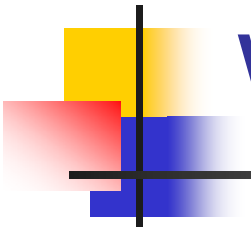
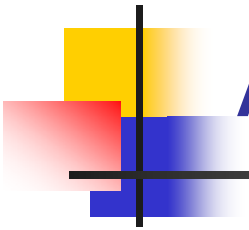
Reading assignment

- G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," **IEEE Transactions on Software Engineering**, 22 (8), August 1996, pp. 529-551.
- T. L. Graves, M. J. Harrold, J. M. Kim, A. Porter and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," **ACM Transactions on Software Engineering and Methodology,** 10 (2), April 2001, pp. 184-208.
- Y. F. Chen, D. S. Rosenblum and K. P. Vo, "TestTube: A System for Selective Regression Testing," **Sixteenth International Conference on Software Engineering**, Sorrento, Italy, May 1994, pp. 211-220.
- T. J. Ostrand, E. J. Weyuker, R. M. Bell, "Where the Bugs Are," **Proceedings of the 2004 International Symposium on Software Testing and Analysis,** Boston, MA, July 2004



What is Agile Development?

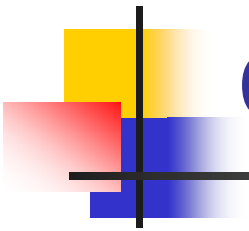
- Family of development processes
  - Allow the team to respond to changes in any phase of development
  - Accepts change as a fact of life and uses process to enable change
- Most notorious Agile Development Process: Extreme Programming



Agile Processes

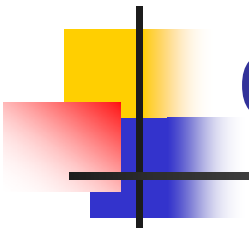
Software processes that are:

- Incremental (small software releases with rapid cycles)
- Cooperative (customer and developer working together with close communication)
- Straightforward (method is easy to learn and modify)
- Adaptive (able to make last moment changes)



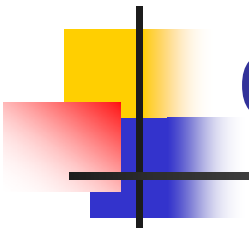
eXtreme Programming (XP)

“Extreme Programming is a discipline of software development based on the values of simplicity, communication, feedback, and courage...



eXtreme Programming (XP)

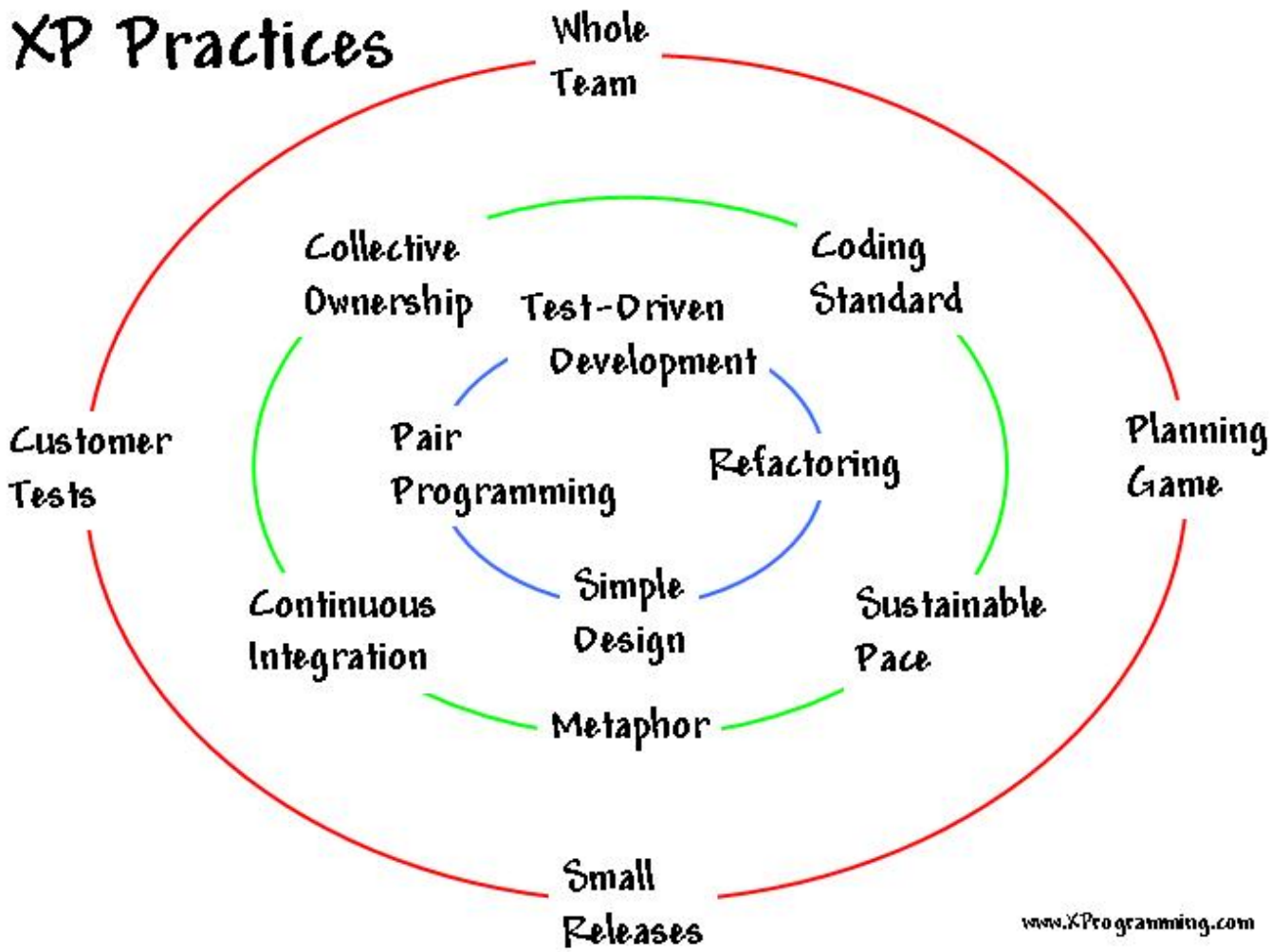
… It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable to the team to see where they are and to tune the practices to their unique situation.” – Ron Jeffries

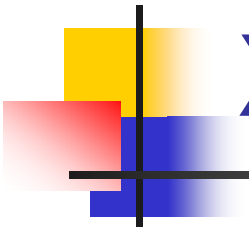


eXtreme Programming (XP)

- Defined by 12 practices
- Claim: XP ‘flattens’ the cost-of-change curve
- Most literature on XP is experience reports
  - Rigorous evaluation is needed

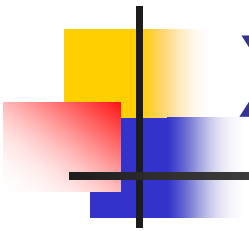
12 Practices of XP





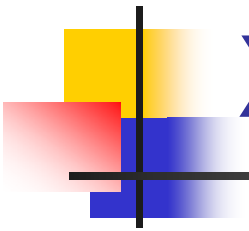
XP Practices: Whole Team

- Everyone working on project is on one team
- “Customer” – provides requirements, steers planning
- Different roles on team, but:
  - No specialists
  - Generally competent people with special skills



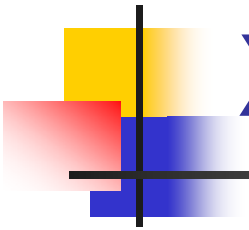
XP Practices: Planning Game

- Address two questions:
  - What will be accomplished by the due date?
  - What to do next?
- Release Planning
- Iteration Planning



XP Practices: Customer Tests

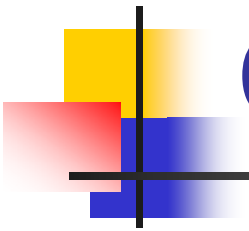
- Automated acceptance tests
- Defined by customer
- Implemented by team



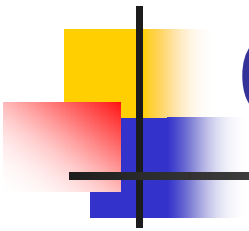
XP Practices: Small Releases

- Release functional, “useful” software every iteration
  - For evaluation by customer, or release to end-users
- Releases are kept reliable by testing

XP Practices: Continuous Integration

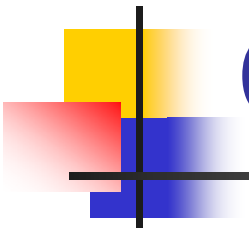


- Constantly keep the entire system integrated
- Multiple daily builds (10-20 in practice!)
- Problems with infrequent integration
  - Team not experienced with integration
  - Buggy code (problems introduced by integration)
  - Long code freezes



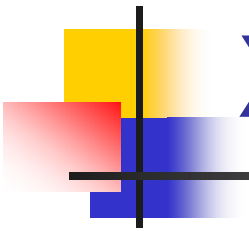
XP Practices: Collective Code Ownership

- Anyone on team can work to improve any piece of code
- Avoids asking code “owner” to add feature
- Problem: working with unfamiliar code
  - Pair with someone familiar with it
  - Automated tests



XP Practices: Coding Standard

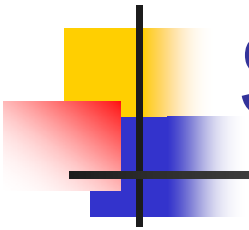
- Common coding standard followed by everyone on team
- Specifics unimportant – as long as code all looks familiar
- Supports collective code ownership



XP Practices: Metaphor

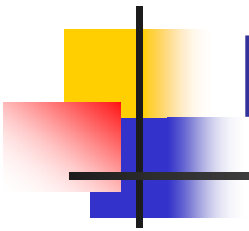
- Metaphor for function of system shared by team
- For example, an agent-based information retrieval system:

“This program works like a hive of bees, going out for pollen and bringing it back to the hive.” [www.xprogramming.com]



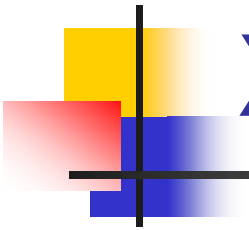
XP Practices: Sustainable Pace

- Maintain pace that will be successful in the long run
- Pace should be sustainable indefinitely
- Work overtime when necessary, but don’t burn out and lose productivity



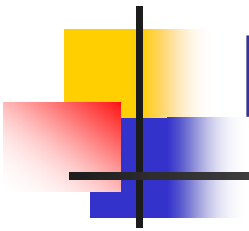
XP Practices: Pair Programming

- All code written by two programmers working side-by-side
- Ensures code is reviewed
- Communicates knowledge throughout team
- Claim: results in better code



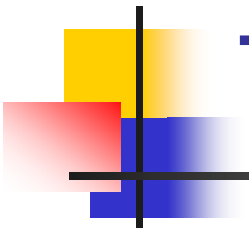
XP Practices: Simple Design

- Start with a simple design and keep it that way through design improvement
- Don’t make code unnecessarily general
- No wasted effort – design suited for current functionality



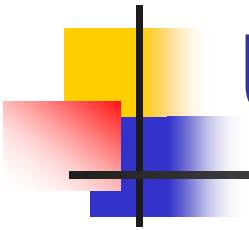
XP Practices: Design Improvement

- Continuous process of design improvement called “Refactoring”
- Implementation of code changed without altering interface
- Remove duplication



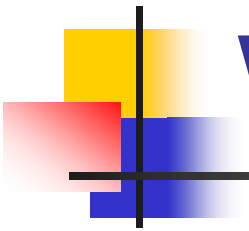
XP Practices: Test Driven Development

- Write test first, then write code to make it work
  - A form of Design by Contract
- Every test must pass at every build
  - Supports Continuous Integration



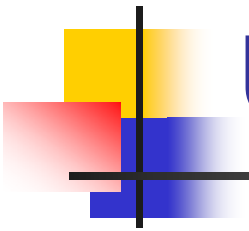
Unit Testing

- Based on the idea that classes should contain their own tests
- Highly localized; test(s) work within a single package
- Tests the interfaces to other packages, but just assumes other packages work



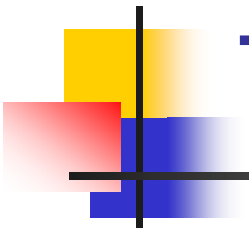
Why Unit Testing?

- Better able to exercise all code
- Can write tests before writing code:
  - Helps programmer to focus on the interface rather than the implementation
  - Provides a clear finish point: when the test works
- Cuts down significantly on debugging time
  - Run tests every time code is compiled
  - If new code breaks a previously-passed test, bug location is easier to pinpoint



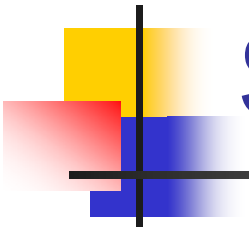
Unit Testing Difficulties

- Detraction: seem to be writing code twice
- Many programmers have never learned to write tests or even to think about tests
- Overhead of test framework



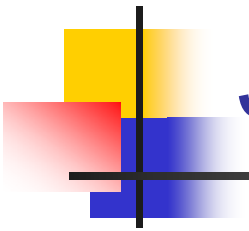
The Junit Testing Framework

- Used for writing unit tests in Java
- Helps automate testing process
- Provides some basic constructs for running tests



Structure

- Any class that contains tests must subclass the TestCase class
  - Typically one for each class being tested
- Junit framework allows tests to be grouped into suites
  - TestSuites can contain TestCases or other TestSuites
  - Makes it easy to build a range of large test suites and run the tests automatically



Junit Example: I/O Class

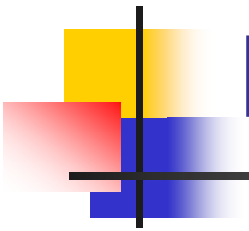
- The test must have a constructor:

```
class FileReaderTester extends TestCase {
    public FileReaderTester (String name) {
        super(name);
    }
}
```

First step: Set up a test fixture

- A *test fixture* is the set of objects that act as samples for testing. In the case of I/O testing, a test file: data.txt

Bradman	99.94	52	80	10	6996	334	29
Pollock	60.97	23	41	4	2256	274	7
Headley	60.83	22	40	4	2256	270*	10
Sutcliffe	60.73	54	84	9	4555	194	16

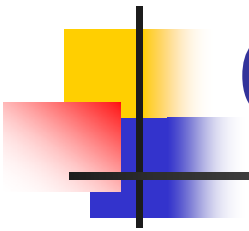


Manipulating the test fixture

- TestCase provides:
  - protected void setUp() – creates objects
  - protected void tearDown() – removes them
- Important to execute both methods for each test so that the tests are isolated from each other; thus can run them in any order

setUp & tearDown

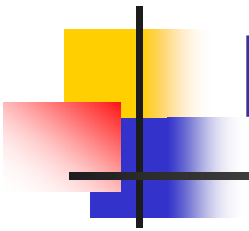
```
class FileReaderTester…
    protected void setUp() {
        try {
            _input = new FileReader(“data.txt”);
        } catch (FileNotFoundException e) {
            throw new RuntimeException (“unable to open test file”);
        }
    }
    protected void tearDown(){
        try {
            _input.close();
        } catch (IOException e) {
            throw new RuntimeException (“error on closing test file”);
        }
}
```



Create the first test

```
public void testRead() throws IOException {
    char ch;
    for (int i = 0; i < 4; i++) {
            ch = (char) _input.read();
    }
    Assert.assertEquals('d', ch);
}
```

- `assertEquals` is the automatic Junit test

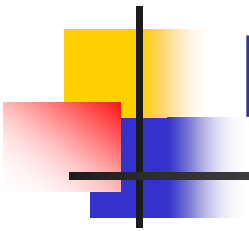


How to run the test

- Create a test suite:

```
class FileReaderTester ...
    public static Test suite(){
        TestSuite suite = new TestSuite();
        suite.addTest(new FileReaderTester("testRead"));
        return suite;
    }
```

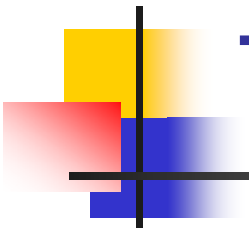
- The test is bound to the method testRead()



How to run the test (cont'd)

- Use a separate TestRunner class
  - can use a GUI version, but character interface can be called within the code

```
class FileReaderTester …
    public static void main(String[] args) {
            junit.textui.TestRunner.run(suite());
    }
```



TestRunner success output

.

Time: 0.110

OK (1 tests)

- Junit prints a period (“.”) for every test run
- Junit prints a single “OK” if no test fails

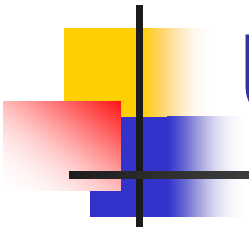
TestRunner failure output

```
public void testRead() throws
    IOException
{
    char ch;
    for (int i = 0; i < 4; i++) {
        ch = (char) _input.read();
    }
    Assert.assertEquals('2', ch);
    // deliberate error
}
```

Result:

```
.F
Time: 0.220

FAILURES!!!
Test Results:
Run: 1 Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead
"expected: "2" but was: "d"
```



Usefulness of Failures

- Can start by making tests fail, to prove:
  - the test does actually run
  - the test is actually testing what it's supposed to
- A common testing error is to be testing something other than what is supposed to be tested

Catching errors

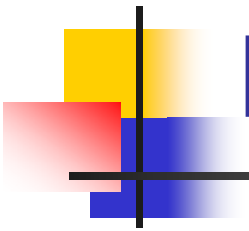
- In addition to catching failures (assertions are false), Junit's framework also catches errors (unexpected exceptions)

```
public void testRead() throws IOException
 {
            char ch;
            _input.close();
            for (int i = 0; i < 4; i++)
            {
               ch = (char) _input.read();
               // will throw exception
            }
            Assert.assertEquals('d', ch);
}
```

Result:

```
.E
Time: 0.110

!!!FAILURES!!!
Test Results:
Run: 1 Failures: 0 Errors: 1
There was 1 error:
1) FileReaderTester.testRead
java.io.IOException: Stream closed
```



Running multiple tests

- Write new test methods
  - public void testReadAtEnd()
- Put them in the suite to run them:
  - suite.addTest(new FileReaderTester("testReadAtEnd"));
- Junit has a lazy-programmer shortcut:
  - Naming convention: "testX()"
  - Replace main() method with:

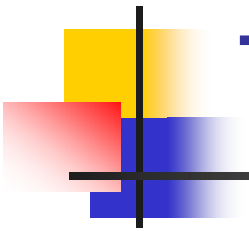
```
public static void main(String[] args) {
   junit.textui.TestRunner.run(new TestSuite(FileReaderTester.class));
}
```

Can run a Master Test Suite

```
class MasterTester extends TestCase {
    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
    }
    public static Test suite() {
        TestSuite result = new TestSuite();
        result.addTest(new TestSuite(FileReaderTester.class));
        result.addTest(new TestSuite(FileWriterTester.class));
        // and so on…
        return result;
    }
}
```

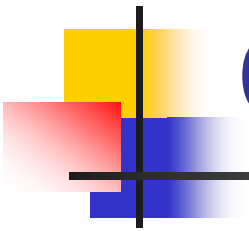
User-defined comments in Junit

```
public void testReadBoundaries() throws IOException {
    assertEquals(“read first char”, ‘B’, _input.read());
    char ch;
    for (int i = 1; i < 140; i++){
            ch = _input.read();
    }
    Assert.assertEquals(“read last char”, ‘6’, _input.read());
    Assert.assertEquals(“read at end”, -1, _input.read());
}
```



Testing philosophies

- Testing should be risk-driven
  - “test every public method” is not enough
- A little testing goes a long way
  - Better to focus on complex code and areas that are at most risk of going wrong
  - Helps to keep the task of test-writing to a doable size
- Focus on boundary conditions and special conditions that make the test fail
  - e.g. for an I/O class, an empty file



Conclusions

- Cannot prove a program has no bugs by testing
- Tests will not find every bug, but they will make it easier to find many bugs
- The process of writing tests sparks consideration of boundary and error conditions and helps with understanding interfaces