Lecture 03

ECE 1145: Software Construction and Evolution

Test Driven Development (CH 2, 4, 5)

Announcements

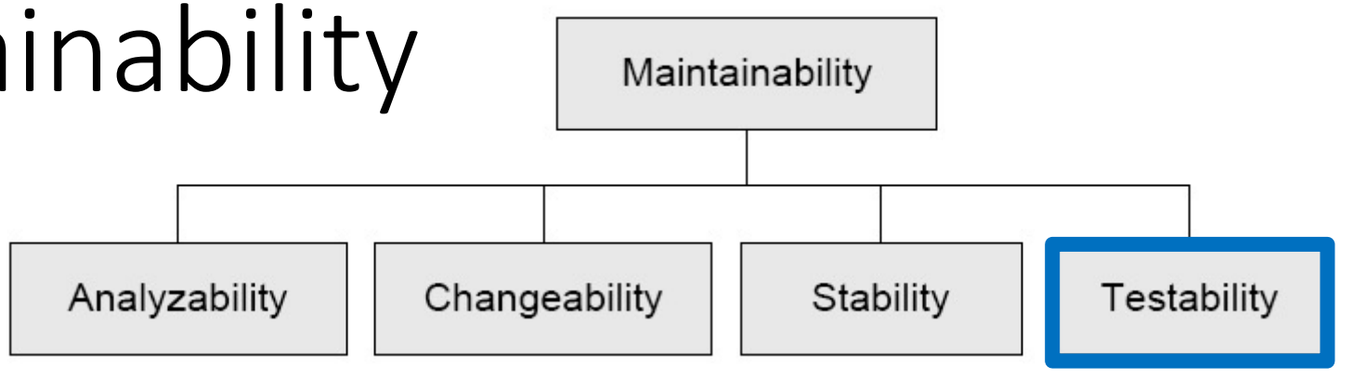
- Iteration 0 due Sept. 12
  - Teams assigned – **contact me with any issues before Friday Sept. 10**
  - Create one GitHub repository for your team, each team member should clone it locally
- Iteration 1 due Sept. 19
- Relevant Exercises: 2.3 (2.4) (5.2) (5.3) 5.5 **36.4**
  - **Most important**
  - Important
  - (Less important)

Questions for Today

How do we build reliable software?

How do we ensure that software is testable?

Recall: Maintainability



Definition: Testability (ISO 9126)

The capability of the software product to enable a modified system to be validated.

Can we **verify** the code? (with tests)

```
/** A testing tool written from scratch.
*/
public class TestDayOfWeek {
  public static void main(String[] args) {
    // Test that December 25th 2010 is Saturday
    Date d = new Date(2010, 12, 25); // year, month, day of month
    Date.Weekday weekday = d.dayOfWeek();
    if ( weekday == Date.Weekday.SATURDAY ) {
      System.out.println("Test case: Dec 25th 2010: Pass");
    } else {
      System.out.println("Test case: Dec 25th 2010: FAIL");
    }
    // ... fill in more tests
  }
}
```

Reliability

Definition: Reliability (ISO 9126)

The capability of the software product to maintain a specified level of performance when used under specified conditions.

How do we build reliable software?

Reliability

Definition: Reliability (ISO 9126)

The capability of the software product to maintain a specified level of performance when used under specified conditions.

How do we build reliable software?

- Utilize programming language constructs (e.g., variable scope) and established coding techniques (e.g., patterns)
- **Code review** – ensure understandable code and documentation (more later)
- **Test** – execute software to find situations where it does not perform its required function (try to break it!)
- **Test-Driven Development** – quickly produce reliable and maintainable software through a well-structured programming process

Test-Driven Development

Tests can be considered a **specification** of desired behavior.

Test-Driven Development is a structured programming technique focused on using tests to ensure continued reliability, especially with changes to the code.

Test-Driven Development: Concepts

Speed

- Frequent releases, updates, etc.
- “Continuous integration”
- Not rushing, but rather focusing only on necessary functionality

Simplicity

- Maximize the work **not done**!
- Avoid: “I can make a wonderful recursive solution for situations X, Y and Z … that will never ever occur in practice”

Test-Driven Development: Concepts

Take small steps

- Backtrack easily
- Sometimes means writing code that will only be used temporarily

Keep focus

- Make one thing at a time!
- Avoid: “Fixing this requires fixing that, hey this could be smarter, I’ll fix it too while I’m here, oh I could make a new class for this … “ etc., etc.
- Use TODOs and your test list (more today)

Test-Driven Development (TDD)

The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

(6. All tests pass again after refactoring!)

Case: PayStation

Welcome to **PayStation LLC!**

Customer: AlphaTown

Task: Program a pay station

http://pittsburghparking.com/

Case: PayStation

Welcome to **PayStation LLC!**

Customer: AlphaTown

Task: Program a pay station

http://pittsburghparking.com/

Requirements

- accept coins for payment
  - 5, 10, 25 c.
- show time bought on display
- print parking time receipts
- 2 minutes costs 5 c.
- handle buy and cancel

Case: PayStation

Story 1: Buy a parking ticket. A car driver walks to the pay station to buy parking time. He enters several valid coins (5, 10, and 25 cents) as payment. For each payment of 5 cents he receives 2 minutes parking time. On the pay station's display he can see how much parking time he has bought so far. Once he is satisfied with the amount of time, he presses the button marked "Buy". He receives a printed receipt, stating the number of minutes parking time he has bought. The display is cleared to prepare for another transaction.

Story 2: Cancel a transaction. A driver has entered several coins but realize that the accumulated parking time shown in the display exceeds what she needs. She presses the button marked "Cancel" and her coins are returned. The display is cleared to prepare for another transaction.

Story 3: Reject illegal coin. A driver has entered 50 cents total and the display reads "20". By mistake, he enters a 1 euro coin which is not a recognized coin. The pay station rejects the coin and the display is not updated.

Case: PayStation

Requirements

- accept coins for payment
  - 5, 10, 25 c.
- show time bought on display
- print parking time receipts
- 2 minutes costs 5 c.
- handle buy and cancel

Test list

* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

Case: PayStation

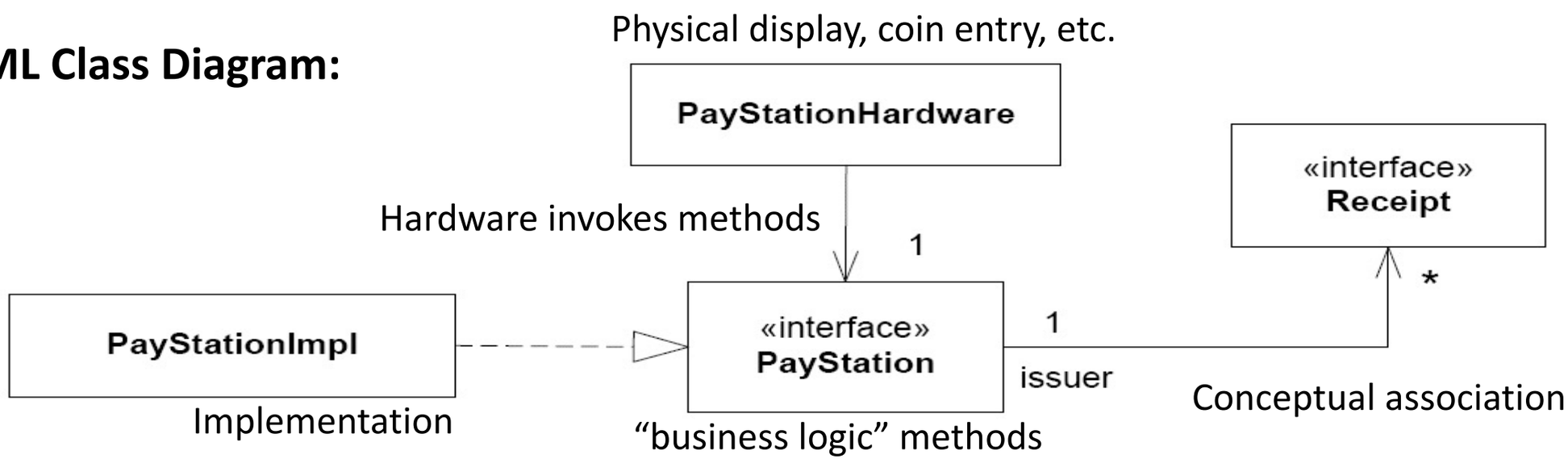
Requirements

- accept coins for payment
  - 5, 10, 25 c.
- show time bought on display
- print parking time receipts
- 2 minutes costs 5 c.
- handle buy and cancel

Test list

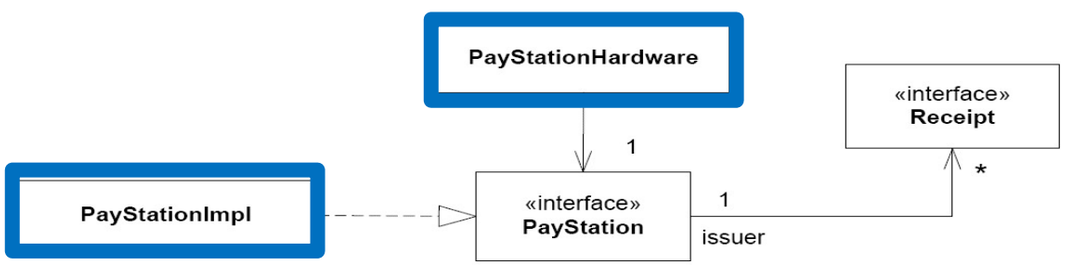
* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

UML Class Diagram:

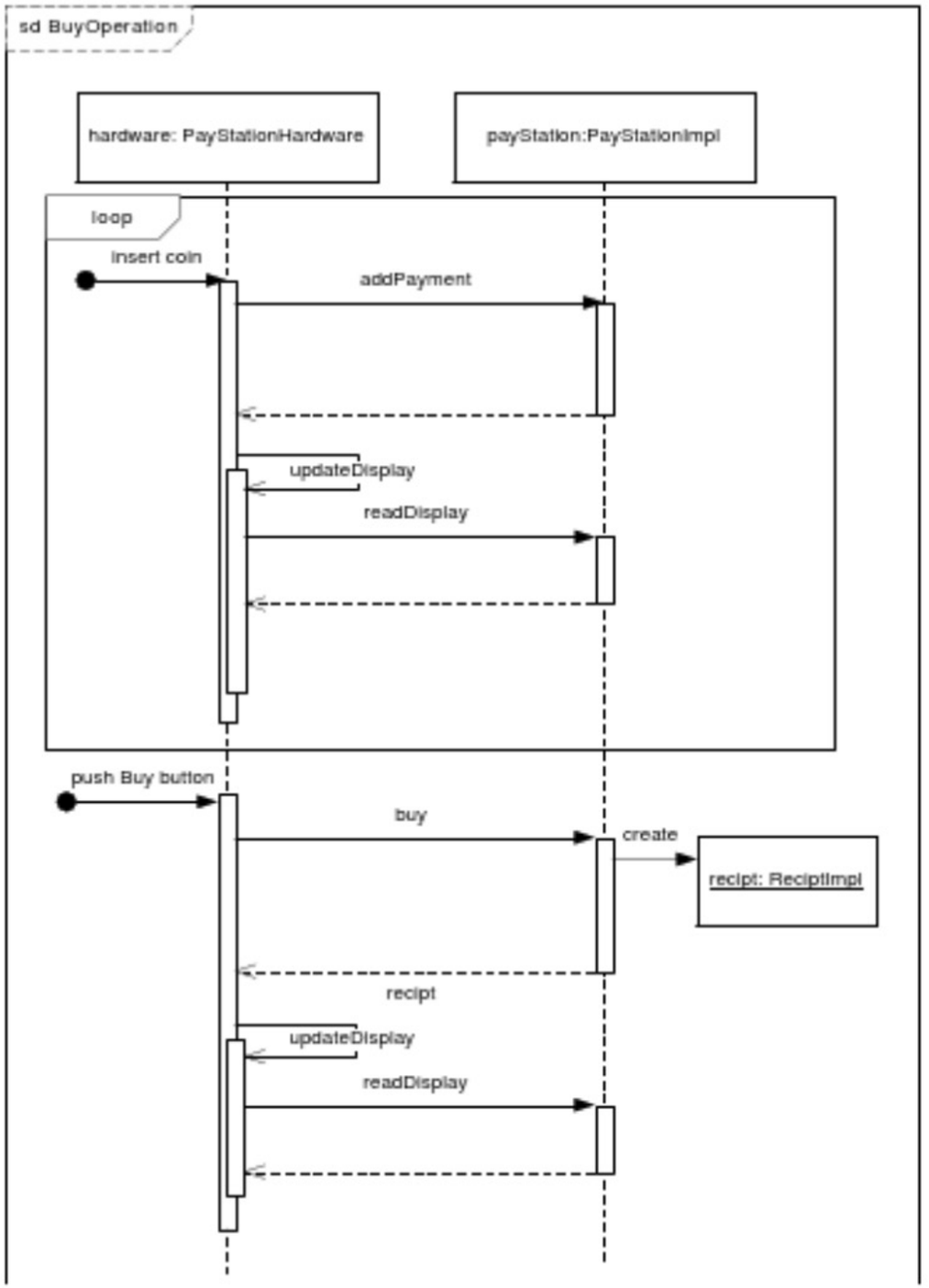


Case: PayStation

- accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin
- readDisplay
- buy produces valid receipt
- cancel resets pay station



UML Sequence Diagram:



Case: PayStation

- accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin
- readDisplay
- buy produces valid receipt
- cancel resets pay station

PayStation.java

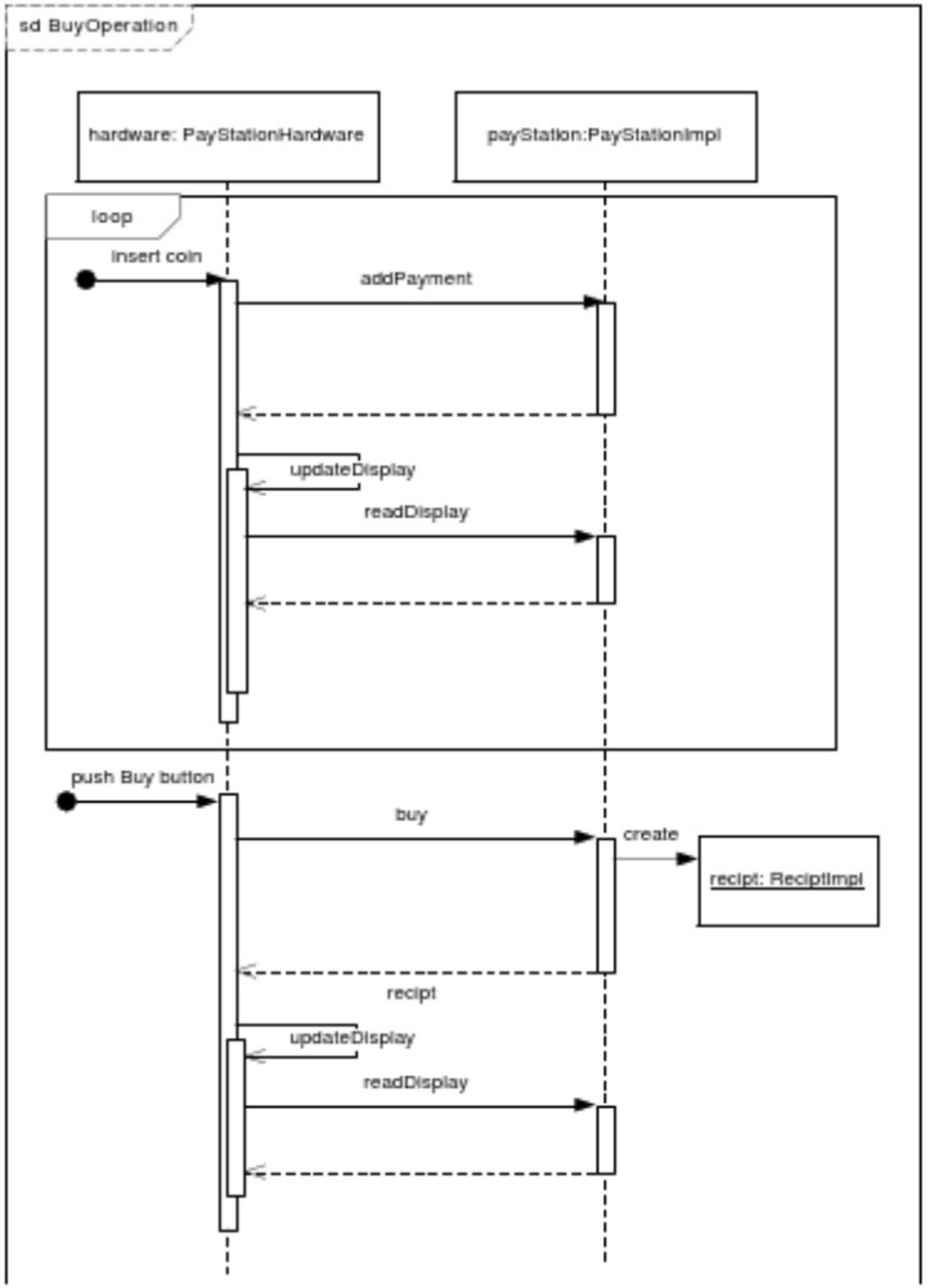
```
/** The business logic of a Parking Pay Station.
*/
public interface PayStation {

  /**
   * Insert coin into the pay station and adjust state accordingly.
   * @param coinValue is an integer value representing the coin in
   * cent. That is, a quarter is coinValue=25, etc.
   * @throws IllegalCoinException in case coinValue is not
   * a valid coin value
   */
  public void addPayment( int coinValue )
          throws IllegalCoinException;

  /**
   * Read the machine's display. The display shows a numerical
   * description of the amount of parking time accumulated so far
   * based on inserted payment.
   * @return the number to display on the pay station display
   */
  public int readDisplay();

  /**
   * Buy parking time. Terminate the ongoing transaction and
   * return a parking receipt. A non-null object is always returned.
   * @return a valid parking receipt object.
   */
  public Receipt buy();

  /**
   * Cancel the present transaction. Resets the machine for a new
   * transaction.
   */
  public void cancel();
}
```



Javadoc

https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

Tags identify aspects of documentation:

@param : method parameter (name and description)

@return : method return value

PayStation.java (Interface)

```
/** The business logic of a Parking Pay Station.
*/
public interface PayStation {

  /**
   * Insert coin into the pay station and adjust state accordingly.
   * @param coinValue is an integer value representing the coin in
   * cent. That is, a quarter is coinValue=25, etc.
   * @throws IllegalCoinException in case coinValue is not
   * a valid coin value
   */
  public void addPayment( int coinValue )
          throws IllegalCoinException;

  /**
   * Read the machine's display. The display shows a numerical
   * description of the amount of parking time accumulated so far
   * based on inserted payment.
   * @return the number to display on the pay station display
   */
  public int readDisplay();

  /**
   * Buy parking time. Terminate the ongoing transaction and
   * return a parking receipt. A non-null object is always returned.
```

Javadoc

Preconditions:

Describe the method's input arguments, mention any global constants that it uses, and list any assumptions that it makes

Postconditions:

Describe changes the module has made

Game.java (Interface)

```
/** move a unit from one position to another. If that other position
 * is occupied by an opponent unit, a battle is conducted leading to
 * either victory or defeat. If victorious then the opponent unit is
 * removed from the game and the move conducted. If defeated then
 * the attacking unit is removed from the game. If a successful move
 * results in the unit entering the position of a city then this
 * city becomes owned by the owner of the moving unit.
 * Precondition: both from and to are within the limits of the
 * world.  Precondition: there is a unit located at position from.
 * @param from the position that the unit has now
 * @param to the position the unit should move to
 * @return true if the move is valid (no mountain, move is valid
 * under the rules of the game variant etc.), and false
 * otherwise. If false is returned, the unit stays in the same
 * position and its "move" is intact (it can be moved to another
 * position.)
 */
public boolean moveUnit( Position from, Position to );
```

Iteration 1

Case: PayStation

Test list

* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

Iteration 1

Case: PayStation

Test list

* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

TDD Principle: **One Step Test**

Which test should you pick next from the test list? Pick a test that will teach you something and that you are confident you can implement.

Iteration 1

Case: PayStation

- accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin
- readDisplay
- buy produces valid receipt
- cancel resets pay station

PayStationImpl.java

```
/** Implementation of the pay station.
*/
public class PayStationImpl implements PayStation {

  public void addPayment( int coinValue )
          throws IllegalCoinException {
  }

  public int readDisplay() {
    return 0;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

← Implementation here,

according to method definitions in interface →

PayStation.java

```
/** The business logic of a Parking Pay Station.
*/
public interface PayStation {

  /**
   * Insert coin into the pay station and adjust state accordingly.
   * @param coinValue is an integer value representing the coin in
   * cent. That is, a quarter is coinValue=25, etc.
   * @throws IllegalCoinException in case coinValue is not
   * a valid coin value
   */
  public void addPayment( int coinValue )
          throws IllegalCoinException;

  /**
   * Read the machine's display. The display shows a numerical
   * description of the amount of parking time accumulated so far
   * based on inserted payment.
   * @return the number to display on the pay station display
   */
  public int readDisplay();

  /**
   * Buy parking time. Terminate the ongoing transaction and
   * return a parking receipt. A non-null object is always returned.
```

TestPayStation.java

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

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

← Tests go here

Iteration 1

Case: PayStation

* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

Display 2 minutes if input value is 5 cents (addPayment)

TestPayStation.java

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

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

1. Add a test

Iteration 1

Case: PayStation

* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

2. See the test fail (as expected)!

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

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5
                  2, ps.readDisplay() );
  }
}
```

```
JUnit version 4.4
.E
Time: 0,047
There was 1 failure:
1) shouldDisplay2MinFor5Cents(TestPayStation)
java.lang.AssertionError: Should display 2 min for 5 cents
  expected:<2> but was:<0>
     at org.junit.Assert.fail(Assert.java:74)
     at org.junit.Assert.failNotEquals(Assert.java:448)
     at org.junit.Assert.assertEquals(Assert.java:102)
     at org.junit.Assert.assertEquals(Assert.java:323)
     at TestPayStation.shouldDisplay2MinFor5Cents(Test
                                                  PayStation.java:20)
  [lines removed here]
     at org.junit.runner.JUnitCore.main(JUnitCore.java:44)

FAILURES!!!
Tests run: 1,  Failures: 1
```

Iteration 1

Case: PayStation

* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

3. Make a little change

What is the simplest possible PayStation class implementation that will cause the test to pass?

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

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

Iteration 1

Case: PayStation

* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

3. Make a little change

What is the simplest possible PayStation class implementation that will cause the test to pass?

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

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

```
/** Implementation of the pay station.
*/
public class PayStationImpl implements PayStation {

  public void addPayment( int coinValue )
          throws IllegalCoinException {
  }

  public int readDisplay() {
    return 0;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

PayStationImpl

Iteration 1

Case: PayStation

* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

3. Make a little change

What is the simplest possible PayStation class implementation that will cause the test to pass?

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

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

```
/** Implementation of the pay station.
*/
public class PayStationImpl implements PayStation {

  public void addPayment( int coinValue )
          throws IllegalCoinException {
  }

  public int readDisplay() {
    return 2;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

PayStationImpl

Iteration 1

Case: PayStation

* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

3. Make a little change

What is the simplest possible PayStation class implementation that will cause the test to pass?

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

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

```
/** Implementation of the pay station.
*/
public class PayStationImpl implements PayStation {

  public void addPayment( int coinValue )
          throws IllegalCoinException {
  }

  public int readDisplay() {
    return 2;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

PayStationImpl

TDD Principle: Fake It ('Til You Make It)

What is your first implementation once you have a broken test? Return a constant. Once you have your tests running, gradually transform it.

Iteration 1

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* *25 cents = 10 minutes*

TDD Principle: **Triangulation**

How do you most conservatively drive abstraction with tests? Abstract only when you have two or more examples.

You should have two, three, or more examples in order to generalize and drive an abstraction like an algorithm or class into existence

→ Wait to implement the parking time calculation until it is necessary

→ Add a test case to the test list as a reminder

Iteration 2

Case: PayStation

1. Add a test

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* *25 cents = 10 minutes*

```
@Test
public void shouldDisplay2MinFor5Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 5 );
  assertEquals( "Should display 2 min for 5 cents",
                2, ps.readDisplay() );

  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

Iteration 2

Case: PayStation

1. Add a test

- ~~accept legal coin~~
- reject illegal coin, exception
- ~~5 cents should give 2 minutes parking time.~~
- ~~readDisplay~~
- buy produces valid receipt
- cancel resets pay station
- *25 cents = 10 minutes*

```
@Test
public void shouldDisplay2MinFor5Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 5 );
  assertEquals( "Should display 2 min for 5 cents",
                2, ps.readDisplay() );

  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

```
@Test
public void shouldDisplay10MinFor25Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

TDD Principle: Isolated Test

How should the running of tests affect one another? Not at all.

Iteration 2

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* *25 cents = 10 minutes*

```
@Test
public void shouldDisplay10MinFor25Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

2. See the test fail

```
1) shouldDisplay10MinFor25Cents(TestPayStation)
java.lang.AssertionError: Should display 10 min for 25 cents
  expected:<10> but was:<2>
  [lines omitted]
  at TestPayStation.shouldDisplay10MinFor25Cents
                                    (TestPayStation.java:32)
  [lines omitted]
```

Iteration 2

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* *25 cents = 10 minutes*

```
@Test
public void shouldDisplay10MinFor25Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

```
/** Implementation of the pay station.
*/
public class PayStationImpl implements PayStation {

  public void addPayment( int coinValue )
          throws IllegalCoinException {
  }

  public int readDisplay() {
    return 2;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  public void addPayment( int coinValue )
          throws IllegalCoinException {
    insertedSoFar = coinValue;
  }
  public int readDisplay() {
    return insertedSoFar / 5 * 2;
  }
```

3. Make a little change
4. See the tests pass

Iteration 2

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* *25 cents = 10 minutes*
* enter two or more legal coins

Key point: Production code is driven into existence by tests

In the extreme, you do not enter a single character into production code unless there is a test case that demands it.

```
@Test
public void shouldDisplay10MinFor25Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  public void addPayment( int coinValue )
          throws IllegalCoinException {
    insertedSoFar = coinValue;
  }
  public int readDisplay() {
    return insertedSoFar / 5 * 2;
  }
```

Iteration 2

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* *25 cents = 10 minutes*
* enter two or more legal coins

Definition: **Refactoring**

Refactoring is the process of modifying and restructuring the source code to improve its maintainability and flexibility without affecting the system's external behavior when executing.

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

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
```

```
@Test
public void shouldDisplay10MinFor25Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

5. Refactor to remove duplication (also applies to test code)

Iteration 2

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* *25 cents = 10 minutes*
* enter two or more legal coins

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

/** Testcases for the Pay Station system.
*/
public class TestPayStation {
  PayStation ps;
  /** Fixture for pay station testing. */
  @Before
  public void setUp() {
    ps = new PayStationImpl();
  }

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }

  /**
   * Entering 25 cents should make the display report 10 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay10MinFor25Cents()
          throws IllegalCoinException {
    ps.addPayment( 25 );
    assertEquals( "Should display 10 min for 25 cents",
                  10, ps.readDisplay() );
  }
}
```

@Before: run before each test case (@Test methods)

→ Keeps test cases independent without duplicating code

Iteration 2

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* *25 cents = 10 minutes*
* enter two or more legal coins

```
import org.junit.*;
```

TDD Principle: **Evident Data**

How do you represent the intent of the data? Include expected and actual results in the test itself, and make their relationship apparent. You are writing tests for the reader, not just for the computer.

```
  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }

  /**
   * Entering 25 cents should make the display report 10 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay10MinFor25Cents()
          throws IllegalCoinException {
    ps.addPayment( 25 );
    assertEquals( "Should display 10 min for 25 cents",
                  10, ps.readDisplay() );
  }
}
```

Iteration 2

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* *25 cents = 10 minutes*
* enter two or more legal coins

```
import org.junit.*;
```

TDD Principle: **Evident Data**

How do you represent the intent of the data? Include expected and actual results in the test itself, and make their relationship apparent. You are writing tests for the reader, not just for the computer.

```
  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }

  /**
   * Entering 25 cents should make the display report 10 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay10MinFor25Cents()
          throws IllegalCoinException {
    ps.addPayment( 25 );
    assertEquals( "Should display 10 min for 25 cents",
                  10, ps.readDisplay() );
  }
}
```

Change code or add comments (why not both?)

```
    ps.addPayment( 25 );
    assertEquals( "Should display 10 min for 25 cents",
                  25 / 5 * 2, ps.readDisplay() );
    // 25 cent in 5 cent coins each giving 2 minutes parking
```

Iteration 2

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* enter two or more legal coins

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

/** Testcases for the Pay Station system.
*/
public class TestPayStation {
  PayStation ps;
  /** Fixture for pay station testing. */
  @Before
  public void setUp() {
    ps = new PayStationImpl();
  }

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }

  /**
   * Entering 25 cents should make the display report 10 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay10MinFor25Cents()
          throws IllegalCoinException {
    ps.addPayment( 25 );
    assertEquals( "Should display 10 min for 25 cents",
                  10, ps.readDisplay() );
  }
}
```

The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

(6. All tests pass again after refactoring!)

Iteration 3

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* enter two or more legal coins

1. Add a Test

```
@Test(expected=IllegalCoinException.class)
public void shouldRejectIllegalCoin() throws IllegalCoinException {
  ps.addPayment(17);
}
```

Iteration 3

Case: PayStation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* enter two or more legal coins

1. Add a Test

```
@Test(expected=IllegalCoinException.class)
public void shouldRejectIllegalCoin() throws IllegalCoinException {
  ps.addPayment(17);
}
```

2. See the test fail

```
JUnit version 4.4
...E
Time: 0,031
There was 1 failure:
1) shouldRejectIllegalCoin(TestPayStation)
java.lang.AssertionError: Expected exception: IllegalCoinException
  [lines omitted]
FAILURES!!!
Tests run: 3,  Failures: 1
```

3. Make a little change

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  insertedSoFar = coinValue;
}
```

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar = coinValue;
}
```

4. See the tests succeed

Iteration 3

Case: PayStation

TDD Principle: **Representative Data**

What data do you use for your tests? Select a small set of data where each element represents a conceptual aspect or a special computational processing.

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* *enter ~~two or more legal~~ a 10 and 25 coin*

Iteration 4

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* *enter ~~two or more legal~~ a 10 and 25 coin*

1. Add a test: shouldDisplay14MinFor10And25Cents

2. See the test fail?

```
JUnit version 4.4
....E
Time: 0,016
There was 1 failure:
1) shouldDisplay14MinFor10And25Cents(TestPayStation)
IllegalCoinException: Invalid coin: 10
  [...]
```

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar = coinValue;
}
```

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 10: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar = coinValue;
}
```

Iteration 4

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* *enter ~~two or more legal~~ a 10 and 25 coin*

1. Add a test: shouldDisplay14MinFor10And25Cents

```
JUnit version 4.4
....E
Time: 0,016
There was 1 failure:
1) shouldDisplay14MinFor10And25Cents(TestPayStation)
IllegalCoinException: Invalid coin: 10
  [...]
```

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 10: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar = coinValue;
}
```

2. See the test fail

```
JUnit version 4.4
....E
Time: 0,016
There was 1 failure:
1) shouldDisplay14MinFor10And25Cents(TestPayStation)
java.lang.AssertionError: Should display 14 min for 10+25 cents
    expected:<14> but was:<10>
```

Iteration 4

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~

3. Make a little change

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 10: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar = coinValue;
}
```

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 10: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
}
```

4. See the test pass

Iteration 5

Case: PayStation

```
* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
```

PayStation.java

```
/**
 * Buy parking time. Terminate the ongoing transaction and
 * return a parking receipt. A non-null object is always returned.
 * @return a valid parking receipt object.
 */
public Receipt buy();
```

Receipt.java (interface)

```
/** The receipt returned from a pay station.
*/
public interface Receipt {

  /**
   * Return the number of minutes this receipt is valid for.
   * @return number of minutes parking time
  */
  public int value();
}
```

TestPayStation.java

```
@Test
public void shouldReturnCorrectReceiptWhenBuy()
        throws IllegalCoinException {
  ps.addPayment(5);
  ps.addPayment(10);
  ps.addPayment(25);
  Receipt receipt;
  receipt = ps.buy();
  assertNotNull( "Receipt reference cannot be null",
                 receipt );
  assertEquals( "Receipt value must be 16 min.",
                (5+10+25) / 5 * 2 , receipt.value() );
}
```

1. Add a test

Iteration 5

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~

PayStation.java

```
/**
 * Buy parking time. Terminate the ongoing transaction and
 * return a parking receipt. A non-null object is always returned.
 * @return a valid parking receipt object.
 */
public Receipt buy();
```

Receipt.java (interface)

```
/** The receipt returned from a pay station.
*/
public interface Receipt {

  /**
   * Return the number of minutes this receipt is valid for.
   * @return number of minutes parking time
  */
  public int value();
}
```

TestPayStation.java

```
@Test
public void shouldReturnCorrectReceiptWhenBuy()
        throws IllegalCoinException {
  ps.addPayment(5);
  ps.addPayment(10);
  ps.addPayment(25);
  Receipt receipt;
  receipt = ps.buy();
  assertNotNull( "Receipt reference cannot be null",
                 receipt );
  assertEquals( "Receipt value must be 16 min.",
                (5+10+25) / 5 * 2 , receipt.value() );
}
```

1. Add a test

Why both?

Iteration 5

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~

PayStation.java

```
/**
 * Buy parking time. Terminate the ongoing transaction and
 * return a parking receipt. A non-null object is always returned.
 * @return a valid parking receipt object.
 */
public Receipt buy();
```

Receipt.java (interface)

```
/** The receipt returned from a pay station.
*/
public interface Receipt {

  /**
   * Return the number of minutes this receipt is valid for.
   * @return number of minutes parking time
  */
  public int value();
}
```

TestPayStation.java

```
@Test
public void shouldReturnCorrectReceiptWhenBuy()
        throws IllegalCoinException {
  ps.addPayment(5);
  ps.addPayment(10);
  ps.addPayment(25);
  Receipt receipt;
  receipt = ps.buy();
  assertNotNull( "Receipt reference cannot be null",
                 receipt );
  assertEquals( "Receipt value must be 16 min.",
                (5+10+25) / 5 * 2 , receipt.value() );
}
```

1. Add a test

Why both?

Helps with debugging

Iteration 5

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~

```
@Test
public void shouldReturnCorrectReceiptWhenBuy()
        throws IllegalCoinException {
  ps.addPayment(5);
  ps.addPayment(10);
  ps.addPayment(25);
  Receipt receipt;
  receipt = ps.buy();
  assertNotNull( "Receipt reference cannot be null",
                 receipt );
  assertEquals( "Receipt value must be 16 min.",
                (5+10+25) / 5 * 2 , receipt.value() );
}
```

2. See the test fail

```
1) shouldReturnCorrectReceiptWhenBuy(TestPayStation)
java.lang.AssertionError: Receipt reference should not be null
```

Iteration 5

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy for 40 cents produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* *receipt can store values*
* *buy for 100 cents*

```
@Test
public void shouldReturnCorrectReceiptWhenBuy()
        throws IllegalCoinException {
  ps.addPayment(5);
  ps.addPayment(10);
  ps.addPayment(25);
  Receipt receipt;
  receipt = ps.buy();
  assertNotNull( "Receipt reference cannot be null",
                 receipt );
  assertEquals( "Receipt value must be 16 min.",
                (5+10+25) / 5 * 2 , receipt.value() );
}
```

```
1) shouldReturnCorrectReceiptWhenBuy(TestPayStation)
java.lang.AssertionError: Receipt reference should not be null
```

3. Make a little change

TDD Principle: Fake It ('Til You Make It)

What is your first implementation once you have a broken test? Return a constant. Once you have your tests running, gradually transform it.

Iteration 5

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy for 40 cents produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* *receipt can store values*
* *buy for 100 cents*

```
@Test
public void shouldReturnCorrectReceiptWhenBuy()
        throws IllegalCoinException {
  ps.addPayment(5);
  ps.addPayment(10);
  ps.addPayment(25);
  Receipt receipt;
  receipt = ps.buy();
  assertNotNull( "Receipt reference cannot be null",
                 receipt );
  assertEquals( "Receipt value must be 16 min.",
                (5+10+25) / 5 * 2 , receipt.value() );
}
```

```
1) shouldReturnCorrectReceiptWhenBuy(TestPayStation)
java.lang.AssertionError: Receipt reference should not be null
```

3. Make a little change in PayStationImpl.java

```
public Receipt buy() {
  return new Receipt() {
      public int value() { return (5+10+25)/5*2; }
    };
}
```

Faking it: Create a new instance of an **anonymous class** that implements Receipt and has this implementation of the value method

4. See the tests pass

Iteration 6

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* ~~buy for 40 cents produces valid receipt~~
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* *receipt can store values*
* *buy for 100 cents*

1. Add a test

TDD Principle: Assert First

When should you write the asserts? Try writing them first.

```
@Test
public void shouldStoreTimeInReceipt() {
  ...
  assertEquals( "Receipt can store 30 minute value",
                30, receipt.value() );
}
```

```
@Test
public void shouldStoreTimeInReceipt() {
  Receipt receipt = new ReceiptImpl(30);
  assertEquals( "Receipt can store 30 minute value",
                30, receipt.value() );
}
```

Iteration 6

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* ~~buy for 40 cents produces valid receipt~~
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* *receipt can store values*
* *buy for 100 cents*

TDD Principle: Assert First

When should you write the asserts? Try writing them first.

```
@Test
public void shouldStoreTimeInReceipt() {
  ...
  assertEquals( "Receipt can store 30 minute value",
                30, receipt.value() );
}
```

```
@Test
public void shouldStoreTimeInReceipt() {
  Receipt receipt = new ReceiptImpl(30);
  assertEquals( "Receipt can store 30 minute value",
                30, receipt.value() );
}
```

2. See the test fail

```
TestPayStation.java:81: cannot find symbol
symbol  : class ReceiptImpl
location: class TestPayStation
    Receipt receipt = new ReceiptImpl(30);
                          ^
1 error
```

Iteration 6

Case: PayStation

```
@Test
public void shouldStoreTimeInReceipt() {
  Receipt receipt = new ReceiptImpl(30);
  assertEquals( "Receipt can store 30 minute value",
                30, receipt.value() );
}
```

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* ~~buy for 40 cents produces valid receipt~~
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* *receipt can store values*
* *buy for 100 cents*

3. Make a little change

ReceiptImpl.java

```
/** Implementation of Receipt.
*/

public class ReceiptImpl implements Receipt {
  private int value;
  public ReceiptImpl(int value) { this.value = value; }
  public int value() { return value;}
}
```

TDD Principle: Obvious Implementation

How do you implement simple operations? Just implement them.

When implementation is simple, we can skip fake-it and triangulation

4. See the tests pass

Iteration 7

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* ~~buy for 40 cents produces valid receipt~~
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* ~~receipt can store values~~
* buy for 100 cents

1. Add a test

```
@Test
public void shouldReturnReceiptWhenBuy100c()
  throws IllegalCoinException {
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(25);
  ps.addPayment(25);

  Receipt receipt;
  receipt = ps.buy();
  assertEquals((5*10+2*25) / 5 * 2 , receipt.value() );
}
```

Iteration 7

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* ~~buy for 40 cents produces valid receipt~~
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* ~~receipt can store values~~
* buy for 100 cents

1. Add a test

```
@Test
public void shouldReturnReceiptWhenBuy100c()
  throws IllegalCoinException {
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(25);
  ps.addPayment(25);

  Receipt receipt;
  receipt = ps.buy();
  assertEquals((5*10+2*25) / 5 * 2 , receipt.value() );
}
```

TDD Principle: Evident Tests

How do we avoid writing defective tests? By keeping the testing code evident, readable, and as simple as possible.

Test cases should ideally only contain assignments, method calls, and assertions (maybe very simple loops, definitely not recursion). Private helper methods are okay

Iteration 7

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* ~~buy for 40 cents produces valid receipt~~
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* ~~receipt can store values~~
* buy for 100 cents

2. See tests fail

3. Make a little change

```
public Receipt buy() {
  return new Receipt() {
      public int value() { return (5+10+25)/5*2; }
    };
}
```

```
public Receipt buy() {
  return new ReceiptImpl(insertedSoFar * 2 / 5);
}
```

4. See tests pass

Iteration 7

Case: PayStation

5. Refactor

```
* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* ~~buy for 40 cents produces valid receipt~~
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* ~~receipt can store values~~
* buy for 100 cents
```

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 10: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
}
```

```
public int readDisplay() {
  return insertedSoFar * 2 / 5;
}
public Receipt buy() {
  return new ReceiptImpl(insertedSoFar * 2 / 5);
}
```

```
/** Implementation of the pay station.
*/

public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

  public void addPayment( int coinValue )
          throws IllegalCoinException {
    switch ( coinValue ) {
    case 5: break;
    case 10: break;
    case 25: break;
    default:
      throw new IllegalCoinException("Invalid coin: "+coinValue);
    }
    insertedSoFar += coinValue;
    timeBought = insertedSoFar / 5 * 2;
  }
  public int readDisplay() {
    return timeBought;
  }
  public Receipt buy() {
    return new ReceiptImpl(timeBought);
  }
  public void cancel() {
  }
}
```

Rerun tests after refactoring!

TDD Notes

Testing and Test-Driven development cannot prove the absence of defects, only the **existence** of one.

Testing code must follow the invariants, preconditions, and specifications of the classes it tests, provide proper and valid parameters and call methods in the correct sequence, etc.

- Otherwise, it will break in later iterations

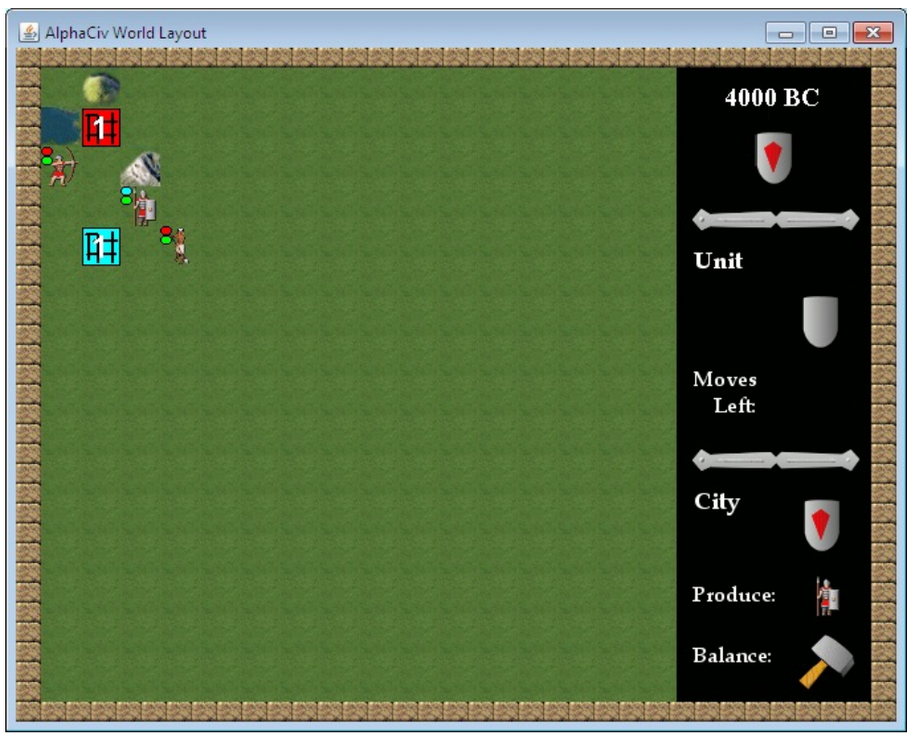
If an interface needs to be changed, all the tests do too.

- Don’t just throw them away though – update/refactor!

Iteration 1: Test-Driven Development I

AlphaCiv simplifications (from 36.1):

- Two players: Red and Blue
- All tiles are Plains except (1,0) is Ocean, (0,1) is Hills, and (2,2) is Mountains
- Only one unit allowed on a tile
- Red starts with an Archer and a Settler, Blue starts with a Legion
- Units do not have actions, and attacking units always win
- Cities are always population/size 1, and gain 6 production each round (Blue and Red have one city each)
- The game starts at 4000 BC and ages 100 years each round
- Red wins in 3000 BC



Next time: Configuration management