayIndexOutOfBoundsException.class)`

- Parameterized tests: A series of tests which differ only in the inputs and expected results can be written as a single test by providing:
  - A static method that generates and returns a list of data items
  - A single constructor that stores one data item to test
  - A test method
Example of parameterized test

```java
RunWith(Parameterized.class)
public class ComplexTestParam {
    private ComplexNumber fZeroOne;
    private ComplexNumber expectedResult;
    private ComplexNumber secondTerm;

    public ComplexTestParam(ComplexNumber expected, ComplexNumber secondTerm) {
        this.expectedResult = expected;
        this.secondTerm = secondTerm;
        fZeroOne = new ComplexNumber(0, 1);
    }

    @Parameters
    public static Collection<Object[]>
generateInputs() {
        return Arrays.asList(new Object[][]{
            {new ComplexNumber(-1, 0), new ComplexNumber(0, 1)},
            {new ComplexNumber(1, 0), new ComplexNumber(1, 0)}, //wrong
        }));
    }

    @Test
    public void testMultiply() {
        ComplexNumber result = fZeroOne.multiple(secondTerm);
        assertTrue(result.equals(expectedResult));
    }
}
```
Test suites in JUnit 4.x

- A test suite is defined by an annotation to some (possibly empty) class

- Example:

```java
@RunWith(Suite.class)
@Suite.SuiteClasses({ ComplexTest.class, ComplexTestParam.class })
public class AllComplexTests {
    //empty
}
```

// Complete source code available online (see course homepage)