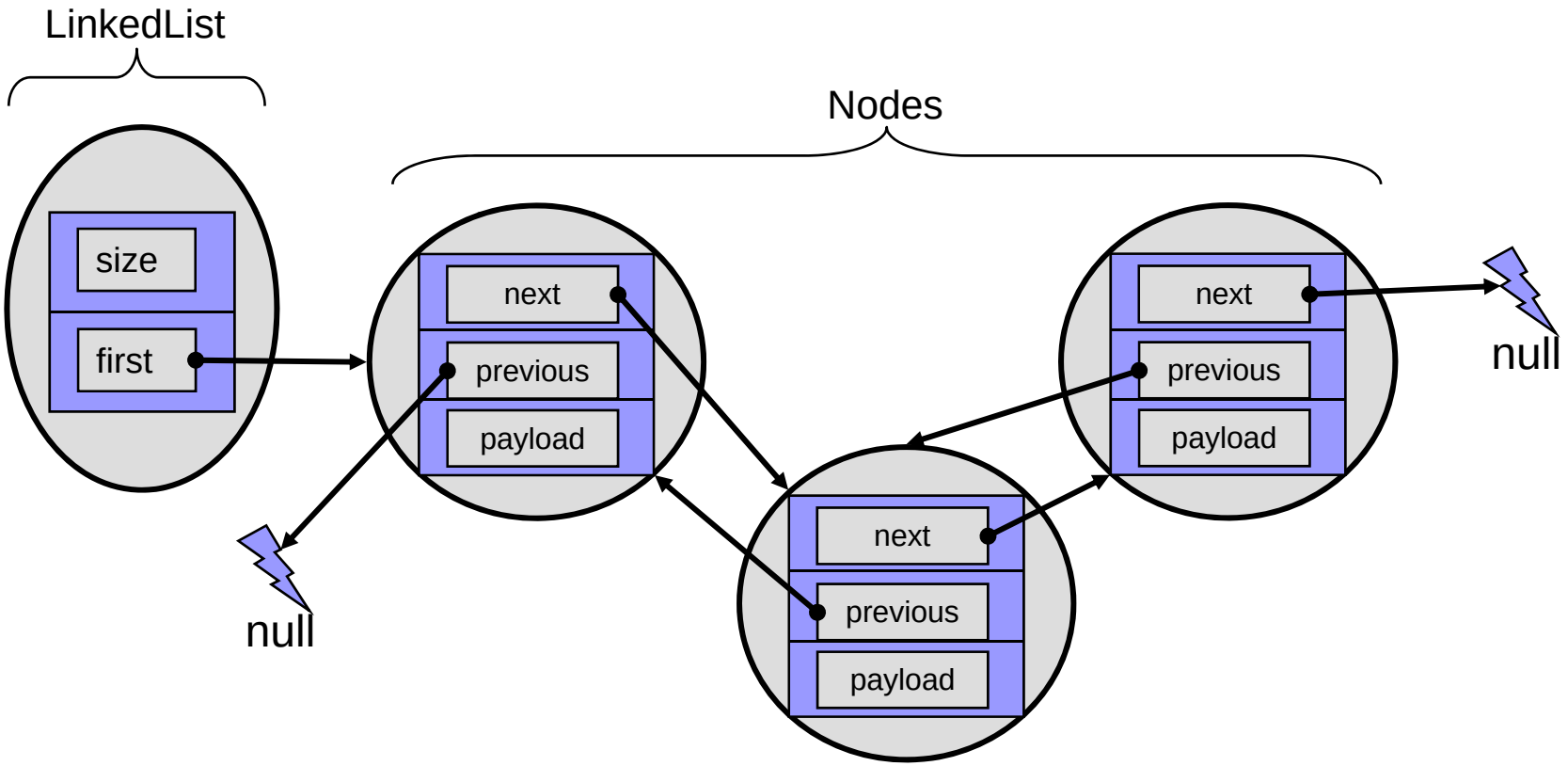
A Simple Example of TDD

garth@ggilmour.com

A Linked List Example

- Consider the design of a Linked List
  - Using objects and encapsulation
- Each value will be held in a ‘Node’ object
  - Each node has three private fields
    - A reference to the next node in the list
    - A reference to the previous node in the list
    - The payload to be kept for the list user
- A ‘LinkedList’ object will manage a chain of nodes
  - It will have a field which refers to the first Node in the list
  - When we need to add or remove nodes we ‘walk the list’
    - We follow each Nodes ‘next’ reference till we reach null

A Linked List Example



Making a Start…

- When we add a new class we also add a unit test class
  - An exception can be made for simple classes like JavaBeans

```
package demos.tdd;

public class LinkedLIst {

}
```

```
package demos.tdd;

public class LinkedListTest {

}
```

The First Test

```
package demos.tdd;

import static org.junit.Assert.*;
import org.junit.Test;

public class LinkedListTest {
	@Test
	public void newListShouldBeEmpty() {
		LinkedList list = new LinkedList();
		assertTrue(list.isEmpty());
		assertEquals(0,list.size());
	}
}
```

The First Implementation

```
package demos.tdd;

public class LinkedList {
	public boolean isEmpty() {
		return true;
	}
	public int size() {
		return 0;
	}
}
```

The Second Test

```
@Test
public void listWithItemsShouldNotBeEmpty() {
    LinkedList list = new LinkedList();
    list.add("abc");
    list.add("def");
    list.add("ghi");
    assertFalse(list.isEmpty());
    assertEquals(3,list.size());
}
```

The Second Implementation

```
package demos.tdd;

public class LinkedList {
	public boolean isEmpty() {
		return size == 0;
	}
	public int size() {
		return size;
	}
	public void add(String item) {
		size++;
	}
	private int size;
}
```

Refactoring the Unit Test

- Our implementation is very simple
  - But our test could do with some cleanup
- This is a good time to refactor the test
  - We can see lots of duplicated code
  - This will get worse as we add more tests
- It looks like every test will begin the same way
  - With ‘LinkedList list = new LinkedList()’
  - So lets move this to a method called before each test
    - The ‘list’ variable can become a field
- Many tests will require a populated list
  - So lets extract a method for loading test items

```
public class LinkedListTest {
	@Before
	public void start() {
		list = new LinkedList();
	}
	@Test
	public void newListShouldBeEmpty() {
		assertTrue(list.isEmpty());
		assertEquals(0,list.size());
	}
	@Test
	public void listWithItemsShouldNotBeEmpty() {
		addItems();
		assertFalse(list.isEmpty());
		assertEquals(3,list.size());
	}
	private void addItems() {
		list.add("abc");
		list.add("def");
		list.add("ghi");
	}
	private LinkedList list;
}
```

The Third Test

```
@Test
public void canRetrieveSingleItem() {
    list.add("xyz");
    assertEquals("xyz",list.get(0));
}
```

The Node (Obvious Code)

```
class Node {
        Node(Node next, Node prev, String item) {
                this.next = next;
                this.prev = prev;
                this.item = item;
        }
        Node getNext() { return next; }
        void setNext(Node next) { this.next = next; }}
        String getItem() { return item; }

        private Node next;
        private Node prev;
        private String item;
}
```

The Third Implementation

```
public void add(String item) {
    if(isEmpty()) {
        first = new Node(null,null,item);
    }
    size++;
}
public String get(int index) {
    if(index == 0) {
        return first.getItem();
    }
    return null;
}
private Node first;
```

The Fourth Test

```
@Test
public void canRetrieveMultipleItems() {
    addItems();
    assertEquals("abc",list.get(0));
    assertEquals("def",list.get(1));
    assertEquals("ghi",list.get(2));
}
```

The Fourth Implementation (1)

```
public void add(String item) {
    if(isEmpty()) {
        first = new Node(null,null,item);
    } else {
        Node current = first;
        while(current.getNext() != null) {
            current = current.getNext();
        }
        current.setNext(new Node(null,current,item));
    }
    size++;
}
```

The Fourth Implementation (2)

```
public String get(int index) {
    if(index == 0) {
        return first.getItem();
    } else {
        Node current = first;
        for(int i=0;i<index;i++) {
            current = current.getNext();
        }
        return current.getItem();
    }
}
```

Refactoring the Finished Code

- Our list now has full functionality
  - But our job isn't complete till we refactor
- The ‘LinkedList’ class can be tidied quite a bit
  - We can extract methods for:
    - Walking to the end of the list
    - Walking to a particular node
  - The complexity of the ‘get’ method can be reduced
    - See if you can work out how…
- The test class can be tidied a little
  - The ‘addItems’ method is too encapsulated
  - We can open it up cleanly using Java 5 features…

Refactoring the ‘add’ Method

```
public void add(String item) {
    if(isEmpty()) {
        first = new Node(null,null,item);
    } else {
        Node last = walkToEnd();
        last.setNext(new Node(null,last,item));
    }
    size++;
}
private Node walkToEnd() {
    Node current = first;
    while(current.getNext() != null) {
        current = current.getNext();
    }
    return current;
}
```

Refactoring the ‘get’ Method (1)

```
public String get(int index) {
    if(index == 0) {
        return first.getItem();
    } else {
        Node chosen = walkToNode(index);
        return chosen.getItem();
    }
}
private Node walkToNode(int index) {
    Node current = first;
    for(int i=0;i<index;i++) {
        current = current.getNext();
    }
    return current;
}
```

Refactoring the ‘get’ Method (2)

```
public String get(int index) {
        Node chosen = walkToNode(index);
        return chosen.getItem();
}
private Node walkToNode(int index) {
        Node current = first;
        for(int i=0;i<index;i++) {
                current = current.getNext();
        }
        return current;
}
```

Refactoring the Unit Test

```
@Test
public void listWithItemsShouldNotBeEmpty() {
    addItems("abc","def","ghi");
    //rest of method unchanged
}
@Test
public void canRetrieveSingleItem() {
    addItems("xyz");
    assertEquals("xyz",list.get(0));
}
@Test
public void canRetrieveMultipleItems() {
    addItems("abc","def","ghi");
    //rest of method unchanged
}
private void addItems(String ... items) {
    for(String item : items) { list.add(item); }
}
```

Taking Things Further…

- Which important tests have been omitted?
- What non-functional tests could we write?
- What extra functionality could be added?
- How could we make the design:
  - More flexible?
  - Better encapsulated?

Re-examining the Example

- The list example is slightly contrived
  - We deliberately chose a concept that is well suited to TDD
  - Collection classes are self-contained by design
- Think about unit testing in terms of M-V-C
  - Model classes are easy (possibly trivial) to unit test
  - Controller classes are accessible to unit testing
    - But we need some way to ‘fake out’ the classes they use
    - We will see how to do this later in the course…
  - View classes are hard (but not impossible) to test
    - A certain amount of creativity is required…

Introduction

The Origin of Agile Methods

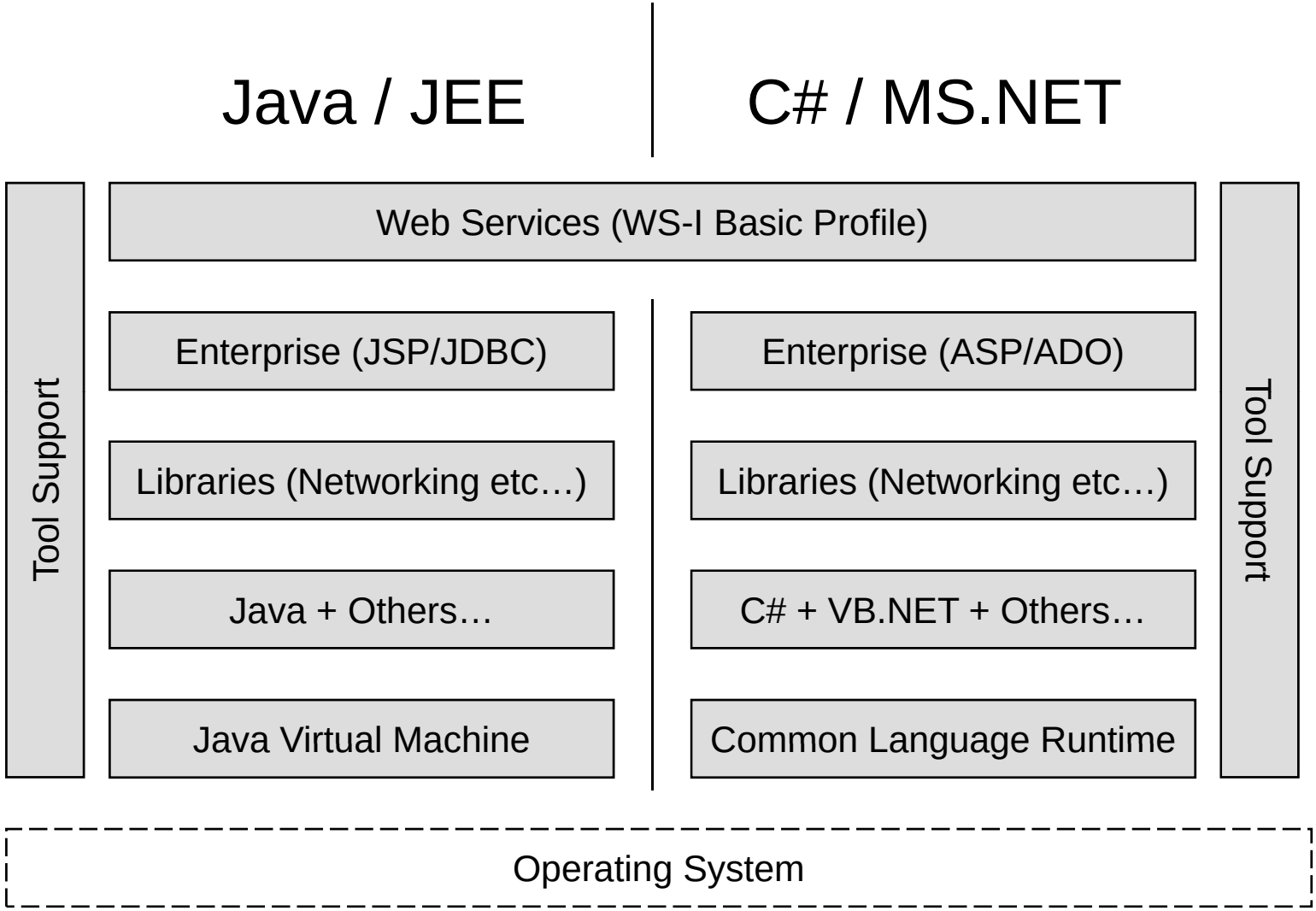
garth@ggilmour.com

The Evolution of Programming

- C/C++ builds were very fragile
  - Shared header files meant lengthy compilation times
  - Tracking dynamically allocated memory is difficult
  - Automation was based around Shell Scripts and Make
- Java/JEE builds are much more flexible
  - The VM and class file format minimizes build times
  - Garbage collection makes developers more productive
  - Tools like Ant and Maven are a vast improvement
  - Containers re-deploy components without a restart
- This gives developers the freedom to be flexible
  - Big up front design is an option rather than a necessity

The Evolution of Programming

- C/C++ programs used bespoke architectures
  - The language provided limited libraries which had to be supplemented with custom or 3rd party functionality
  - E.g. the range a variety of C++ string classes
- Java/C# programs are built on a much stronger base
  - Common libraries for GUI's, networking, security etc…
  - Vendor independent database access API's
  - Component based architectures for the web
- Java is particularly strong in the area of frameworks
  - The Java Persistence API for object-relational mapping
  - Spring for dependency injection and AOP



Testing in Different Eras

- Testing in the C/C++ era was manual and slow
  - Often a new release was made on a Friday evening and the test team spent Mon-Thurs evaluating it and producing reports
  - When a build of the system took 3 hours it was acceptable for a full test run to take 3 days and require dedicated staff
- Testing in the Java era must be automated and fast
  - Builds of the system are made in minutes not hours
  - JEE containers deploy components without a restart
  - We want to have a ‘big red button’ mentality
    - Push a button and a full build is made, deployed, tested and a detailed report published to the project website

How Software Processes Evolve

- Software processes emerged alongside languages
  - Unfortunately the evolution of software processes lags behind the development of new programming languages
- C/C++ era processes are ‘heavyweight’
  - They stress building and verifying models, up front design before coding and formal interface based architectures
- Processes developed around Java / C# are ‘Agile’
  - Also known as ‘lean’ or ‘lightweight’ processes
  - Designs are flexible and evolve during development
  - Iterations are brief and based on close contact with users

How Software Processes Evolve

> One information processing device that cannot easily accommodate itself to the universal acceleration (of change) is the human brain. Generals are always fighting the last war; and educators… are always instructing the last generation... The average college student is a very badly programmed computer.
>
> John Wilkinson (1964)

The Growth of Software Processes

Heavyweight Processes

- Designed for large projects
- Allocate staff to many roles
- Well defined stages / formal plans
- Lots of documentation and reviews

Lightweight Processes

- Designed for small projects
- Staff take on many roles
- Informal, rapidly changing plans
- Code and tests are the primary artifacts

Waterfall vs. Iterative Development

- Modern processes agree on at least one point
  - Waterfall development is an outdated concept
  - Despite the fact that many organizations still use it
- The waterfall method contains a fatal flaw
  - It ignores the fact that each step in the waterfall can only be validated by the activities that follow it
  - Oversights and assumptions in requirements become evident during design, which in turn is critiqued via coding
- The recommendation is that progress should be iterative
  - Instead of one big waterfall do lots of mini-ones (iterations)
  - Each mini-waterfall both implements new functionality and fixes the problems which were identified during the last one

Core Concepts

The Basic Theory of TDD

garth@ggilmour.com

Introducing TDD

- Software is tested in many ways
  - Functional tests make sure the requirements are met
  - Non functional tests make sure each function is Usable, Reliable, has good Performance and is Supportable
  - Acceptance tests represent the criteria set by customers
  - Load testing overwhelms the code with numerous concurrent requests, large volumes of data transfer and slow networks
  - Regression testing validates all the functionality added so far
- Unit tests are the most fundamental kind of test
  - Developers test their own code before they add it to the build

Undisciplined Unit Testing

- Unit testing is rarely systematic
  - Developers do just enough to satisfy themselves
  - This may be a little or a lot…
- Unit tests are mostly junked after use
  - Developers write small test applications (console or GUI based)
    - These serve as ad hoc testing frameworks
  - When these apps have served their purpose they are discarded
    - Because they are ad hoc code they have little reuse value
- This kind of unit testing is never regressive
  - The tests only check the new functionality
  - They fail to ensure that existing functionality is still OK

The Potential of Unit Testing

- Undisciplined unit testing ignores a potential source of massive improvements in code quality
  - The information lost by junking unit tests is priceless
- Systematic unit testing requires two things
  - We retain our unit tests and treat them as artefacts
  - There is a standard pattern for writing tests and a developer friendly set of tools for creating and running them
- Unit tests could then be integrated into our process
  - They can be combined into a test suite which will continuously verify the quality of our code by testing each class in isolation

Test Driven Development

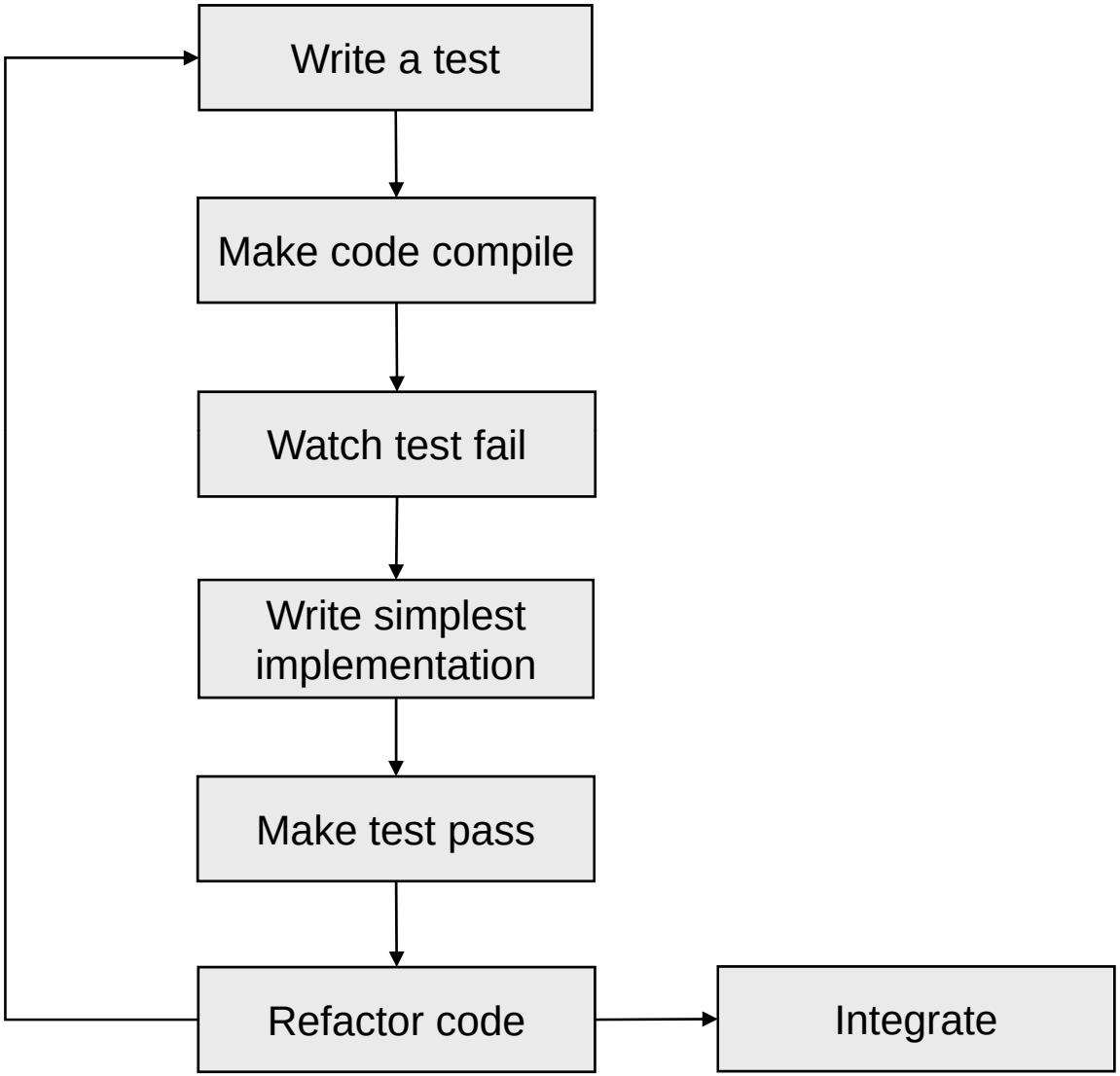
- TDD is disciplined unit testing
  - It provides a methodology, a toolset and a philosophy
- It adds an extra dimension to unit testing
  - The idea that the tests should be written before the code
  - We write tests to define functionality as much as to validate it
- Writing the tests before the code makes good sense
  - It forces you to write code from the clients perspective
    - Leading to simpler and more intuitive interfaces
  - It encourages the developer to work incrementally
    - Preventing hours of ‘heroic’ programming between tests
  - It encourages you to think about the essentials
    - How to get this test to pass without breaking any of the others

Test Driven Development

- Communicating via test cases also has benefits
  - When new requirements arrive write tests for them
  - When bugs are reported write tests and make them work
  - Use the test cases to learn how to use a particular class
  - Run the suite of tests to make changes with confidence
- Test cases give new developers a starting point
  - Arbitrarily browsing around a new system is inefficient
    - But this is how new developers typically learn a system
  - UML diagrams can be used to discover the major classes, which can then be dissected via the test cases

The TDD Development Cycle

1. Write a test for the new requirement
   - Each test should be as fine grained as possible
   - Don't be afraid to make the tests very simple
2. Make the test work
   - Write just enough code to pass the test
   - This usually leads to a 'but it still wont work for…'
3. Write further tests until the code works right
   - Until you can no longer think of useful tests to add
4. Refactor the code to keep the design simple
   - We will discuss refactoring later
5. Integrate your work with the main build



The TDD Development Cycle

- Three critical points to note are:
  - Celebrate the ‘but that’s stupid’ moments!
    - This is what keeps us going the right way
  - Always watch the test fail first
    - Sometimes your test will run without adding code…
  - Don’t implement and refactor at the same time
    - They are two separate activities
    - Mixing them only causes confusion
    - Remember tests need refactoring as well

The TDD Development Cycle

- We do a ‘check-in’ every time we add functionality
  - Not when we have implemented a complete requirement
- Up front design is discouraged
  - We always do the simplest thing that could possibly work
  - All code is refactored before being checked in
- The system is complete when it passes all the tests
  - If functionality is absent or incomplete write more tests…
- Test cases also need to be maintained
  - To avoid duplicated code and functionality
  - To introduce common ways of loading test data

Keeping a Notebook

- Many possible tests will occur to you as you code
  - Its important not to get distracted by tangential issues
    - Stick with the simplest possible solution
  - But you should always have a notebook handy
    - To jot down ideas for tests, enhancements etc…
- Sometimes multiple tests help scope out the next step
  - This is known as ‘triangulation’
- If you are doing TDD right it can become boring
  - Sticking to the simplest solution instead of exploring alternatives
  - Some projects like to leave an ‘escape route’ for developers
    - A period of time set aside specifically for exploring radical ideas

Limitations of TDD

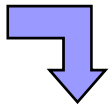
- TDD does not replace the QA department
  - It does not guarantee that components, layers, subsystems etc… will operate correctly when they are assembled
- However it does simplify the work of the QA team
  - The QA team will not waste time discovering low level bugs
  - Developers should be more willing to involve themselves in QA
  - TDD provides a foundation on which it is possible to build a rigorous set of acceptance tests using feature rich testing tools
- TDD is hard work to maintain under pressure
  - In any software process code quality is the most tempting thing to sacrifice in a crisis and the hardest thing to regain afterwards

Creative Use of Java Libraries

- Many things that don't appear testable actually are
  - The flexibility of the J2SE libraries is very helpful
- Take the example of drawing a pyramid:
  - Refactor the drawing code into a 'printPyramid' method
    - The method should take a stream parameter
    - This breaks the hardwired link to 'System.out'
  - Create a stream which writes into a byte array
  - Pass the stream into the 'printPyramid' method
  - Build a String object from the byte array
  - Check that the String contains the correct output

```
private static void main(String [] args) {
        int height = readHeightFromConsole();
        System.out.println("A pyramid of size " + height);
        int spaces = height - 1;
        int hashes = 1;

        while(spaces >= 0) {
                for(int x=0;x<spaces;x++) {
                        System.out.print(" ");
                }
                for(int x=0;x<hashes;x++) {
                        System.out.print("#");
                }
                System.out.println();
                spaces--;
                hashes+=2;
        }
}
```



```
private static void printPyramid(PrintStream output, int height) {
        int numHashes = 1;
        for (int rows = 1; rows <= height; rows++) {
                int spaces = height - rows;
                while (spaces > 0) {
                        output.print(" ");
                        spaces--;
                }
                for (int hashes=0; hashes<numHashes; hashes++) {
                        output.print("#");
                }
                output.println();
                numHashes += 2;
        }
}
```

```
public static void main(String[] args) {
        int height = readHeightFromConsole();
        System.out.println(“A pyramid of size " + height);
        printPyramid(System.out,height);
}
```

```
public class DrawPyramidTest extends TestCase {

        public void testPyramid() throws IOException {
                ByteArrayOutputStream out = new ByteArrayOutputStream();
                PrintStream ps = new PrintStream(out);
                DrawPyramid.printPyramid(ps,6);
                String pyramid = new String(out.toByteArray());
                BufferedReader br = new BufferedReader(new StringReader(pyramid));
                assertEquals("Bad Row1","     #",br.readLine());
                assertEquals("Bad Row2","    ###",br.readLine());
                assertEquals("Bad Row3","   #####",br.readLine());
                assertEquals("Bad Row4","  #######",br.readLine());
                assertEquals("Bad Row5"," #########",br.readLine());
                assertEquals("Bad Row6","###########",br.readLine());
                assertNull("End Not Found",br.readLine());
    }
}
```