Agile/Automated Testing

By automating test cases, software engineers can easily run their test cases often. In this chapter, we will explain the following:

- Guidelines on when to automate test cases, considering the cost of creating the test cases
- the XP test-driven development practice and the open-source JUnit tool which is used to create these automated test cases
- the automation of acceptance tests, particularly with the open source FIT framework

The test practices discussed in this chapter come from the test-centric XP methodology. XP has two important test practices: test-driven development (TDD) (Beck, 2003) and customer acceptance testing. *Acceptance testing* is *formal testing conducted to determine whether or not a system satisfies its acceptance criteria (the criteria the system must satisfy to be accepted by a customer) and to enable the customer to determine whether or not to accept the system.* (IEEE, 1990) In this chapter, we will discuss both of these practices along with the open source tools that are often used to support them. We will also provide an extensive code example of the practices in action. Because agile methods emphasize automating all testing, this chapter provides a good deal of information about automated testing in general.

One overriding emphasis of both TDD and acceptance testing is that the tests should be automated (Crispen and House, 2003). By automated, we mean that the tests themselves are code. The tests can then be run over and over again with very little effort, at any time, and by anyone (Kaner, Bach et al., 2002). There are three main advantages to automating tests:

- Running the tests over and over again gives you *confidence* that the new work just added to the system didn't break or destabilize anything that used to work and that the new code does what it is supposed to do.
- Running the tests over and over (particularly acceptance tests) can also help you understand *what portion of the desired functionality has been implemented*.
- Together the set of automated tests can form a regression test suite. *Regression testing* is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements (IEEE, 1990). The purpose of these regression tests is to show that the software's behavior is unchanged unless it is specifically changed due to the latest software or data change (Beizer, 1990).

When tests have to be run manually (with someone sitting at the computer typing the input on the keyboard), the execution of the manual tests and the examination of the results can be error-prone and time consuming. When schedule pressures rise, manual testing often gets forgotten. So, automating tests can be very beneficial and is emphasized in agile development.

We do, however, need to be somewhat flexible and sensible in our quest for total test automation. Automating tests can be time consuming and expensive. Writing an automated test can take several orders of magnitude more time (2X – 10X more) than executing the test by hand once (Kaner, Bach et al., 2002). Often, XP projects have at least as much test code as production code (Beck, 2003) and, therefore, are themselves software applications (Craig and Jaskiel, 2002). This test code needs to be maintained just as implementation code does. Debugging and handling customer complaints can also be time consuming and expensive – so there is a tradeoff between spending the time to automate tests and spending time and money on customer complaints. The benefits of automated testing include: (1) production of a reliable system, (2) improvement of the quality of the test effort, (3) reduction of the test effort and (4) minimization of the schedule (Dustin, Rashka et al., 1999). We need to prudently trade off the costs and benefits of test automation.

Based on many years of building and maintaining automated unit and acceptance tests, Meszaros et al. (Meszaros, Smith et al., 2003) created their Test Automation Manifesto. The Manifesto contains lots of good advice to remember as you create your automated tests.

Automated tests should be:

- **Concise** – Test should be as simple as possible and no simpler.
- **Self Checking** – Test should report its results such that no human interpretation is necessary.
- **Repeatable** – Test can be run repeatedly without human intervention.
- **Robust** – Test produces same result now and forever. Tests are not affected by changes in the external environment.
- **Sufficient** – Tests verify all the requirements of the software being tested.
- **Necessary** – Everything in each test contributes to the specification of desired behavior.
- **Clear** – Every statement is easy to understand.
- **Efficient** – Tests run in a reasonable amount of time.
- **Specific** – Each test failure points to a specific piece of broken functionality (e.g. each test case tests one possible point of failure).
- **Independent** – Each test can be run by itself or in a suite with an arbitrary set of other tests in any order.
- **Maintainable** – Tests should be easy to modify and extend.
- **Traceable** – Tests should be traceable to the requirements; requirements should be traceable to the tests.

1 Test-Driven Development

The first testing practice we'll discuss is test-driven development (TDD). TDD is a unit testing practice that is used by software developers as they write code. With TDD, software engineers develop production code through rapid iterations of the following six steps:

1. Spending some time doing high- and/or low-level design (optional, see Section 1.1);
2. Writing a small number of automated unit test cases;
3. Running these unit test cases to ensure they fail (since there is no code to run yet);
4. Implementing code that should allow the unit test cases to pass;
5. Re-running the unit test cases to ensure they now pass with the new code; and
6. Restructuring the production and the test code, as necessary, to make it run better and/or to have better design.

As we said, there are rapid iterations of these six steps. In implementing some code, the programmer will often iterate steps 2 through 5 on a minute-by-minute basis. You might wonder about step 3. Why run the test cases to make sure they fail? There are three reasons why step 3 is done – all involving the unexpected event that the new test cases actually pass even though the new code hasn't been added yet:

1. there's a problem with the test, and it isn't testing what you think it is testing;
2. there's a problem with the code, and it's doing something you didn't expect it to do (it's a good idea to check this area of the code to find out what other unexpected things it's doing); and
3. maybe the code legitimately already performs the functionality correctly – and no more new code is needed (this is a good thing to know).

When programmers thoroughly follow TDD, a good set of automated unit test cases are produced. These test cases can be run over and over again – potentially multiple times each hour or at least once per day. There are three advantages to running these automated unit tests often:

- The test results tell us when we inadvertently break some existing functionality (Martin, 2003).
- You can add functions to a program or change the structure of the program without fear that you are breaking something important in the process (Martin, 2003). A good set of tests will probably tell you if you break something.
- Automated unit tests prevent backtracking and maintain development momentum (Kaner, Bach et al., 2002).

Programmers who use TDD find these automated unit-tests very helpful when maintaining code. When a problem is found in the code (by a software tester or by the customer), the first thing the programmer does is add a unit test case that would have found that error. The programmer runs the test to make sure the code fails in a similar manner to the newly identified defect and then fixes the code until the test case passes. In this way, the programmer also learns more about the kinds of tests that need to be written for a high quality system.

Research has been done to see whether or not TDD is a good practice to follow. Two major research studies found that the TDD practice helps programmers produce higher quality systems (George and Williams, 2003; Williams, Maximilien et al., 2003). One research study found that TDD did not help to produce a higher quality system (Müller and Hagner, 2002). However, in this last study, the programmers involved in the study had to write all their automated unit test cases before writing any production code. Normally, TDD test cases are written in a highly iterative manner, as described above. So, these research results support the need for this rapid iteration between test and production code to achieve the best benefits of TDD.

TDD shortens the decision-code feedback loop for the developer – in which the developer makes a decision on what to do, implements the decision, and is provided with feedback on this decision. Programmers who use TDD often become "test infected" (Beck, 2003) and really enjoy the security they get by repeatedly running an extensive set of automated tests on their code and seeing the results.

1.1 Test and Implementation Code as Design

In XP, TDD begins without any major/formal design effort occurring beforehand. Possibly, a pair of developers will decide to brainstorm a design and will sketch it on a whiteboard or a piece of paper. Alternately, the pair will decide to do a CRC card session (perhaps including a few more teammates in the activity.) But, either of these two activities is done informally without consuming much time. For the most part, a pair of developers looks at the user story and gets started iteratively writing tests and production code to satisfy the user story. Because the creation of automated unit test cases requires that the developer know the structure of code, the developer must decide what the code will look like in order to write the test(s). For example, the programmer will have to decide what method will be called, what parameters will need to be passed to this method, and what kind of value the method will return. Through these many small decisions, the pair of developers designs the production code as part of the TDD cycles..

1.2 Design before TDD

Alternatively, a team can spend some time and devise a documented design before starting the TDD cycles. Then, as developers implement code, they refer to this design to incrementally write test cases and production code. The initial design will tell them what methods are called, what parameters need to be passed, and what the return values need to be. Naturally, the developer can always change the initial design as part of the TDD cycles. No matter what development methodology is used, the initial design almost never exactly matches the actual design of the code that is implemented. This is also likely the case when developers do a design prior to starting TDD.

We will do an extensive TDD example in Section 3. In this example we will use the "Design before TDD" approach.

2 JUnit

Never in the field of software development was so much owed by so many to so few lines of code. -- Martin Fowler

JUnit[1] is an award-winning open source testing framework for Java written by Erich Gamma and Kent Beck. JUnit is used for white box testing. White box testing is testing that takes into account the internal mechanism of a system or component. (IEEE, 1990) Therefore, you must know the internal structure of the code. The framework can be used for white box testing for both unit test and integration test. It is fairly easy to learn to use JUnit because it is a Java framework. You download the framework, put it in your classpath, and create test cases by inheriting from the classes in the framework.

In the next section of this chapter, we provide an extensive example of TDD and writing test cases with JUnit. The following list summarizes the steps for creating test cases. For your reference, Figure 1 provides a class diagram of the JUnit framework. These steps are demonstrated via extensive code examples in the next section of this chapter.

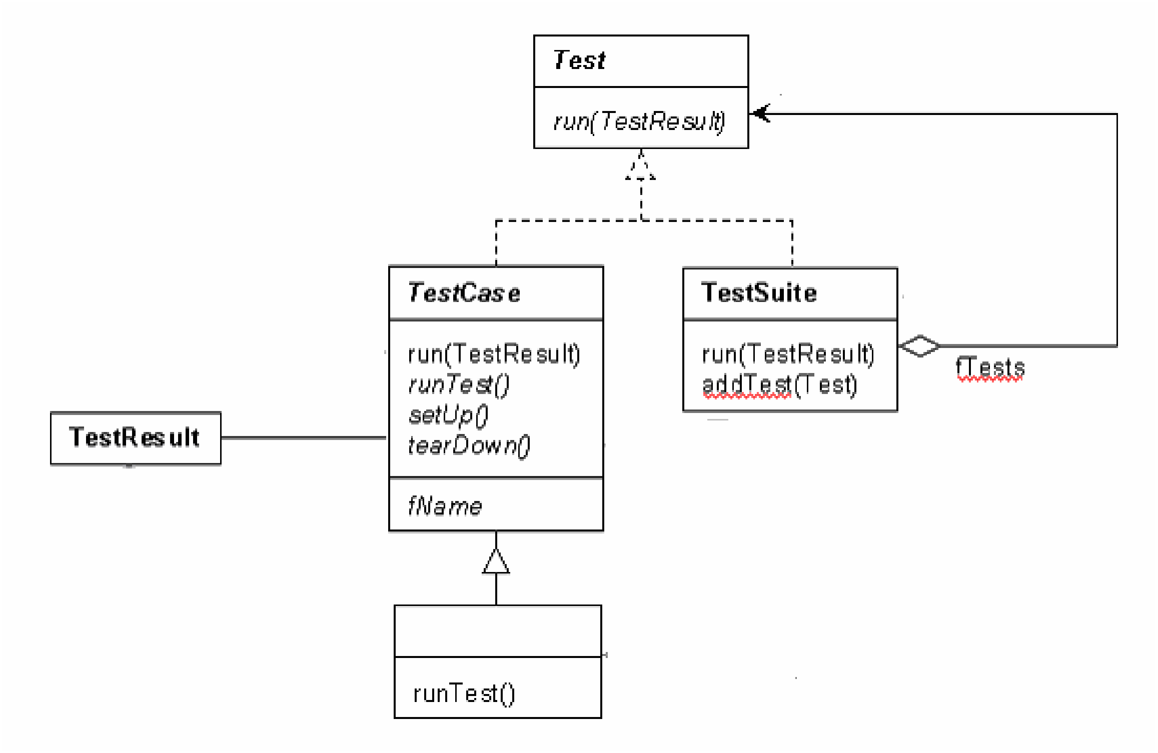


Figure 1: UML class diagram for JUnit

Define a subclass of `TestCase`. For instance, `MyTest`.
Override the `setUp()` method to initialize the object(s) and resources under test. Override the `tearDown()` method to release the object(s) and resources under test. Each test runs in the context of its own fixture, calling `setUp()` before and `tearDown()` after each test method to ensure there can be no side effects among test runs.
Define one or more public `testXXX()` methods that exercise the object(s) under test and assert expected results. There are various forms of assert available in the tool. See Table 1 for a description of these. You will probably use `assertTrue` and `assertEquals` most often. The JUnit framework defines an error class called `AssertionFailedError`. All the assertion methods in the JUnit framework throw an `AssertionFailedError`

[1] JUnit is available at http://junit.org/index.htm. At this website, there are also many resources and articles written by JUnit users around the world.

whenever an assertion fails. The JUnit framework catches this error and reports that the test has failed. If the `AssertionFailedError` object has any detail about the failure, the user interface displays that information to the user.

1. Alternatively, you can test whether the program throws an exception to verify that the test's execution path ends up inside the exception handler as expected using JUnit's `fail()` method.

Define in class `MyTest` a `main()` method that runs `MyTest` in batch mode. Running a `TestCase` automatically invokes all of its public `testXXX()` methods. JUnit provides both a textual and a Swing graphical user interface. Both user interfaces indicate how many tests were run, any errors or failures, and a simple completion status (a text message or a red/green bar). You specify your choice of interface in your main method. The simplicity of the user interfaces is the key to running tests quickly. You can run your tests and know the test status with a glance.
Optionally, define a static `suite()` method that creates a `TestSuite` containing all the `testXXX()` methods of `MyTest.` A `TestSuite` is a composite of other tests, either instances of `TestCase` subclasses or other `TestSuite` instances. The composite behavior of the `TestSuite` allows you to assemble test suites of tests and run all the tests automatically and uniformly to yield a single pass or fail status. Commonly, there is a one-to-one correspondence between classes in the implementation code and subclasses of `TestCase` (for example, the `Auction` class in the code hierarchy would have a corresponding `AuctionTest` class in the test code hierarchy`)`. A `TestSuite` can be used to gather together all the `TestCase` instances and run their test cases.

Table 1: JUnit Asserts

assert	Description
`assertEquals(a,b, delta)`	Asserts that a and b are equal. a and b could be Booleans, bytes, chars, doubles, floats, ints, longs, shorts, Strings, or any Java Objects. Doubles and floats require a third parameter, delta, which specifies the maximum variance under which a and b would be declared equal.
`assertTrue(a)`	Asserts that a Boolean condition, a, is true.
`assertFalse(a)`	Asserts that a Boolean condition, a, is false.
`assertNull(a)`	Asserts that an object, a, is null.
`assertNotNull(a)`	Asserts that an object, a, is not null.
`assertSame(a, b)`	Asserts that two objects, a and b, refer to the same object.
`assertNotSame (a, b)`	Asserts that two objects, a and b, do not refer to the same object.

When you write JUnit test cases, you want to use all the white box testing strategies, such as boundary value analysis and equivalence class partitioning. You also should strive to get the maximum method, statement, branch, and condition coverage with your tests. With automated testing, it is unnecessary to *instrument* the code. When you instrument code, you add lines of code to the program that are only intended to help in the testing – for example, adding a line that will print out a value. Instrumenting code is a concern because these extra lines of code could cause errors, affect performance, and/or may need to be commented out when testing is complete. A big advantage of the JUnit framework is that the test code is completely independent of the program being tested because it lives in a totally separate code hierarchy. Thus, you don't run the risk of introducing a bug just because you add a test.

3 Test-Driven Development Example

We will now go through a TDD example using our Auction System example to show you how JUnit works. For the example, we will use the "Design before TDD" version of TDD. We think this is an appropriate approach for this book because we want you to be able to follow our thought processes and understand where we are going.

3.1 Starting Point

First of all, let's have a simple starting point. What are the most essential things in a Monopoly game? The game board and the cells! After all, Monopoly is a board game. A game board has many cells, and a cell, regardless of the type (property, utility, railroad, etc.), has a name. We should be able to add cells to a game board. A cell has no reason to exist if not for the GameBoard – so we use composition. This seems good enough to get started. Figure 2 is a UML class diagram that depicts the idea.

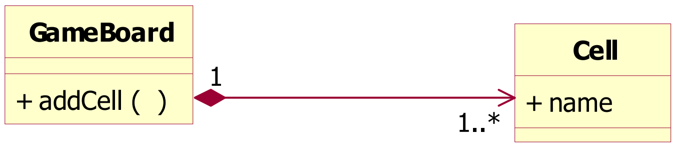


Figure 2: A starting point of the Monopoly game

Let's write a test for the game board. This test should be just enough to show our design.

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

      public void testAddCell() {
            GameBoard gameboard = new GameBoard();
            assertEquals(0, gameboard.getCellNumber());
            Cell cell = new Cell();
            gameboard.addCell(cell);
            assertEquals(1, gameboard.getCellNumber());
      }
}
```

The test shows that when a game board is initialized, it has no cell. After we add a cell to the game board, it'll have one cell. The test does not pass the compiler because we do not have the `GameBoard` and the `Cell` classes created yet. From the class diagram, we want an `addCell` method in `GameBoard`, and a `name` attribute in `Cell`. From the test, we see that we need a method to get the number of cells from a `GameBoard`. We can write these two classes:

```
public class GameBoard {
      public void addCell(Cell cell) {
      }

      public int getCellNumber() {
            return 0;
      }
}

public class Cell {
      private String name;

      public String getName() {
            return this.name;
      }

      public void setName(String name) {
            this.name = name;
      }
}
```

The initial purpose of writing these two classes is just to pass the compiler. We can compile the program now. If we run thee test, we can see that we can pass the first assertion, but not the second one. This is because we simply return 0 in `getCellNumber` of `GameBoard`. We need to use some data structure to store the cells. We use `ArrayList` here, because the cells should be put in an ordered list. We don't want the cells to change their orders in the middle of a game. Therefore, `GameBoard` can be implemented as:

```
public class GameBoard {
      ArrayList cells = new ArrayList();

      public void addCell(Cell cell) {
            cells.add(cell);
      }

      public int getCellNumber() {
            return cells.size();
      }
}
```

We pass the first test now. Let's move on to the next step.

3.2 Let the Game Begin

We have a game board, and we can add cells to the game board. What is missing if we want to play the game? The players! Look at the requirements and find some requirements that are related to players. We start with the three requirements that seem to be easy:

1. Before the game begins, one player shall enter the number of players and the names of the players.
2. At the beginning of the game, all the players shall be at the Go cell.
3. The players shall move based on the dice roll. When the player reaches the end of the game board (the Go cell), he shall cycle around the board again.

Even with these three “easy” requirements, we have some design considerations.

1. What takes care of the players? Adding the players to the game board should work, but can we say that a game board has some players? It does not sound right. Therefore, we decided to create a new class to manage the players. We gave this class a cool name: the `GameMaster`.
2. There is always only one `GameMaster` in the game. We can use a design pattern and make it a singleton.
3. It is reasonable to let the `GameMaster` to move the players on the game board. Therefore, the `GameMaster` should have the knowledge of the players and the game board.
4. From the requirement, the Go cell is indispensable. When a game board is created, there should already be one Go cell.
5. What puts the players at the Go cell at the beginning of the game? Since there is guaranteed to be a Go cell in a game board, we decided to put a player at the Go cell as soon as the player instance is created.
6. We created a simple game board so that we could test the player’s movement. The simple game board looks like Figure 3. To simplify the situation, there are only a Go cell and five different property cells. While later we can test with a more “realistic” game board – it is first good to write the simplest test cases that can force the conditions we want to occur. This Simple Game Board does that for us.

   What is the difference between the Go cell and a property cell? Well, they are totally different cells actually, except that they both have names. Thus, we decided to make the Go cell and the property cell subclasses of the `Cell` class.

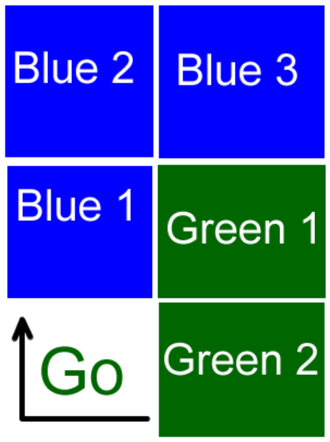


Figure 3: The Simple Game Board

These ideas are summarized in the UML class diagram in Figure 4. The rest of the JUnit example will show how to apply TDD to develop a system that satisfies these three requirements.

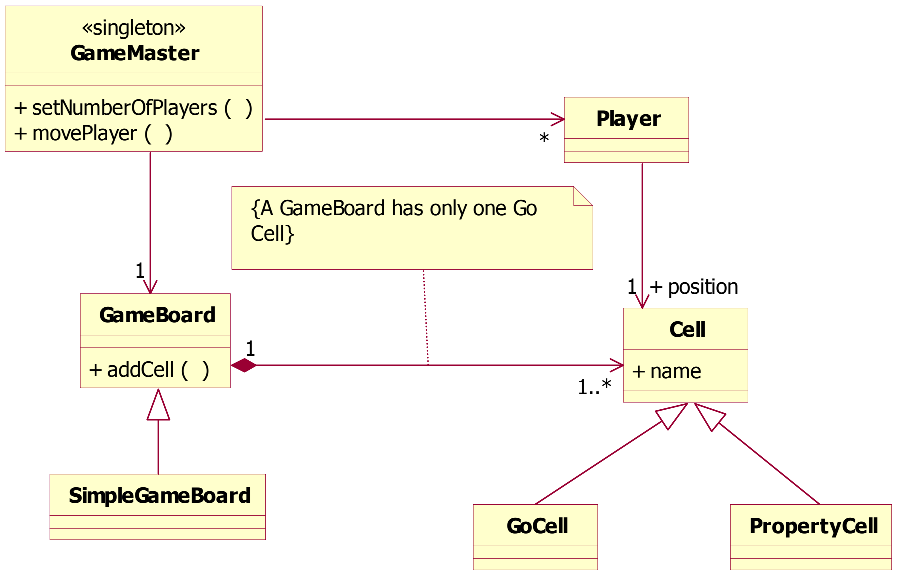


Figure 4: System Design – Introducing `GameMaster` and `Player`.

3.3 *The One and Only GameMaster*

We want to apply the singleton design pattern to the `GameMaster` class. Singleton means there can be only one instance of this class. Whenever we request an instance from the singleton class, we will always get the same instance. We write our intent in a test case:

```
public class GameMasterTest extends TestCase{
    public void testSingleton() {
        GameMaster instance1 = GameMaster.instance();
        assertNotNull(instance1);
        GameMaster instance2 = GameMaster.instance();
        assertNotNull(instance2);
        assertSame(instance1, instance2);
    }
}
```

We need to create the `GameMaster` class, and also a static method `instance`, to make the test case compile. We may start this with a code skeleton for `GameMaster`:

```
public class GameMaster {
      public static GameMaster instance() {
            return null;
      }
}
```

Because we only return null in the `instance` method, we cannot pass the test case. There is a standard way to implement the singleton pattern in Java: create a static member for the singleton instance, and use lazy instantiation to initialize the instance. We modified `GameMaster` so that the singleton instance is always returned:

```
public class GameMaster {
      static private GameMaster singleton;

      public static GameMaster instance() {
            if(singleton == null) {
                  singleton = new GameMaster();
            }
            return singleton;
      }
}
```

The test passes, and we have a singleton instance of the `GameMaster`.

3.4 The Go Cell

At this moment, the only thing special about the Go cell is that the name of the cell is always Go. We cannot change the name of the Go cell. We may do so by setting the name of the Go cell in `GoCell`'s constructor, and override `setName` method so that this method does nothing:

```
public class GoCell extends Cell {
      public GoCell() {
            super.setName("Go");
      }

      void setName(String name) {
      }
}
```

The implementation is so easy that we do not even bother to write a test. Usually, we do not need to write tests for the accessor (getters and setters).

In our design, the game board has a Go cell when it is created. We need to modify the `GameboardTest` to show this. We also need to add a new test to make sure that the first cell is the Go cell.

```
public class GameboardTest extends TestCase {
      :
      public void testAddCell() {
            GameBoard gameboard = new GameBoard();
            assertEquals(1, gameboard.getCellNumber());
            Cell cell = new Cell();
            gameboard.addCell(cell);
            assertEquals(2, gameboard.getCellNumber());
      }

      public void testFirstCell() {
            GameBoard gameboard = new GameBoard();
            Cell firstCell = gameboard.getCell(0);
            assertEquals("Go", firstCell.getName);
      }
}
```

The compiler tells us that we need a `getCell` method for `GameBoard`. No problem:

```
public class GameBoard {
      :
      public Cell getCell(int index) {
            return (Cell)cells.get(index);
      }
}
```

Although we can compile the code now, we cannot pass the test. JUnit reports that `testAddCell` has an assertion error, and `testFirstCell` has an index out of bound exception. The reason for these errors is that the game board has no cell when it is created. We want the game board to have a Go cell when it is created. We can put the code in the constructor of `GameBoard`:

```
public class GameBoard {
      public GameBoard() {
            addCell(new GoCell());
      }
      :
}
```

Now when the game board has a Go cell when instantiated. What about `PropertyCell`? For this time being, the `PropertyCell` does not have any special behaviors. Just making it a subclass of `Cell` is good enough:

```
public class PropertyCell extends Cell {
}
```

One may argue that we need a more detailed test for adding a new cell. When several cells are added to a game board, shouldn't we write a test to make sure that these cells are added in order? Looking at `addCell` method in `GameBoard`, we can see that it only calls a method of `ArrayList`. If we write a test to see that the cells are added in order, whether the test will pass depends on the correct implementation of `ArrayList`, which is out of our control. This is another example of unnecessary test.

3.5 The SimpleGameBoard

The SimpleGameBoard is a subclass of GameBoard. It does not have additional methods or member variables. However, several cells are created and added to the game board when a new instance of SimpleGameBoard is created. The code of SimpleGameBoard is listed below. There is nothing worthy of testing in the SimpleGameBoard since we already tested all that functionality before, so we don't need to create any test for it.

```
public class SimpleGameBoard extends GameBoard {
      public SimpleGameBoard() {
            super();
            Cell blue1 = new PropertyCell();
            Cell blue2 = new PropertyCell();
            :

            blue1.setName("Blue 1");
            blue2.setName("Blue 2");
            :

            addCell(blue1);
            addCell(blue2);
            :
      }
}
```

3.6 The Player

Figure 4 shows that the player knows his or her position. Let’s create the `Player` class, with a member variable and the accessors:

```
public class Player {
      private Cell position;

      public Cell getPosition() {
            return this.position;
      }

      public void setPosition(Cell newPosition) {
            this.position = newPosition;
      }
}
```

Again, we don’t need to write tests for the accessors. However, we do need a test to show that when a player is created, the position is at the Go cell:

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

      public void testStartPosition() {
            GameBoard board = new SimpleGameBoard();
            GameMaster.instance().setGameBoard(board);
            Player player1 = new Player();
            Player player2 = new Player();
            Cell go = board.getCell(0);
            assertSame(go, player1.getPosition());
            assertSame(go, player2.getPosition());
      }
}
```

First of all, the GameMaster needs a method to set up the game board:

```
public class GameMaster {
      :
      private GameBoard gameBoard;
      :
      public void setGameBoard(GameBoard board) {
            this.gameBoard = board;
      }
}
```

This test fails, because the players' positions are both null. We can initialize the player's position in the constructor of the `Player` class:

```
public class Player {
      :
      public Player() {
            position = GameMaster.instance().getGameBoard().getCell(0);

      }
}
```

The test passes now. This means when we create a new instance of `Player`, the position of the player is set to the Go cell (cell 0) of the current game board.

One of the requirements states that at the beginning of the game, the players shall enter the number of players. In our design, the players are initialized when calling `setNumberOfPlayers` on the `GameMaster`. We can write a test to show that after this call, we will have exactly the same number of players, all at the Go cell.

```
public class GameMasterTest extends TestCase{
    :
    public void testPlayerInit() {
        master = GameMaster.instance();
        master.setGameBoard(new SimpleGameBoard());
        master.setNumberOfUsers(6);
        assertEquals(6, master.getNumberOfPlayers();
        Cell go = master.getGameBoard().getCell(0);
        for (int i = 0; i < 6; i++) {
            Player player = master.getPlayer(i);
            assertSame(go, player.getPosition());
        }
    }
}
```

The compiler is complaining about the missing methods. We need to add those methods to `GameMaster` to pass the compiler:

```
public class GameMaster {
      :
      public void setNumberOfPlayers(int number) {
      }

      public int getNumberOfPlayers() {
            return 0;
      }

      public Player getPlayer(int index) {
            return null;
      }
}
```

Again, this is just a code skeleton. It helps us pass the compiler. Since there is no real implementation in the code, the test fails. We need to think about how we may store the players in the game master. The players should be stored in order, so `ArrayList` would be a nice choice. When setting up the number of players, we can simply create several instances of `Player` and put them in the `ArrayList`. We may also get the number of players or query a player via index from the `ArrayList`.

```
public class GameMaster {
      private ArrayList players;
      :
      public void setNumberOfPlayers(int number) {
            players = new ArrayList(number);
            for(int i = 0; i < number; i++) {
                  Player player = new Player();
                  players.add(player);
            }
      }

      public int getNumberOfPlayers() {
            return players.size();
      }

      public Player getPlayer(int index) {
            return (Player)players.get(index);
      }
}
```

The test passes. We now can specify the number of players, and these players are put at the Go cell. Finally, we are ready to deal with the player movement.

3.7 Test Makes the Players Go Round

We have a game board. We have players. It's time to move the players. To make the example simpler, we will just write test cases to move a single player. First, let's consider the case in which the player does not reach the end of the game board:

```
public class GameMasterTest extends TestCase{
    :
    public void testMovePlayerSimple() {
        master = GameMaster.instance();
        master.setGameBoard(new SimpleGameBoard());
        master.setNumberOfUsers(1);
        Player player = master.getPlayer(0);
        master.movePlayer(0, 2);
        assertEquals("Blue 2", player.getPosition.getName());
        master.movePlayer(0, 3);
        assertEquals("Green 2", player.getPosition.getName());
    }
}
```

Because the player's movement is based on the dice roll, we put two parameters for `movePlayer` method. The first one is the index of the player; the second the value of the dice role. In this test, we move the first player two steps forward, and check if it lands on Blue 2; and then we move him three steps further, and check if it lands on Green 2. We need to add this method to make the test compile:

```
public class GameMaster {
    :
    public void movePlayer(int playerIndex, int diceRoll) {
    }
}
```

After the program compiles OK, we may run the test. We have not written anything in `movePlayer`, so the player always stays at the Go cell. Therefore, the test fails. How do we move the player? We can think of a straightforward algorithm:

1. Find out the player's position. (`GameMaster` can find out a player with an index. The player knows its position.)
2. Find out the index of the cell the player is in. (`GameBoard` has the knowledge. However, it doesn't have an interface for this.)
3. Add the index with the dice roll value. The result is the index of the cell the player is moving to.
4. Find the cell object with an index. (`GameBoard` already has an interface for this.)
5. Set the position of the player to the cell object.

The only missing piece in this algorithm is that we cannot find out the index of a certain cell. GameBoard knows all the cells, so it must know the index of every cell. We just need to add a method. With TDD, of course, we need to write a test first.

```
public class GameboardTest extends TestCase {
      :
      public void testGetCellIndex() {
            GameBoard gameBoard = new SimpleGameBoard();
            Cell blue2 = gameBoard.getCell(2);
            Int index = gameBoard.getCellIndex(blue2);
            assertEquals(2, index);
            Cell notExist = new Cell();
            Index = gameBoard.getCellIndex(notExist);
            assertEquals(-1, index);
      }
}
```

In this test, we not only state that `GameBoard` should have `getCellIndex` method, but also specify the behavior of this method. If the cell is found, the index is returned. However, if the cell is not found, the method returns -1. Actually this is easy if we are familiar with the `ArrayList` API[2]:

```
public class GameBoard {
      :
      public int getCellIndex(Cell cell) {
            return cells.indexOf(cell);
      }
}
```

Then we can finish the `movePlayer` method for `GameMaster`:

```
public class GameMaster {
    :
    public void movePlayer(int playerIndex, int diceRoll) {
         Player player = getPlayer(playerIndex);
         Cell playerPosition = player.getPosition();
         int oldIndex = gameBoard.getCellIndex(playerPosition);
         int newIndex = oldIndex + diceRoll;
         Cell newPosition = gameBoard.getCell(newIndex);
         player.setPosition(newPosition);
    }
}
```

We pass the test! However, we have not finished yet. When a player reaches the end of the game board, he or she shall cycle around. Let's write a test to test this situation:

```
public class GameMasterTest extends TestCase{
    :
    public void testMovePlayerCycle() {
        master = GameMaster.instance();
        master.setGameBoard(new SimpleGameBoard());
        master.setNumberOfUsers(1);
        Player player = master.getPlayer(0);
        master.movePlayer(0, 2);
        master.movePlayer(0, 5);
        assertEquals("Blue 1", player.getPosition.getName());
    }
}
```

In this test, we move the player two steps then five steps. The player should reach the end of the game board, and then start again from the Go cell, and finally land on Blue 1. When we try to run the test, we will run into an array index out of bound exception. This

[2] To be honest, we did not know `ArrayList` has such a method. We were planning to do a linear search through the `ArrayList`. That was why we had this test. If we had known this method before we wrote the test, we would not have written the test.

is because the value of the new index is 7, and there is no 8th cell in the game board. We need to modify `movePlayer` in `GameMaster` to pass this test:

```
public class GameMaster {
    :
    public void movePlayer(int playerIndex, int diceRoll) {
         :
         int newIndex =
                  (oldIndex + diceRoll) % gameBoard.getCellNumber();
         :
    }
}
```

Run the test again, and we can see that the implementation passes the test. Do we need to care about the situation when the player's position is not found on the game board? No, because it is not possible.

Looking at the `GameMasterTest`, we can see some repeated code to initialize the `GameMaster`. We can use the `setUp` method to remove the repetition. The `setUp` method is called before each test method is called. There is a similar method called `tearDown`. However, `tearDown` is called after each test method is called. It is usually used to free the resources that are allocated in `setUp` (such as file handle, network connection, or database connection). In this example, we do not allocate any resource in `setUp`, so `tearDown` is not needed. After the cleaning up, `GameMasterTest` looks like this:

```
public class GameMasterTest extends TestCase{
    GameMaster master;

    public void setUp() {
        master = GameMaster.instance();
        master.setGameBoard(new SimpleGameBoard());
    }
    :

    public void testPlayerInit() {
        master.setNumberOfUsers(6);
        assertEquals(6, master.getNumberOfPlayers();
        :
    }

    public void testMovePlayerSimple() {
        master.setNumberOfUsers(1);
        Player player = master.getPlayer(0);
        :
    }

    public void testMovePlayerCycle() {
        master.setNumberOfUsers(1);
        Player player = master.getPlayer(0);
        :
    }
}
```

3.8 What Have We Here?

Let's take a look at what we have so far. Figure 5 shows the class diagram. The blue classes are test cases. Their super class, `TestCase`, is now shown in this diagram. Accessor methods are now shown in this diagram.

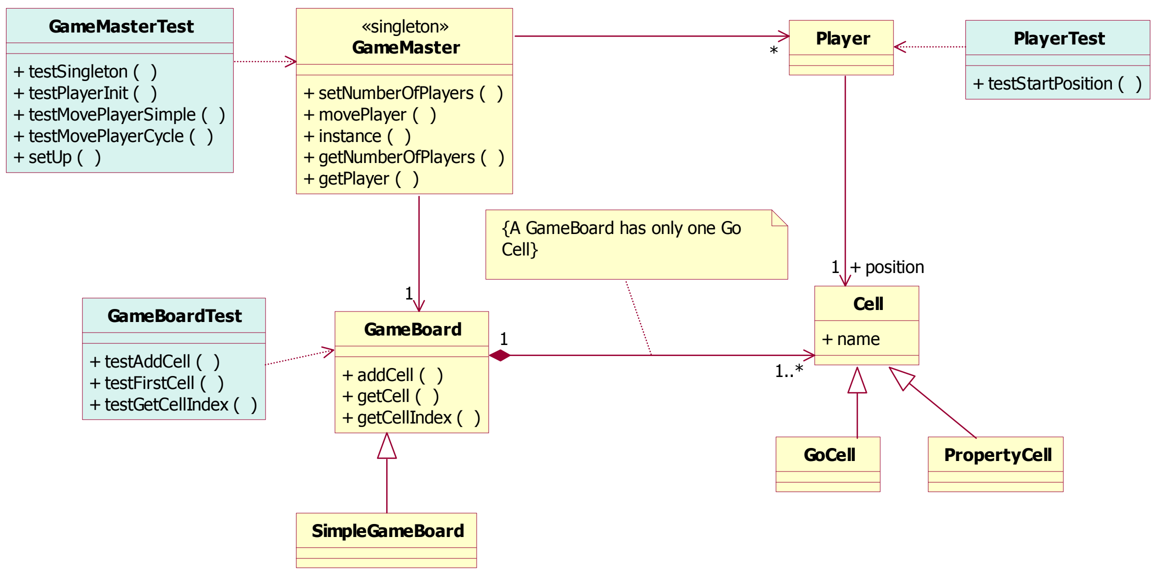


Figure 5: A Snap Shot of Current System

4 Acceptance Testing

Acceptance tests are black box test cases that are jointly written by a developer and a customer. An acceptance test is a concrete situation, or scenario, that the system might encounter when using the functionality of a user story. When an acceptance test case runs properly, this lets the customer know that the user story has been properly implemented – at least for the scenarios defined in the acceptance tests. Customers are generally not software engineers so they don't understand about equivalence class partitioning, boundary value analysis, test coverage, or the like. They usually provide one very basic "success" test case based on the requirements. So we must not take the acceptance test cases written by the customer as the only black box test cases we run. We must write all those test cases that test all the different combinations of bad and good things that users of our software might try to do.

Acceptance tests have the same four parts as all black box test cases: a test ID; a description that describes the preconditions of the test and the steps of the test; the expected results of running the test; and the actual results of running the test.

The dialog between the customer and the developer when the acceptance test cases are created usually leads to the discussion of many details about the user story – details the developer needs to know about what is entailed in the user story. The conversation also helps the development team to understand how difficult a simple user story can get. (Astels, Miller et al., 2002)

With XP, the progress of a development effort is often tracked by the number of acceptance test cases that run successfully. Additionally, in XP there is an emphasis on *automating* the acceptance test cases so that they can be run many times as the functionality of the program grows. We always want to have a level of confidence that the new functionality we just added did not break any of the functionality that used to work. Running the automated acceptance test cases often can help us do that.

5 FIT

As with JUnit for unit testing, there is an open source framework for automating an acceptance test. This framework is called the *Framework for Integrated Test*, or FIT[3]. FIT was developed by Ward Cunningham. With FIT, acceptance test cases are created in tables of tests and their associated expected results, as shown in Figure 6. Tables are understandable to non-technical customers so we can more easily partner with them for writing automated tests when the tests are expressed in this form. As you will see with the extensive example in the next section, the development team writes simple "fixtures" that serve as linkage between the program and the HTML documents to automate the tests.

The test cases are written in HTML tables. The results are then provided based on the background colors in the cells. Like JUnit, a green background means the test case ran and our expected results match the actual results. A yellow background indicates the test case failed either because the code could not run or an exception was thrown. A red background means the code ran but the actual output of the program and the expected results did not match.

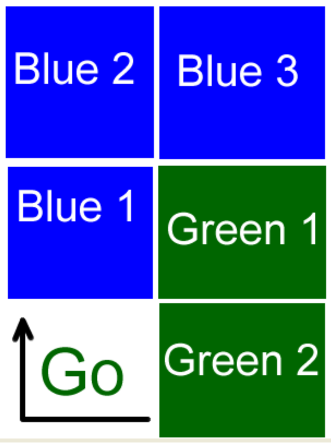
6 Acceptance Testing Example: Using FIT

In this example, we are going to use FIT to develop acceptance tests for the Monopoly game. Earlier in this chapter, we wrote unit tests for the Monopoly game using JUnit. With JUnit, we need to have some knowledge of the programming language, and whenever any test cases are added or modified, we need to recompile the test programs. Therefore, JUnit is not an appropriate tool for writing acceptance tests.

While unit tests are written by the developers, acceptance tests are written by the customers, with some assistance from the developers. Customers usually do not have decent programming knowledge, so it is better if the acceptance tests are written in an easily understood script. Additionally, the customers should be able to add, delete, or modify tests cases quickly. The tables that are used with FIT can be created with applications customers are familiar with, such as Microsoft Word or Excel. Figure 6 shows an example of a test document that can be written by a customer using a word processing application. With FIT, the test cases are saved as HTML. Most word processing and spreadsheet programs allow you to save your document in HTML form.

[3] http://fit.c2.com

A simple game board is set up as follows. A player moves based on his dice roll. When the user reaches the end of the board, he cycles around.

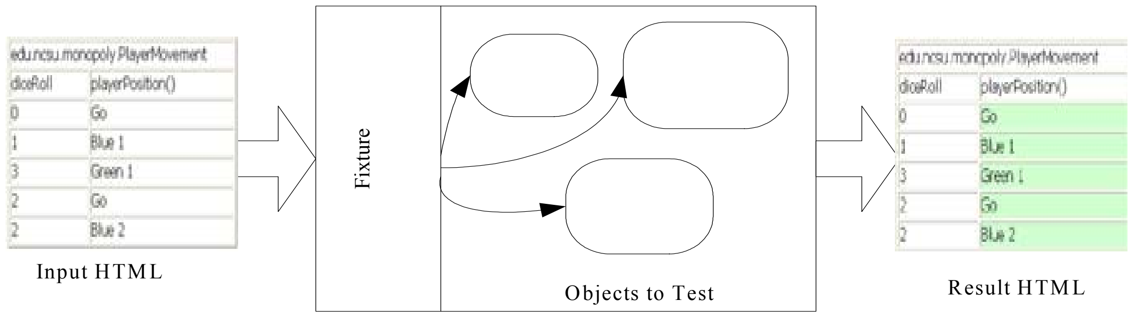


edu.ncsu.monopoly.PlayerMovement	
diceRoll	playerPosition()
0	Go
1	Blue 1
3	Green 1
2	Go
2	Blue 2

Figure 6: An Example of Acceptance Test.

In the table in Figure 6, the first row identifies the name of the test case. The rest of the table shows a scenario of player movement. We enter the values of the dice rolls, and check whether the player's position is as expected.

After the tables are created, the developers write "fixtures" to exercise the objects the customers would like to test. We use a FIT runner to run the tests. FIT binary download comes with several FIT runners. It is also easy to create customized FIT runners. FIT runners parse the HTML tables, feed the data from the tables to the fixtures, run the tests, and generate the HTML files that report the results. If the test passes, the cells in the table have a green background (which can only be shown as shaded here). Cells with a yellow or red background indicate unexpected values. Figure 7 shows the way FIT works.



edu.ncsu.monopoly.PlayerMovement	
diceRoll	playerPosition()
0	Go
1	Blue 1
3	Green 1
2	Go
2	Blue 2

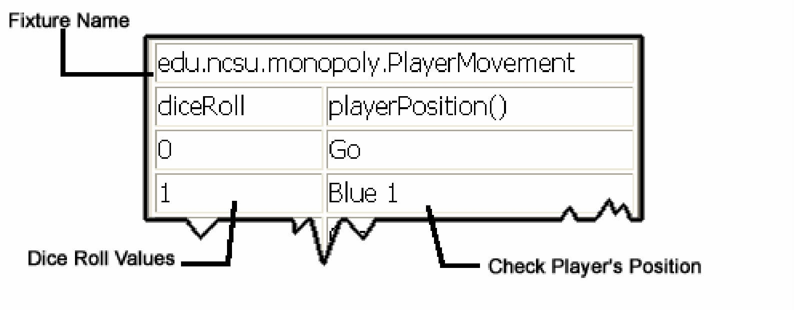
edu.ncsu.monopoly.PlayerMovement	
diceRoll	playerPosition()
0	Go
1	Blue 1
3	Green 1
2	Go
2	Blue 2

Figure 7: How FIT Works.

As with JUnit, FIT is a Java framework that provides classes we can extend to write our test fixtures. There are three classes we can extend or use to write our test fixtures: `RowFixture`, `ColumnFixture`, and `ActionFixture`. If these are not enough, we can still create customized fixtures. `ColumnFixture` and `ActionFixture` are used most often and are described in the next two subsections of this chapter.

6.1 Using Column Fixture

`ColumnFixture` is probably the easiest fixture. It works like a spreadsheet. We can enter some values in the spread sheet, and check out whether the result is correct. Figure 8 shows the structure of a `ColumnFixture` table. The first row of the table is the name of the class that implements the test fixture. The second row names the different attributes and values returned from some methods that we want to check. The conditions and test values are listed beginning from the third row. Each column represents a different condition or test value of the test. For example, in the third row, we want to make sure that the player starts at the Go cell.


Fixture Name
Dice Roll Values
Check Player's Position

edu.ncsu.monopoly.PlayerMovement	
diceRoll	playerPosition()
0	Go
1	Blue 1

Figure 8: A `ColumnFixture` Table.

The methods of `ColumnFixture` map to the columns in the test data to fields or methods of its subclasses. The column processes each row in order, top to bottom, and each column within each row, left to right. The code for this fixture is listed below. Keep in mind that the intention of this program is to make acceptance testing easier. Good program design is not our primary concern.

```
public class PlayerMovement extends ColumnFixture {
    public int diceRoll;
    private GameMaster gameMaster;

    public PlayerMovement() {
        gameMaster = GameMaster.instance();
        gameMaster.reset();
        gameMaster.setGUI(new MockGUI());
        gameMaster.setGameBoard(new SimpleGameBoard());
        gameMaster.setNumberOfPlayers(1);
    }

    public void execute() throws Exception {
        gameMaster.movePlayer(0, diceRoll);
    }
    public String playerPosition() {
        return gameMaster.getCurrentPlayer().getPosition().getName();
    }
}
```

The constructor of this class initializes the test. We reset the whole game, assign a game board to the game, and set the number of players to 1. The `execute()` method is invoked before the fixture runs each row. With some observation, we can see that:

1. The input of the test, `diceRoll`, is a public attribute of the class.
2. The check of the test, `playerPosition()`, is a public method of the class.

After we have both the fixture and the HTML test case, we can use a FIT runner to run the test. Here we use FileRunner to generate the result. FileRunner is a command line tool. The usage of FileRunner is shown below. Fit.jar, the library for FIT, must be included in the CLASSPATH environment variable. Also, your program must be in the CLASSPATH too.

```
java fit.FileRunner <input file> <result file>
```

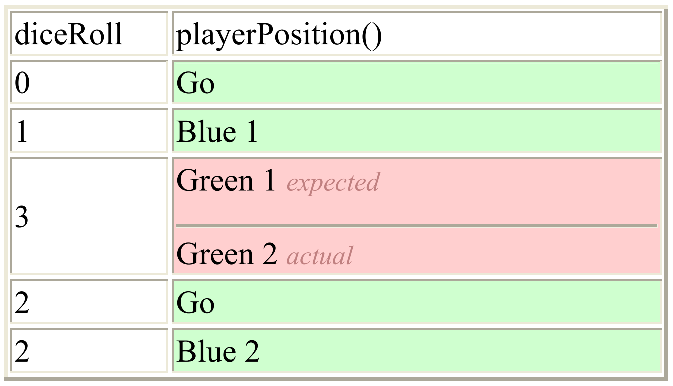
Then we can view the result file. It should look like Figure 9 .

edu.ncsu.monopoly.PlayerMovement	
diceRoll	playerPosition()
0	Go
1	Blue 1
3	Green 1
2	Go
2	Blue 2

Figure 9: The Result of PlayerMovement.

If the test does not pass, the result will show both the expected value and the actual value in red cells. Figure 10 shows the example of such a situation. If there are red cells in the output file, we know that there is some mismatch between the customer's expectation and what the system really does.

edu.ncsu.monopoly.PlayerMovement

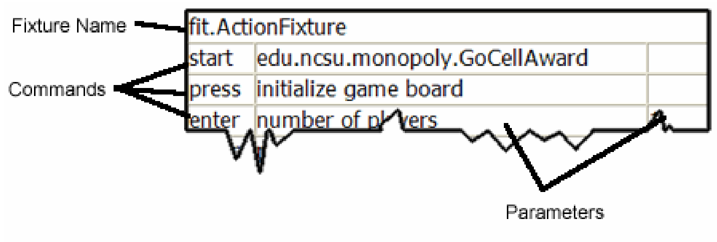


diceRoll	playerPosition()
0	Go
1	Blue 1
3	Green 1 *expected* Green 2 *actual*
2	Go
2	Blue 2

Figure 10: Mismatch of Expected and Actual Values

6.2 Using Action Fixture

Test cases can usually be described as a sequence of actions. This is what the `ActionFixture` does: describing a scenario, and checking whether the states of the system are as expected. `ActionFixture` also uses a table to describe a test case. There are at least three rows, and always three columns, in the test table. Figure 11 shows the structure of an `ActionFixture` table.



fit.ActionFixture		
start	edu.ncsu.monopoly.GoCellAward	
press	initialize game board	
enter	number of pl vers	

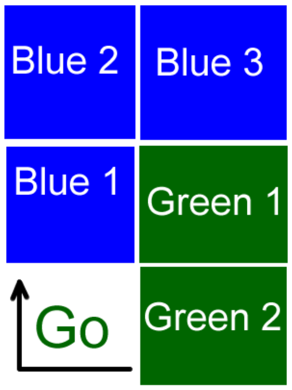
Figure 11: An `ActionFixture` Table.

1. The first column in the first row must specify the name of the test fixture used to run the test. The class that implements the fixture must be an instance of `ActionFixture` or its subclass.
2. Starting from the second row, the first column specifies the "commands" for the fixture, and the second and third columns supply the first and second parameters to the commands. There are four commands for an `ActionFixture`:
   - `start` Starts a fixture. This command has one parameter: the name of the fixture class.
   - `enter:` This command simulates a user entering a value in a text field or text box. It has two parameters. The first parameter is name of the field, and the second is the value to be put in this field.
   - `press`: This command simulates a user clicking a button. It has one parameter: the name of the button.
   - `check:` This command is used to check whether the state of the system is as expected. It has two parameters. The first is the name of the state, and the second is the expected value of this state.

The descriptions of the commands are given from the customers' perspective. Although these commands simulate the users' interaction with the system via the graphical user interface (GUI), FIT is not a GUI testing framework. It is used to test the business logic behind the user interface. When looking from the fixture program developer, those commands interact with the fixture like this:

- `start`: FIT will new an instance of the class specified in the parameter. All the subsequence commands will be operated on this instance.
- `enter`: FIT will call a method with one parameter. The name of the method is the name of the field in the test case. FIT will try to convert the value supplied in the test case to an appropriate type for the method. If the conversion fails, an exception will be reported in the result.
- `press`: FIT will call a method with no parameter. The name of the method is the name of the button in the test case.
- `check`: FIT will call a method with no parameter, and compare the returned value with the expected value. The name of the method is the name of the state in the test case.

Figure 12 shows a test case that uses the `ActionFixture`. In this example, we want to test that when the player passes or lands on the Go cell, he/she may collect $200 award.



fit.ActionFixture		
start	edu.ncsu.monopoly.GoCellAward	
press	initialize game board	
enter	number of players	1
enter	player dice roll	7
check	player money	1700
enter	player dice roll	5
check	player money	1900
enter	player dice roll	2
check	player money	1900

Figure 12: An `ActionFixture` Example.

This test describes the following scenario:
1. Initialize the game board.
2. Tell the game that there is only one player.
3. The player rolls the dice. The dice roll is 7. The player moves forward, passes the Go cell, and lands on Blue 1. The player should receive $200 award.
4. Check that the player has $1700.
5. The player rolls the dice. The dice roll is 5. The player moves forward and lands on the Go cell. The player should receive $200 award.
6. Check that the player has $1900.
7. The player rolls the dice. The dice roll is 2. The player moves two steps forward.
8. Check that the player's money is still $1900.

After the test is specified, the developer needs to write the fixture. From the test table, and the rules of `ActionFixture`, we know that:

1. The fixture's implementation class is `edu.ncsu.monopoly.GoCellAward`. It is a subclass of `ActionFixture`.
2. The command **`press`** needs a method without parameters. We need a `initializeGameBoard()` method in `GoCellAward` class.
3. The command **`enter`** needs a method with one parameter. We need `numberOfPlayers(int)`, `playerDiceRoll(int)` methods in `GoCellAward`.
4. The command **`check`** needs a method with no parameter and returns something. We need a `playerMoney()` method that returns an integer in `GoCellAward`.

The following is the code for this fixture.

```
public class GoCellAward extends ActionFixture {
    public void initializeGameBoard() {
    }

    public void numberOfPlayers(int number) {
    }

    public void playerDiceRoll(int diceRoll) {
    }

    public int playerMoney() {
        return 0;
    }
}
```

We can run the test now. Because this fixture is just a skeleton – it has no interaction with the system – we will have some red cells in the result. All we need to do now is filling in the necessary code to connect the fixture to the system:

```
public class GoCellAward extends ActionFixture {
    private GameMaster gameMaster;

    public void initializeGameBoard() {
        gameMaster = GameMaster.instance();
        gameMaster.reset();
        gameMaster.setGUI(new MockGUI());
        gameMaster.setGameBoard(new SimpleGameBoard());
    }

    public void numberOfPlayers(int number) {
        gameMaster.setNumberOfPlayers(number);
    }

    public void playerDiceRoll(int diceRoll) {
        gameMaster.movePlayer(0, diceRoll);
    }

    public int playerMoney() {
        return gameMaster.getCurrentPlayer().getMoney();
    }
}
```

Figure 13 shows the result of running this test.

fit.ActionFixture		
start	edu.ncsu.monopoly.GoCellAward	
press	initialize game board	
enter	number of players	1
enter	player dice roll	7
check	player money	1700
enter	player dice roll	5
check	player money	1900
enter	player dice roll	2
check	player money	1900

Figure 13: The Result of GoCellAward Acceptance Test.

6.1 Summary

Several practical tips for automated test were presented throughout this chapter. The keys for successful automated test are summarized in Table 3.

Table 3 Key Ideas for Automated Test

🔑	Download and learn to use the JUnit and the FIT testing frameworks. If you don't code in Java, these tools are available for other languages.
🔑	Running automatic tests often will help you see if your new code broke any existing functionality. Collect all the tests from the entire time for the entire code base. Run these tests often – at least once per day.
🔑	Use the "key ideas" in black box testing (Chapter 14 "Testing Overview and Black Box Testing Techniques) and white box testing (Chapter 15 "White Box Testing Techniques") to create your automated tests.
🔑	In automating tests, consider the advice in the Test Automation Manifesto.
🔑	When a defect is found in your code, add automated tests to reveal the defect. Then, fix the defect and re-run the automated tests to make sure they all pass now.
🔑	Work with your customer to create acceptance tests – then automate them. You can use the number (or percent) of acceptance test cases that pass as the means of determining the progress of your project.

The XP software development method uses two forms of automated testing -- white box unit tests that support the TDD practice and black box acceptance tests. In this chapter, you learned how to create both of these types of automated test cases. You can use these techniques and tools to develop automated tests in any software development process. These tests can help you identify defects in your code, can be used to ensure new changes don't cause problems with previously-working code, and can be used to help track project status. Remember that writing automated tests can be expensive, so be reasonable with the investment you make in automated tests.

Glossary of Chapter Terms

Word	Definition	Source
acceptance test	formal testing conducted to determine whether or not a system satisfies its acceptance criteria (the criteria the system must satisfy to be accepted by a customer) and to enable the customer to determine whether or not to accept the system	(IEEE, 1990)
Mock object	debug replacement for a real-world object	(Hunt and Thomas, 2003)
Regression testing	selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements	(IEEE, 1990)
white box testing	testing that takes into account the internal mechanism of a system or component	(IEEE, 1990)

References

Astels, D., G. Miller, et al. (2002). A Practical Guide to Extreme Programming. Upper Saddle River, NJ, Prentice Hall.
Beck, K. (2003). Test Driven Development -- by Example. Boston, Addison Wesley.
Beizer, B. (1990). Software Testing Techniques. London, International Thompson Computer Press.
Craig, R. D. and S. P. Jaskiel (2002). Systematic Software Testing. Norwood, MA, Artech House.
Crispen, L. and T. House (2003). Testing Extreme Programming. Boston, MA, Addison Wesley Pearson Education.
Dustin, E., J. Rashka, et al. (1999). Automated Software Testing. Reading, Massachusetts, Addison Wesley.
George, B. and L. Williams (2003). "A Structured Experiment of Test-Driven Development." Information and Software Technology (IST) **46**(5): 337-342.
Hunt, A. and D. Thomas (2003). Pragmatic Unit Testing in Java with JUnit. Raleigh, NC, The Pragmatic Bookshelf.
IEEE (1990). IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.
Kaner, C., J. Bach, et al. (2002). Lessons Learned in Software Testing. New York, John Wiley and Sons, Inc.
Martin, R. C. (2003). Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River, NJ, Prentice Hall.

Meszaros, G., S. M. Smith, et al. (2003). The Test Automation Manifesto. Extreme Programming and Agile Methods -- XP/Agile Universe 2003, Lncs 2753. F. Maurer and D. Wells. Berlin, Springer.

Müller, M. M. and O. Hagner (2002). Experiment about Test-first Programming. Conference on Empirical Assessment in Software Engineering (EASE).

Williams, L., E. M. Maximilien, et al. (2003). Test-Driven Development as a Defect-Reduction Practice. IEEE International Symposium on Software Reliability Engineering, Denver, CO, IEEE Computer Society.