SOFT 3416
Software Verification and Validation
Week IX

Ahmet Feyzi Ateş

Işık University Faculty of Engineering and Natural Sciences
Department of Computer Science Engineering
2023-2024 Spring Semester

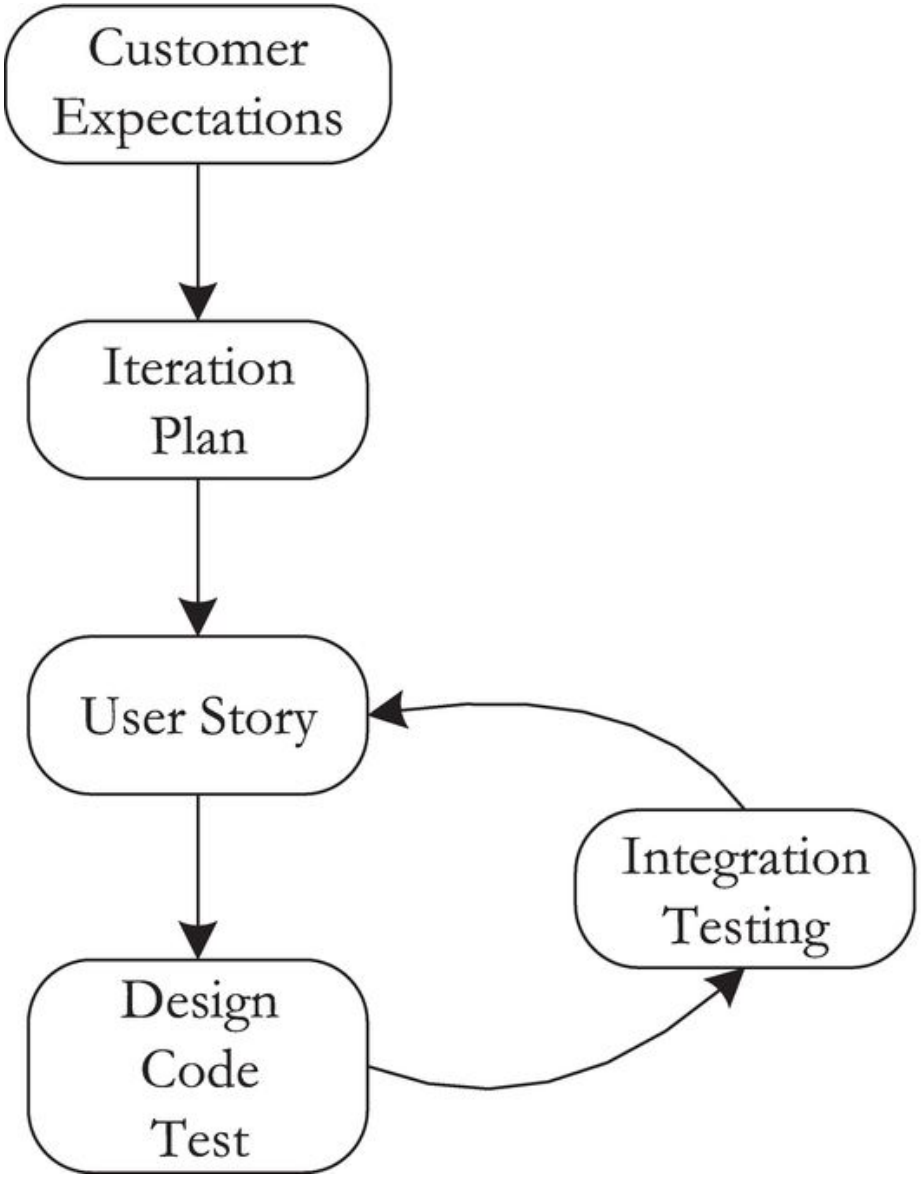
Agile Testing

In 2001 a group of well-known names in the software industry representing the most widely used software development methodologies came up with a solution which they called ***Agile***.

Agreed on a common set of values and principles which became known as the
manifesto for agile software development or the agile manifesto.

Individuals and interactions over
processes and tools
Working software over
comprehensive
documentation
Customer collaboration over
contract negotiation
Responding to change over
following a plan

Generic Agile Life Cycle



Features of Agile Development

- Agile development has no single development methodology or process,
- All Agile development processes promote iterative and incremental development, with significant testing.
- Agile is customer-centric and welcomes change during the process.
- Agile relies on customer involvement, mandates significant testing, and have short, iterative development cycles.

Agile Testing

- Agile testing is a form of collaborative testing, in that everyone is involved in the process through design, implementation, and execution of the test plan.
- Customers are involved in defining acceptance tests by defining use cases and program attributes.
- Developers collaborate with testers to build test harnesses that can test functionality automatically.
- Agile testing requires that everyone be engaged in the test process, which requires a lot of communication and collaboration.

Agile Testing (cont'd)

- As with most aspects of Agile development, Agile testing necessitates engaging the customer as early as possible and throughout the development cycle.

- Once developers produce a stable code base,customers should begin acceptance testing and provide feedback to the development team.

- It also means that testing is not a phase; rather, it is integrated with development efforts to compel continuous progress.

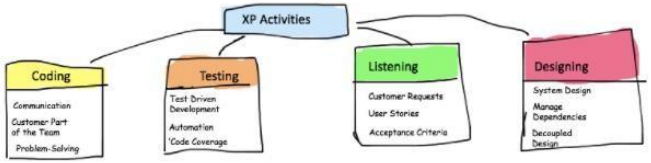
- Developers generally begin by writing unit tests first, then move to coding software units.

Agile Testing (cont'd)

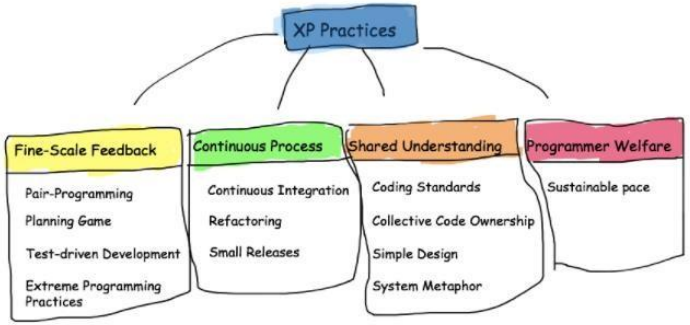
- To facilitate the timely feedback needed for rapid development, Agile testing relies on automated testing.
- Development cycles are short, so time is valuable, and automated testing is more reliable than manual testing approaches.
- Agile development environments often comprise only small teams of developers, who also act as testers.
- Larger projects with more resources may include an individual tester or a testing group.

- Testers job is to move the project forward by providing feedback about the quality of the software so that;
  - developers can implement bug fixes, and
  - make requirement changes and general improvements.

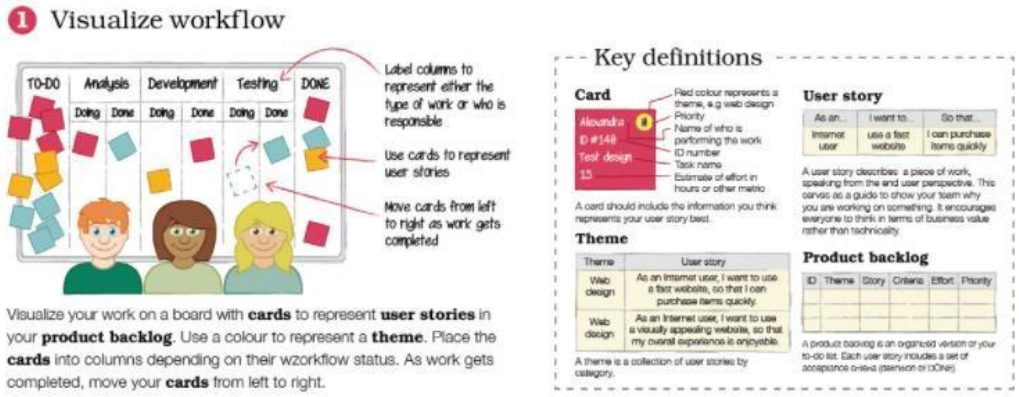
Agile Approaches



- Extreme Programming (XP)

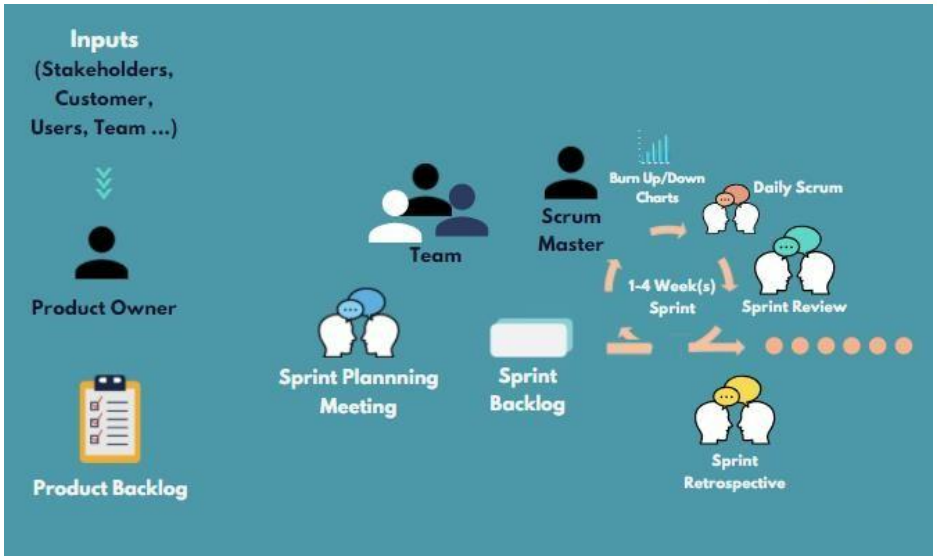


- Kanban

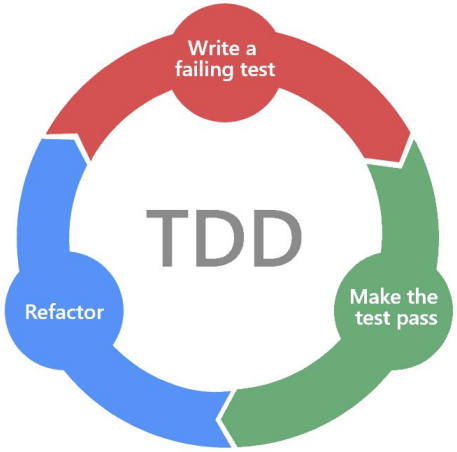


Agile Approaches (cont'd)

- Scrum



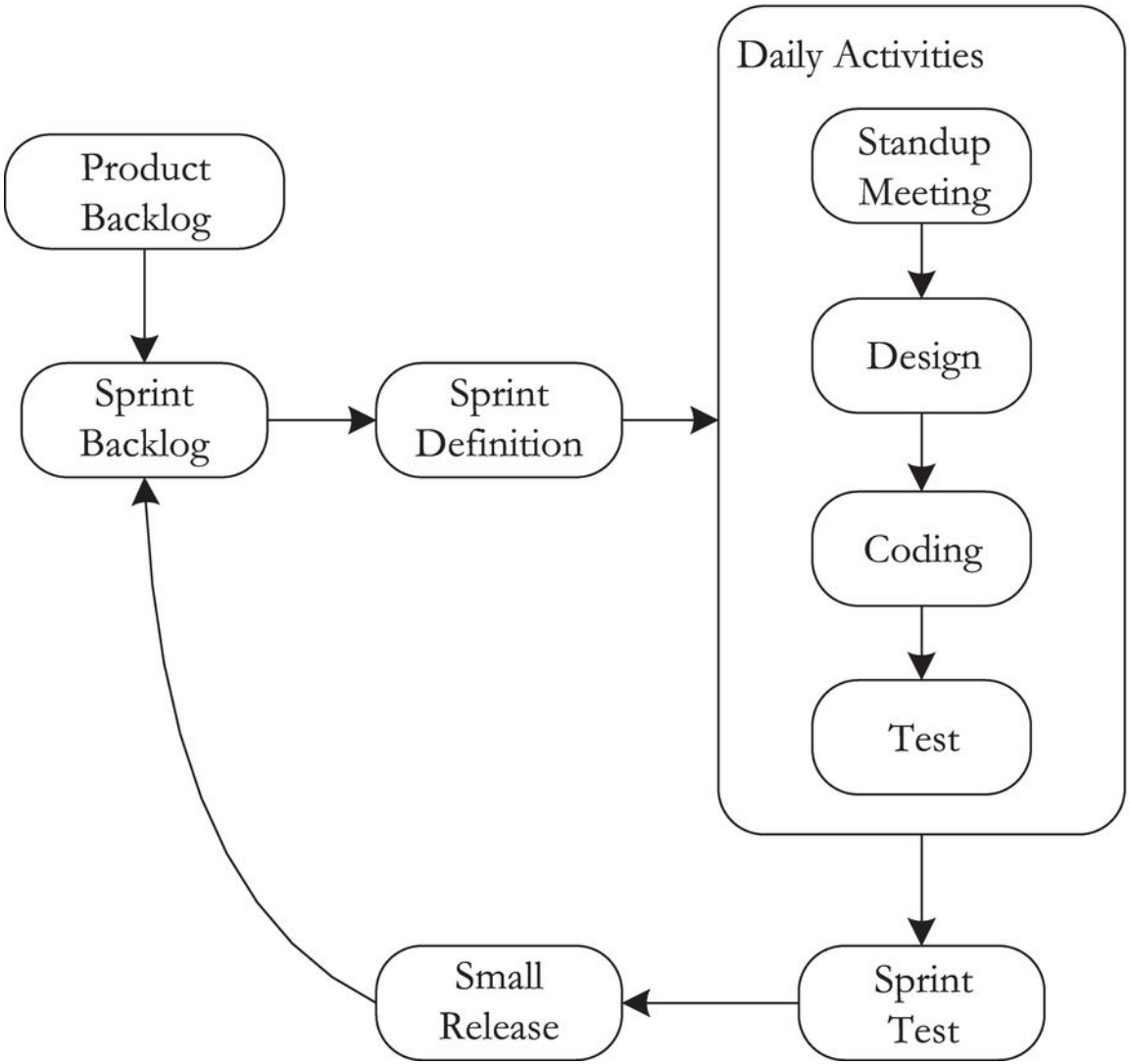
- Test Driven Development (TDD)



Scrum

- Scrum is probably the most frequently used of all the agile life-cycles.
- There is a pervading emphasis on the team members and teamwork in scrum.
- The name comes from the rugby maneuver in which the opposing teams are locked together and try to hook the football back to their respective sides.
- Scrum projects have
  - *Scrum Masters* (who act like traditional supervisors with less administrative power).
  - *Product Owners* are the customers of the traditional, and
  - the *Scrum Team* is a development team.

The Scrum Lifecycle



The Scrum Lifecycle (cont'd)

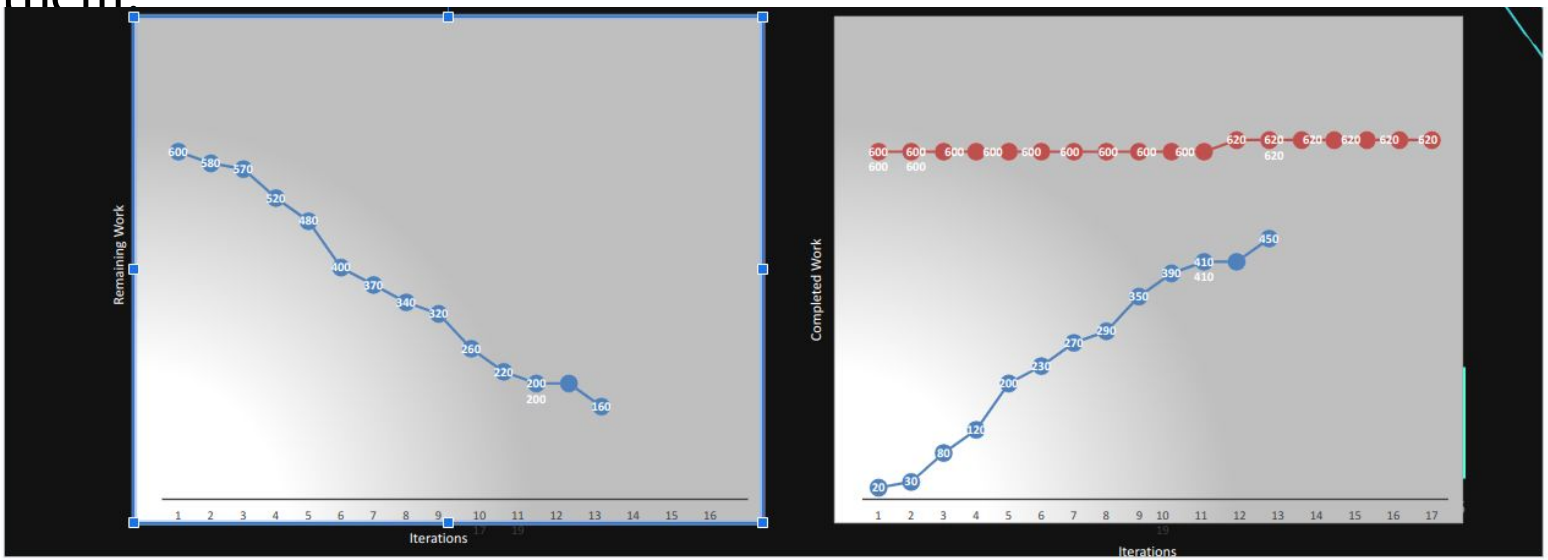
- The traditional iterations become "*sprints*" in scrum, which last from two to four weeks.
- In a sprint, there is a *daily stand-up meeting* of the Scrum Team to focus on what happened the preceding day and what needs to be done in the new day.
- Then there is a short burst of *design-code-test* followed by an integration of the team's work at the end of the day.
  - This is the agile part—a daily build that contributes to a sprint-level work product in a short interval.
- The biggest differences between Scrum and the traditional view of iterative development are the special vocabulary and the duration of the iterations.

Testing in Scrum

- Testing in the Scrum lifecycle occurs at two levels;
  - the unit level at each day's end, and
  - the integration/system level of the small release at the end of a sprint.
- Selection of the *Sprint backlog* (to do list of items) from the product backlog is done by the Product Owner (the customer), which corresponds roughly to a requirements step.
- Sprint definition looks a lot like preliminary design because this is the point where the Scrum Team identifies the sequence and contents of individual sprints.
- The small release is a deliverable product usable by the Product Owner, so it is clearly a *system* level work product;
  - Hence necessitates integration/system testing.

Testing in Scrum (cont'd)

- **BurnDown/BurnUp Charts**, are used to track and report the delivery team's progress towards completing their Sprint work.
- These charts are simple and effective visual representations to communicate the status of a Sprint.
- Normally, an Agile tool will help create these charts automatically, otherwise the Scrum Master is responsible for creating and maintaining them.



- **BurnDown Charts** are graphs indicating the number of remaining work items against the number of iterations
- **BurnUp Charts**, are graphs indicating the number of completed work items against the number of iterations

The Definition of Done in Scrum

- With regard to user stories:
  - The user stories selected for the iteration are complete, understood by the team, and have detailed, testable acceptance criteria
  - All the elements of the user story are specified and reviewed, including the user story acceptance tests, have been completed
  - Tasks necessary to implement and test the selected user stories have been identified and estimated by the team

The Definition of Done in Scrum (cont'd)

- With regard to features:
  - All essential user stories, with acceptance criteria, are defined and approved by the customer
  - The code is complete, with no known technical debt or unfinished refactoring
  - Unit tests have been performed and have achieved the defined level of coverage
  - Integration tests and system tests for the feature have been performed according to the defined coverage criteria
  - No major defects remain to be corrected
  - Feature documentation is complete, which may include release notes, user manuals, and on-line help function.

The Definition of Done in Scrum (cont'd)

- With regard to iterations
  - All features for the iteration are ready and individually tested according to the feature level criteria.
  - Any non-critical defects that cannot be fixed within the constraints of the iteration added to the product backlog and prioritized.
  - Integration of all features for the iteration completed and tested.
  - Documentation written, reviewed, and approved

Extreme Programming (XP)

- XP is a software process that helps developers create high quality code,

- Here, ‘*‘quality”* is defined as a code base that meets the design specification and customer expectation.

- XP focuses on:
  - Implementing simple designs.
  - Communicating between developers and customers.
  - Continually testing the code base.
  - Refactoring, to accommodate specification changes.
  - Seeking customer feedback.

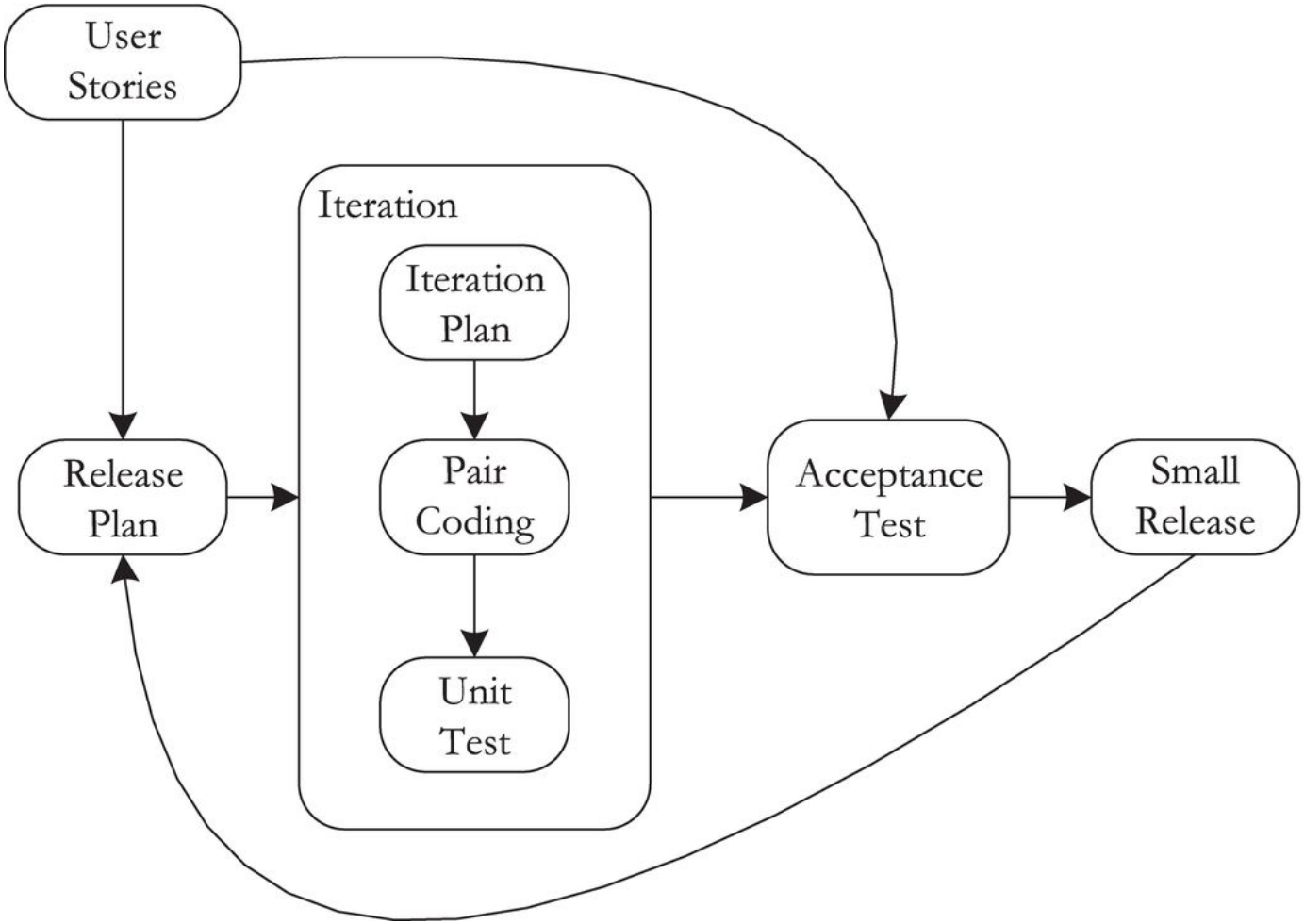
Extreme Programming (cont'd)

- Agile testing fits well into the Extreme Programming methodology whereby developers create unit tests first, then the software.
- The XP development methodology facilitates the creation of quality programs in short time frames.
- Besides customer involvement, the XP model relies heavily on unit and acceptance testing.
- In general, developers run unit tests for every incremental code change, no matter how small, to ensure that the code base still meets its specification.

Extreme Programming (cont'd)

- The primary difference of XP compared to traditional methodologies is its approach to testing.
- Traditional software development models suggest you code first and create testing interfaces later.
- Testing is of such importance in XP that the process requires you to create the unit (module) and acceptance tests first, then your code base.
- In XP, you must create the unit tests first, and then write the code to pass the tests.

The Extreme Programming Lifecycle



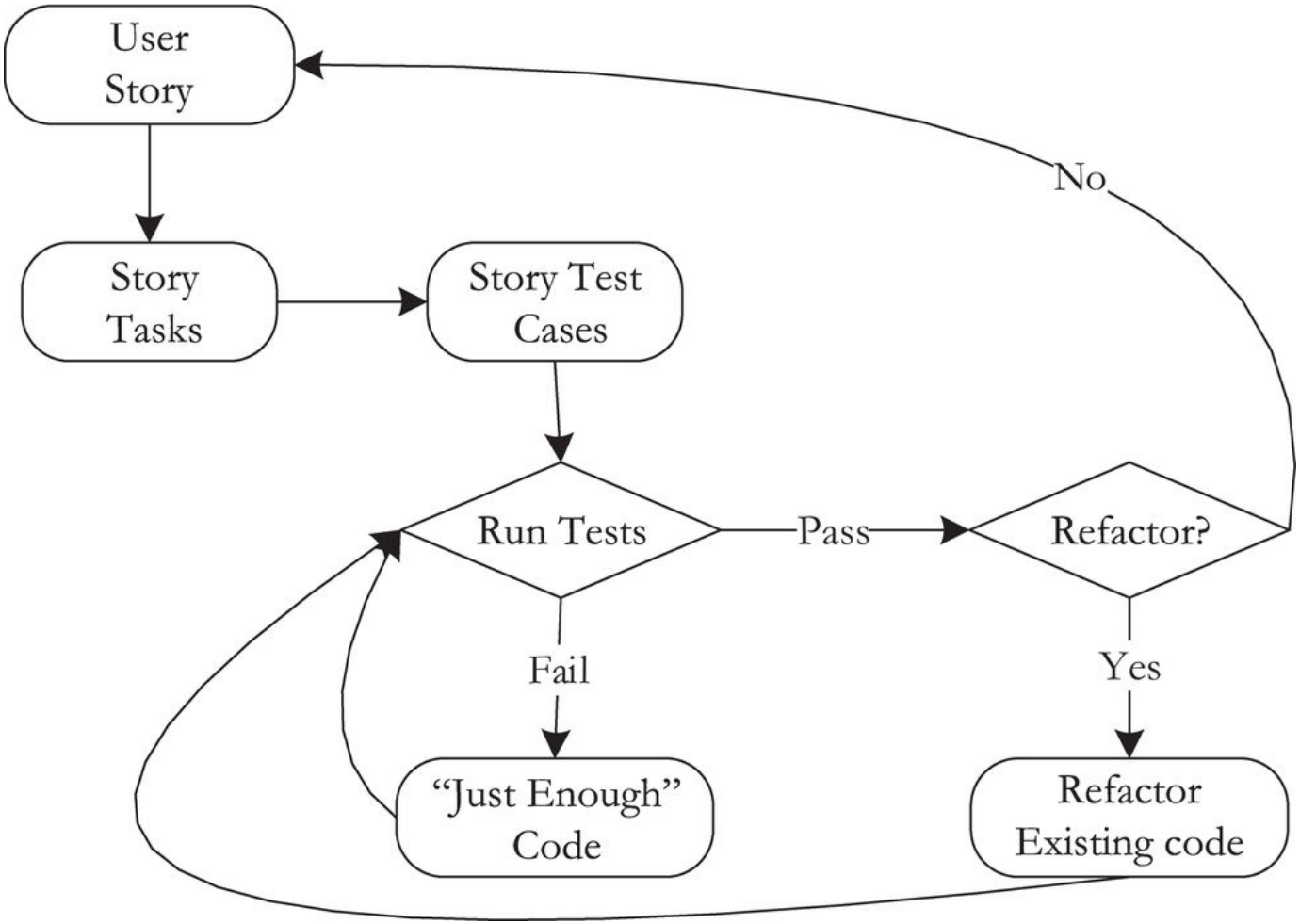
Example XP Project Flow

1. Programmers meet with the customer to determine the product requirements and build user stories.
2. Programmers meet without the customer to divide the requirements into independent tasks and estimate the time to complete each task.
3. Programmers present the customer with the task list and with time estimates, and ask them to generate a priority list of features.
4. The programming team assigns tasks to pairs of programmers, based on their skill sets.
5. Each pair creates unit tests for their programming task using the application's specification.
6. Each pair works on their task with the goal of creating a code base that passes the unit tests.
7. Each pair fixes, then retests their code until all unit tests have passed.
8. All pairs gather every day to integrate their code bases.
9. The team releases a pre-production version of the application.
10. Customers run acceptance tests and either approve the application or produce a report identifying the bugs/deficiencies.
11. Upon successful acceptance tests, programmers release a version into production.
12. Programmers update time estimates based on latest experience.

Test-Driven Development (TDD)

- Test-Driven Development (TDD) is the extreme case of agility.
- It is a practice that comes from Extreme Programming.
  - Tests drive the code
- It is driven by a sequence of user stories.
  - Each test shall cover a single scenario for a single piece of logic
- A user story can be decomposed into several tasks, and this is where the big difference occurs.
- Before any code is written for a task, the developer decides how it will be tested.
  - The tests become the specification.
- The tests are run on non-existent code.
- Naturally, those tests fail, as expected; but this leads to the best feature of TDD:
  - greatly simplified fault isolation.

Test-Driven Development Lifecycle



Test Driven Development (TDD)

- Once the tests have been run (and failed), the developer writes just enough code to make the tests pass, and the tests are rerun.
- If any test fails, the developer goes back to the code and makes a necessary change.
- Once all the tests pass, the next user story is implemented.
- Occasionally, the developer may decide to refactor the existing code.
- The cleaned-up code is then subjected to the full set of existing test cases, which is very close to the idea of regression testing.
- For TDD to be practical, it must be done in an environment that supports automated testing.

Benefits of TDD

- Remember: The unit tests must be defined and created before coding the module.
- Some valid concerns are:
  - how one can, create test drivers for code you haven't yet written.
  - one might not have time to create the tests, and still meet the project deadline.
- Such concerns can be compensated by a number of important benefits associated with TDD approach
  - You gain confidence that your code will meet its specification and requirements.
  - You express the end result before you start coding.
  - **You better understand the application's specification and requirements.**
  - You may implement simple designs initially and confidently refactor the code later to improve performance, without worrying about breaking the specification.

Example Application of TDD

- A command-line application that simply determines whether an input value is a prime number or not.
- We will examine both the source code for the application, and the tests.
- The specification of this program is as follows:

  - Develop a command-line application that accepts any positive integer, n, where 0 <= n <= 1000, and determine whether it is a prime number. If n is a prime number, then the application should return a message stating it is a prime number. If n is not a prime number, then the application should return a message stating it is not a prime number. If n is not a valid input, then the application should display a help message.

Example Application of TDD (cont'd)

- Following the TDD methodology and the principles we begin the application by designing unit tests.
- With this application, we can identify two discrete tasks: *validating inputs* and *determining prime numbers.*
- The TDD practices mandates the black-box approach, to eliminate any bias.
- We will use *boundary analysis* to validate the inputs because this application can only accept positive integers within a certain range.
- All other input values, including character datatypes and negative numbers, should raise an error and not be used.
- The important thing here is to identify, and commit to, a testing approach when designing your tests.

Example Application of TDD (cont'd)

- The next step is to develop a list of test cases based on possible inputs and expected outcome.

Case Number	Input	Expected Output	Comments
1	n = 3	Affirm n is a prime number.	Tests for a valid prime number.
2	n = 1000	Affirm n is not a prime number.	Tests input equal to upper bounds. Tests whether n is an invalid prime.
3	n = 0	Affirm n is not a prime number.	Tests input equal to lower bounds.
4	n = -1	Print help message.	Tests input below lower bounds.
5	n = 1001	Print help message.	Tests input greater than the upper bounds.
6	n = "a"	Print help message.	Tests input is an integer and not a character datatype.
7	Two or more inputs	Print help message.	Tests for correct number of input values.
8	n is empty (blank)	Print help message.	Tests whether an input value is supplied.

Example Application of TDD (cont'd)

- Test case 1 combines two test scenarios. It checks whether the input is a valid prime and how the application behaves with a valid input value. We might use any valid prime in this test.
- We also test two scenarios with test case 2: What happens when the input value is equal to the upper bounds and when the input is not a prime number?
- This case could have been broken out into two unit tests, but one goal of software testing in general is to minimize the number of test cases, while still adequately checking for error conditions.
- Test case 3 checks the lower boundary of valid inputs.
- Test cases 4 and 5 ensure that the inputs are within the defined range, which is greater than or equal to 0 and less than or equal to 1,000.

Example Application of TDD (cont'd)

- Case 6 tests whether the application properly handles character input values. Because we are doing a calculation, it is obvious that the application should reject character datatypes.

- The assumption with this test case is that the programming language will handle the datatype check.

- This application must handle the exception raised when an invalid datatype is supplied. This test will ensure that the exception is thrown.

- Lastly, tests 7 and 8 check for the correct number of input values; any number of inputs other than 1 should fail.

Test Driver and Application

- **JUnit** is a freely available open-source tool used to automate unit tests of Java applications.
- **JUnit** is very small, but very flexible and feature rich.
- One can create individual tests or a suite of tests. You can automatically generate reports detailing the errors.
- Now that we have designed all test cases, we can create the test driver class.

```
//check4PrimeTest.java
import junit.framework.*;

public class check4PrimeTest extends TestCase{
    //Initialize a class to work with.
    private check4Prime check4prime = new check4Prime();

    //constructor
    public check4PrimeTest (String name){
        super(name);
    }

    //Main entry point
    public static void main(String[] args) {
        System.out.println("Starting test...");
        junit.textui.TestRunner.run(suite());
        System.out.println("Test finished...");
    } // end main()

    //Test case 1
    public void testCheckPrime_true(){
        assertTrue(check4prime.primeCheck(3));
    }

    //Test cases 2,3
    public void testCheckPrime_false(){
        assertFalse(check4prime.primeCheck(0));
        assertFalse(check4prime.primeCheck(1000));
    }

    //Test case 4
    public void testCheck4Prime_checkArgs_neg_input(){
        try {
            String [] args= new String[1];
            args[0]="-1";
            check4prime.checkArgs(args);
            fail("Should raise an Exception.");
        } catch (Exception success){
            //successfull test
       }
    }// end testCheck4Prime_checkArgs_neg_input()
```

```
//Test case 5
public void testCheck4Prime_checkArgs_above_upper_bound(){
    try {
        String [] args= new String[1];
        args[0]="1001";
        check4prime.checkArgs(args);
        fail("Should raise an Exception.");
    } catch (Exception success){
    //successfull test
    }
} // end testCheck4Prime_checkArgs_upper_bound()

//Test case 6
public void testCheck4Prime_checkArgs_2_inputs(){
    try {
        String [] args= new String[2];
        args[0]="5";
        args[1]="99";
        check4prime.checkArgs(args);
        fail("Should raise an Exception.");
    } catch (Exception success){
        //successfull test
    }
} // end testCheck4Prime_checkArgs_2_inputs

//Test case 7
public void testCheck4Prime_checkArgs_char_input(){
    try {
        String [] args= new String[1];
        args[0]="r";
        check4prime.checkArgs(args);
        fail("Should raise an Exception.");
    } catch (Exception success){
        //successfull test
    }
} //end testCheck4Prime_checkArgs_char_input()
//Test case 8
public void testCheck4Prime_checkArgs_0_inputs(){
    try {
        String [] args= new String[0];
        check4prime.checkArgs(args);
        fail("Should raise an Exception.");
    } catch (Exception success){
        //successfull test
    }
} // end testCheck4Prime_checkArgs_0_inputs
```

```
    //JUnit required method.
    public static Test suite() {
        TestSuite suite = new TestSuite(check4PrimeTest.class);
        return suite;
    }//end suite()

} //end check4PrimeTest
```

- Table below maps the JUnit methods in ***check4PrimeTest*** to the test cases covered.

Methods	**Test Case(s) examined**
testCheckPrime_true()	1
testCheckPrime_false()	2, 3
testCheck4Prime_checkArgs_char_input()	6
testCheck4Prime_checkArgs_above_upper_bound()	5
testCheck4Prime_checkArgs_neg_input()	4
testCheck4Prime_checkArgs_2_inputs()	7
testCheck4Prime_checkArgs_0_inputs()	8

Test Driver and Application

- The ***testCheckPrime_false()*** method tests two conditions, because the boundary values are not prime numbers.
- JUnit method from the ***check4JavaTest*** class listed above:

```
public void testCheckPrime_false(){
        assertFalse(check4prime.primeCheck(0));
        assertFalse(check4prime.primeCheck(1000));
}
```

- The JUnit method, ***assertFalse()***, checks to see whether the parameter supplied causes the method to return a false Boolean value.
- If false is returned, the test is considered a success.

Test Driver and Application (cont'd)

- The test code snippet also demonstrates one of the benefits of creating test cases and test harnesses first.
- Notice that the parameter in the ***assertFalse()*** method is another method, ***check4prime.primeCheck(n).***
- This method will reside in a class of the application.
- Creating the test harness first forced us to think about the structure of the application itself.
- In some respects, the application is designed to support the test harness.
- Here we need a method to check whether the input is a prime number, so we shall include it in the application.

Test Driver and Application (cont'd)

- With the test harness complete, application coding can begin.
- Based on the program specification, test cases, and the test harness, the resultant Java application will consist of a single class, ***check4Prime***, with the following definition:

```
public class check4Prime {
        public static void main (String [] args)
        public void checkArgs(String [] args) throws Exception
        public boolean primeCheck (int num)
}
```

Test Driver and Application (cont'd)

- As per Java requirements, the ***main()*** procedure provides the entry point into the application.
- The ***checkArgs()*** method asserts that the input value is a positive integer, n, where *0* <= *n* <= *1,000*.
- The ***primeCheck()*** procedure checks the input value against a calculated list of prime numbers.
- We shall implement a known algorithm to calculate the prime numbers.

The Application

```
//check4Prime.java
//Imports
import java.lang.*;

public class check4Prime {
    static final int max = 1000; // Set upper bounds.
    static final int min = 0; // Set lower bounds
    static int input =0; // Initialize input variable

    public static void main (String [] args) {
        //Initialize class object to work with
        check4Prime check = new check4Prime();
        try{
            //Check arguments and assign value to input variable
            check.checkArgs(args);
            //Check for Exception and display help
        }
        catch (Exception e){
            System.out.println("Usage: check4Prime x");
            System.out.println(" where 0<=x<=1000");
            System.exit(1);
        }

        //Check if input is a prime number
        if (check.primeCheck(input))
            System.out.println("Right... " + input + " is a prime number!");
        else
            System.out.println("Sorry... " + input + " is NOT a prime number!");
    } //End main
```

```
    //Calculates prime numbers and compares it to the input
    public boolean primeCheck (int num){
        double sqroot = Math.sqrt(max); // Find square root of n
        //Initialize array to hold prime numbers
        boolean primeBucket [] = new boolean [max+1];
        //Initialize all elements to true, then set non-primes to false
        for (int i=2; i<=max; i++){
            primeBucket[i]=true;
        }

        //Do all multiples of 2 first
        int j=2;
        for (int i=j+j; i<=max; i=i+j){ //start with 2j as 2 is prime
            primeBucket[i]=false; //set all multiples to false
        }
        for (j=3; j<=sqroot; j=j+2){ // do up to sqrt of n
            if (primeBucket[j]==true){ // only do if j is a prime
                for (int i=j+j; i<=max; i=i+j){ // start with 2j as j is prime
                    primeBucket[i]=false; // set all multiples to false
                }
            }
        }

        //Check input against prime array
        if (primeBucket[num] == true) {
            return true;
        }else{
            return false;
        }
    }//end primeCheck()

    //Method to validate input
    public void checkArgs(String [] args) throws Exception{
        //Check arguments for correct number of parameters
        if (args.length != 1) {
            throw new Exception();
        }else{
            //Get integer from character
            Integer num = Integer.valueOf(args[0]);
            input = num.intValue();
            //If less than zero
            if (input < 0) //If less than lower bounds
                throw new Exception();
            else if (input > max) //If greater than upper bounds
                throw new Exception();
        }
    }
}//End check4Prime
```

Running the Application (check4Prime.java)

To compile:

& javac check4Prime.java

To run:

$ java check4Prime 5

Right . . . 5 is a prime number!

$ java check4Prime 10

Sorry . . . 10 is NOT a prime number!

$ java check4Prime A

Usage: check4Prime x

where 0<=x<=1000

Running the Tests (check4PrimeTest.java)

Requires the JUnit api, junit.jar

To compile:

```
$ javac check4PrimeTest.java -classpath .:./junit-4.13.2.jar
```

To run:

```
$ java -classpath .:./junit-4.13.2.jar check4PrimeTest
Starting test...
.......
Time: 0.001

OK (7 tests)

Test finished...
```

Pros, and cons of TDD

- Test-Driven Development has its advantages, and disadvantages, as well.

- Some important benefits associated with TDD approach
  - You gain confidence that your code will meet its specification and requirements.
  - You express the end result before you start coding.
  - You better understand the application's specification and requirements.
  - You may implement simple designs initially and confidently refactor the code later to improve performance, without worrying about breaking the specification.

Pros, and cons of TDD

- Other advantages of TDD are:
  - Due to the extremely tight test/code cycles, something always works.
  - This means a TDD project can be turned over to someone else, likely a programming pair, for continued development.
  - The biggest advantage of TDD is the excellent fault isolation.
  - If a test fails, the cause must be the most recently added code.
  - Finally, TDD is supported by an extensive variety of test frameworks,

Pros, and cons of TDD (cont'd)

- It is nearly impossible, or at best, very cumbersome, to perform TDD in the absence of test frameworks.
- If a tester cannot find a test framework for the project language, Test Driven Development is a poor choice.
  - It is probably better to just change programming languages.
- At a deeper level, TDD is inevitably dependent on the ingenuity of the tester.
- Good test cases are necessary, but not sufficient for TDD to produce good code.
- A final disadvantage of TDD is that the bottom-up process makes it unlikely that "deeper faults," such as those only revealed by data-flow testing, will be revealed by the incrementally created test cases.
- These faults require a more comprehensive understanding of the code,

Open Questions of TDD

- The first open question for TDD is that of how to scale-up to large applications.
- It would seem that there are practical limits as to how much an individual can “keep in mind” during a development.
- This is one of the early motivating factors for program modularity and information hiding, which are the foundations of the object-oriented paradigm.
- If size is a problem, complexity is even more serious.
- Can systems developed with TDD effectively deal with questions such as reliability and safety?
- Such questions usually require sophisticated models, but these are not produced in TDD.

Open Questions of TDD (cont'd)

- There is also the question of support for long-term maintenance.
- The Agile Programming community and the TDD advocates maintain that there is no need for the documentation produced by the more traditional development approaches.
- The more extreme advocates even argue against comments in source code.
- Their view: the test cases are *the specification*, and well-written code, with meaningful variable and method names, is self-documenting.
- ***Time will tell.***