Chapter 5
Unit Testing

This chapter covers:

Concepts and Principles: Unit testing, regression testing, quality attributes of unit tests, test suites, test coverage;
Programming Mechanisms: Unit testing frameworks, JUnit, metaprogramming, annotations;
Design Techniques: Test suite organization, use of test fixtures, testing with stubs, testing private structures, use of test coverage metrics, testing exceptional conditions;
Patterns and Antipatterns: DUPLICATED CODE†.

5.1 Intro

A unit test consists of one or more executions of a *unit under test (UUT)* with some input data and the comparison of the result of the execution against some *oracle*. A UUT is whatever piece of the code we wish to test in isolation. UUTs are often methods, but in some cases they can also be entire classes, initialization statements, or certain paths through the code. The term oracle designates the correct or expected result of the execution of a UUT.
The comparison of the result of executing the UUT with the oracle is also called an *assertion*.

e.g. Tthe statement: `Math.abs(5) == 5;` technically qualifies as a test. Here the UUT is the library method `Math.abs(int)`, the input data is the integer literal `5`, and the oracle is, in this case, also the value `5`.

When testing **non-static** methods, it is important to remember that the input data includes the *implicit argument* (the object that receives the method call).

```
public enum Suit
{
  CLUBS, DIAMONDS, SPADES, HEARTS;

  public boolean sameColorAs(Suit pSuit)
  {
    assert pSuit != null;
    return (ordinal() - pSuit.ordinal()) % 2 == 0;
  }
}
```

```
public static void main(String[] args)
{
  boolean oracle = false;
  System.out.println(oracle == CLUBS.sameColorAs(HEARTS));
}
```

implicit argument

This main method qualifies as a unit test: it includes a UUT (Suit.sameColorAs), some input data (CLUBS and HEARTS), an oracle (**false**), and an assertion that compares the result with the oracle.

If we reorder the suits as `CLUBS, SPADES, DIAMONDS, HEARTS`; In this case, running the test again will immediately reveal a bug introduced by the fact that `sameColorAs` relies on an undocumented and unchecked assumption about the order of enumerated values.

This example illustrates the second major benefit of unit tests: in addition to helping detect bugs in new code, they can also check that tested behavior that used to meet some specific expectation still does meet that expectation even after the code changes. Running tests to ensure what was tested as correct still is (or for detecting new bugs caused by changes) is called regression testing.

Note: Unit testing cannot verify code to be correct! When a test passes, it only shows that the one specific execution of the code that is being tested behaves as expected.

5.2 Unit Testing Framework Fundamentals

The major constructs supported by testing frameworks are *test cases, test suites, test fixtures* and *assertions.*

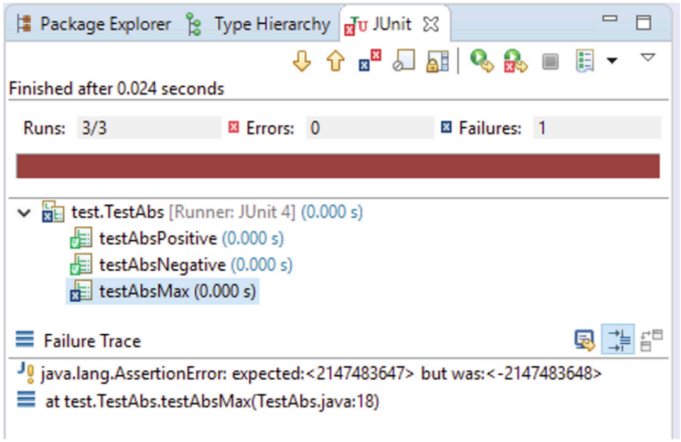
```
public class AbsTest
{
  @Test
  public void testAbs_Positive()
  { assertEquals(5, Math.abs(5)); }

  @Test
  public void testAbs_Negative()
  { assertEquals(5, Math.abs(-5)); }

  @Test
  public void testAbs_Max()
  { assertEquals(Integer.MAX_VALUE,Math.abs(Integer.MIN_VALUE)); }
}
```

The `@Test` *annotation instance* indicates that the annotated method should be run as a unit test. The code example above shows a *test class* that defines three tests, all intended to test the library method `Math.abs(int)`.

To constitute proper tests, test methods should contain at least one execution of a unit under test. The way to automatically verify that the execution of a unit under test has the expected effect is to execute various calls to *assert methods*. Assert methods are different from the **assert** statement in Java. They are declared as static methods of the class `org.junit.Assert` and all they do is basically verify a predicate and, if the predicate is false, report a *test failure*. The JUnit framework includes a graphical user interface component called a *test runner*, which automatically scans some input code, detects all the tests in the input, executes them, and then reports whether the tests passed or failed.



[Javadoc]
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

The reason for this design choice is imposed by physical constraints: because one bit of information is used to encode 0, there is simply no space left available to encode the absolute value of `Integer.MIN_VALUE` in a 32-bit int type.

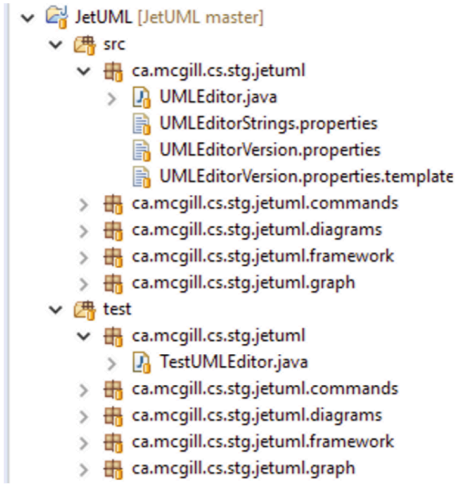
5.3 Organizing Test Code

A collection of tests for a project is known as a *test suite*.
By default, a project's test suite consists of all the unit tests for the production code in the project. However, for various reasons, it may be desirable to run only a subset of different tests at different times (for example, to focus on a specific feature, or save some time).

Common approaches:

- One test class per project class, where the test class collects all the tests that test methods or other usage scenarios that involve the class.

- Locate all the testing code in a different source folder with a package structure that mirrors the package structure of the production code.
  [The rationale for this organization is that in Java classes with the same package name are in the same *package scope* independently of their location in a file system. This means that classes and methods in the test package can refer to non-public (but non-private) members of classes in the production code, while still being separated from the production code.]



5.4 Metaprogramming

In the previous section, we saw that to mark a method as a test, we annotate it with the string `@Test`. The unit testing framework can then rely on this annotation to detect which methods are tests, and then proceed to execute these methods as part of the execution of the test runner. This general approach, employed by most unit testing frameworks, is a bit special in that in requires the code to manipulate other code. Specifically, the testing framework first scans the code to detect tests, and then executes the code, without having any explicit code for calls to test methods.
This strategy is an illustration of a general programming feature called metaprogramming. The idea of metaprogramming is to *write code that operates on a representation of a program's code*
In Java, metaprogramming is called reflection, and library support for metaprogramming features is available through the class `java.lang.Class` and the package `java.lang.reflect`.

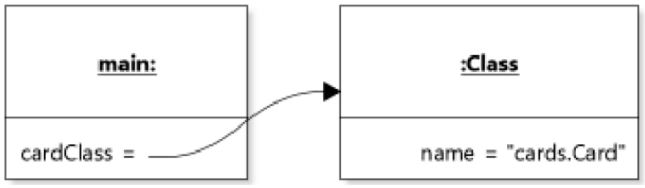
Introspection

The most basic metaprogramming task is to obtain a *reference* to an object that represents a piece of code to learn about it, a procedure called introspection. In Java, the class `Class<T>` is the main access point for metaprogramming.

a) `forName` library method

```
try
{
  String fullyQualifiedName = "cards.Card";
  Class<Card> cardClass =
    (Class<Card>) Class.forName(fullyQualifiedName);
}
catch(ClassNotFoundException e) { e.printStackTrace(); }
```

The call to `Class.forName` returns a reference to an instance of class `Class` that represents class `Card`



Note: any string can be supplied as argument to Class.forName

```
public static void main(String[] args)
{
  try
  { Class<?> theClass = Class.forName(args[0]); }
  catch(ClassNotFoundException e) { e.printStackTrace(); }
}
```

type wildcard as the instance of the type parameter in the type declaration of the variable that receives the reference supplied by forName.

b) class literals

In Java, a *class literal* is a literal expression that consists of the name of a class followed by the suffix . `class`, and that refers to a reference to an instance of class Class that represents the class named before the suffix. So, `Card.class` refers to the instance of class `Class` that represents class `Card`. Because, in the case of class literals, the argument T to the type parameter of `Class<T>` is guaranteed to be known at compile time, we can include it in the variable declaration. Class literals are the least brittle way to obtain a reference to an instance of class `Class`, but they require that we know the exact class to instrospect at compile time.

```
Class<Card> cardClass1 = Card.class;
```

c) an instance of the class of interest.

It is possible to call method `getClass()` on any object in a Java program, and the method will return a reference to an instance of class `Class` that represents the run-time type of the object. Because of polymorphism, this type is not known at compile time, so in this case also we have to use the *type wildcard* in the declaration of the variable. However, because any call to `getClass()` is guaranteed to return a valid reference to an instance of `Class`, the method does not declare to throw an exception.

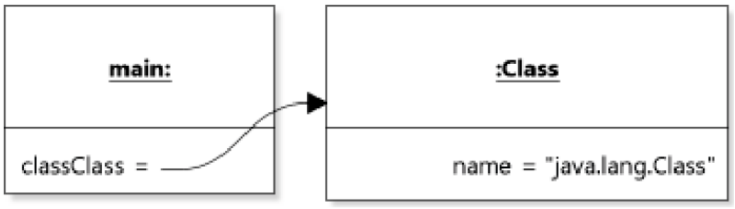
```
Card card = Card.get(Rank.ACE, Suit.CLUBS);
Class<?> cardClass2 = card.getClass();
```

<Property of class `Class`>

Its instances are unique. Indeed, class `Class` has no accessible constructor, and its instances can be considered to be unique flyweight objects.

With metaprogramming, we can introspect any class, including class `Class`. This may at first seem contrived, but it is actually not a special case: class `Class` is just another class.

```
Class<Class> classClass = Class.class;
```



Other usages of class Class

```
// prints the name of all the methods declared in class String. [class Method ]
// Similar classes exist to represent constructors (Constructor) and fields (Field) ]
for( Method method : String.class.getDeclaredMethods() )
{
  System.out.println(method.getName());
}
```

Program Manipulation

In Java it is possible to use metaprogramming features to *change the accessibility of class members, set field values, create new instances of objects, and invoke methods.*

Assuming that our implementation of class `Card` is a realization of the FLY- WEIGHT design pattern and has a private constructor, we will use metaprogramming to "cheat" the pattern and create a duplicate ace of clubs.

```
try
{
  Card card1 = Card.get(Rank.ACE, Suit.CLUBS);
  Constructor<Card> cardConstructor =
    Card.class.getDeclaredConstructor(Rank.class, Suit.class);
  cardConstructor.setAccessible(true);
  Card card2 = cardConstructor.newInstance(Rank.ACE, Suit.CLUBS);
  System.out.println(card1 == card2);
}
catch( ReflectiveOperationException e ) { e.printStackTrace(); }
```

a reference to an instance of class Constructor that represents the (private) constructor of class Card

a new instance of the declaring class of the constructor *represented* by the Constructor instance, as opposed to a new instance of class Constructor

```
try
{
  Card card = Card.get(Rank.TWO, Suit.CLUBS);
  Field rankField = Card.class.getDeclaredField("aRank");
  rankField.setAccessible(true);
  rankField.set(card, Rank.ACE);
  System.out.println(card);
}
catch( ReflectiveOperationException e ) { e.printStackTrace(); }
```

Because rankField represents a *field* of class Card, as opposed to an *instance* of the class, the call to set needs to know on *which instance* the field should be as- signed a value. Thus, the first argument to set is the instance of Card on which we want to assign a new value to the field aRank, and the second argument is the actual value we want to assign.

Program Metadata

With metaprogramming, it is possible for code to operate not only on data that consists of code elements (e.g., classes, methods, fields), but also on metadata about these code elements.

Java provides a system of annotations that can be attached to various code locations. In Java, an annotation type is declared similarly to an interface, for example:

```
public @interface Test {}
```

Then, *annotation instances* can be added to the code, in the form `@Test`. The main advantage of annotations in Java is that they are typed and checked by the compiler. The @Test annotation used to flag unit tests in JUnit is thus a type annotation provided by the JUnit library, and its use is checked by the compiler. For example, writing @test (with a lowercase 't') will result in a compilation error.

5.5 Structuring Tests

- **Fasts**. Unit tests are intended to be run often, and in many cases within a programming-compilation-execution cycle. For this reason, whatever test suite is executed should be able to complete in the order of a few seconds. Otherwise, developers will be tempted to omit running them, and the tests will stop being useful. This means that unit tests should avoid long-running operations such as intensive device I/O and network access, and leave the testing of such function- ality to tests other than unit tests. These could include, for example, acceptance tests or integration tests.
- **Independent**. Each unit test should be able to execute in isolation. This means that, for example, one test should not depend on the fact that another test executes before to leave an input object in a certain state. First, it is often desirable to ex- ecute only a single test. Second, just like code, test suites evolve, with new tests being added and (to a minimum extent) some tests being removed. Test indepen- dence facilitates test suite evolution. Finally, JUnit and similarly designed testing frameworks provide no guarantee that tests will be executed in a predictable order. In practice, this means that each test should start with a fresh initialization of the state used as part of the test.
- **Repeatable**. The execution of unit tests should produce the same result in different environments (for example, when executed on different operating systems). This means that test oracles should not depend on environment-specific proper- ties, such as display size, CPU speed, or system fonts.
- **Focused**. Tests should exercise and verify a slice of code execution behavior that is as narrow as reasonably possible. The rationale for this principle is that the point of unit tests is to help developers identify faults. If a unit test comprises 500 lines of code and tests a whole series of complex interactions between objects, it will not be easy to determine what went wrong if it fails. In contrast, a test that checks a single input on a single method call will

make it easy to home in on a problem. Some have even argued that unit tests should comprise a single assertion. My opinion is that in many cases this is too strict and can lead to inefficiencies. However, tests should ideally focus on testing only *one aspect of one unit under test*. If that unit under test is a method, we can refer to it as the focal method for the test.

- **Readable**. The structure and coding style of the test should make it easy to iden- tify all the components of the test (unit under test, input data, oracle), as well as the rationale for the test. Are we testing the initialization of an object? A special case? A particular combination of values? Choosing an appropriate name for the test can often help in clarifying its rationale.

For example, let us write some unit tests for a method `canMoveTo` of a hypothetical class `FoundationPile` that could be part of the design of the `Solitaire` example application. The method should return true only if it is possible to move the input `pCard` to the top of the pile that an instance of the class represents. According to the rules of the game, this is only possible if the pile is empty and the input card is an ace, or if the input card is of the same suit as the top of the pile, and of a rank immediately above the top of the pile (e.g., you can only put a three of clubs on top of a two of clubs).

```
public class FoundationPile
{
  public boolean isEmpty() { ... }
  public Card peek()       { ... }
  public Card pop()        { ... }
  public void push(Card pCard) { ... }

  public boolean canMoveTo(Card pCard)
  {
    assert pCard != null;
    if( isEmpty() )
    {
      return pCard.getRank() == Rank.ACE;
    }
    else
    {
      return pCard.getSuit() == peek().getSuit() &&
        pCard.getRank().ordinal() ==
          peek().getRank().ordinal()+1;
    }
  }
}
```

As our first test, we will keep things small and only test for the case where the pile is empty:

```
public class TestFoundationPile
{
  @Test
  public void testCanMoveTo_Empty()
  {
    FoundationPile emptyPile = new FoundationPile();
    Card aceOfClubs = Card.get(Rank.ACE, Suit.CLUBS);
    Card threeOfClubs = Card.get(Rank.THREE, Suit.CLUBS);
    assertTrue(emptyPile.canMoveTo(aceOfClubs));
    assertFalse(emptyPile.canMoveTo(threeOfClubs));
  }
}
```

This test respects our five desired properties. It will execute with lightning speed, be independent from any other test that could exist, and is not affected by any environment properties. It is also focused, not only on a single method, but also on a specific input combination for the method. Finally, many properties of this test add to its readability. First, *the name of the test* encodes both the focal method and the input of interest. Second, the *names of the variables* describe their content. Finally, the assertion statements are self-evident. Reading the last line of the test, for exam- ple, we see clearly that calling canMoveTo with a three of clubs on an empty pile will return false, which is correct.

```
@Test
public void testCanMoveTo_NotEmptyAndSameSuit()
{
  Card aceOfClubs = Card.get(Rank.ACE, Suit.CLUBS);
  Card twoOfClubs = Card.get(Rank.TWO, Suit.CLUBS);
  Card threeOfClubs = Card.get(Rank.THREE, Suit.CLUBS);
  FoundationPile pileWithOneCard = new FoundationPile();
  pileWithOneCard.push(aceOfClubs);
  assertTrue(pileWithOneCard.canMoveTo(twoOfClubs));
  assertFalse(pileWithOneCard.canMoveTo(threeOfClubs));
}
```

However, we already note a lot of redundant code between the two tests, namely, the code to create an instance of FoundationPile, and the code to create cards. In test classes that group multiple test methods, it will often be convenient to define a number of "default" objects or values to be used as receiver objects, explicit parameters, and/or oracles. This practice avoids the duplication of setup code in each test method, which constitutes DUPLICATED CODE†.

Baseline objects used for testing are often referred to as a *test fixture*, and declared as fields of a test class. However, for the reasons discussed above, and in particular because JUnit provides no ordering guarantee of any test execution, it is crucial to preserve test independence. This implies that no test method should rely on the fixture being left in a given state by another test. In most cases, this precludes the use of the test class constructor to initialize the fixture, because the constructor is only called once when the test class is instantiated by the framework. The workaround is to nominate a method of the test class to execute before any test method, and

initialize all the required structures afresh. Conveniently, JUnit provides a feature to perform this method call without requiring the test code to include an explicit method call.

By using the @Before annotation, we indicate to JUnit to execute the method before the execution of any test. Similarly, it is also possible to use the @After annotation to mark a method that needs to run after every single test (for instance to free up some resources). Of course, immutable objects do not need to be reinitialized, so they can be stored as static fields of the class. The code below shows the improvement to our test class `TestFoundationPile`, which now uses a test fixture.

```
public class TestFoundationPile
{
  private static final Card ACE_CLUBS =
    Card.get(Rank.ACE, Suit.CLUBS);
  private static final Card TWO_CLUBS =
    Card.get(Rank.TWO, Suit.CLUBS);
  private static final Card THREE_CLUBS =
    Card.get(Rank.THREE, Suit.CLUBS);

  private FoundationPile aPile;

  @Before
  public void setUp() { aPile = new FoundationPile(); }
  @Test
  public void testCanMoveTo_Empty()
  {
    assertTrue(aPile.canMoveTo(ACE_CLUBS));
    assertFalse(aPile.canMoveTo(THREE_CLUBS));
  }

  @Test
  public void testCanMoveTo_NotEmptyAndSameSuit()
  {
    aPile.push(ACE_CLUBS);
    assertTrue(aPile.canMoveTo(TWO_CLUBS));
    assertFalse(aPile.canMoveTo(THREE_CLUBS));
  }
}
```

5.6 Tests and Exceptional Conditions

An important point when writing unit tests is that what we are testing is that the unit under test does what it is expected to. This means that when using design by contract, it does not make sense to test code with input that does not respect the method's preconditions, because the resulting behavior is unspecified.

For example, let us consider a version of method `peek` of class `FoundationPile` which returns the top of the pile.

```
class FoundationPile
{
  boolean isEmpty() { ... }

  /*
   * @return The card on top of the pile.
   * @pre !isEmpty()
   */
  Card peek() { ... }
}
```

The documented precondition implies that the method cannot be expected to fulfill its contract (to return the top card) if the precondition is not met. Thus, if we call the method on an empty pile, there is no expectation to test. The situation is different, however, when raising exceptions is explicitly part of the interface. Let us consider the following slight variant of method `peek()`:

```
class FoundationPile
{
  boolean isEmpty() { ... }

  /*
   * @return The card on top of the pile.
   * @throws EmptyStackException if isEmpty()
   */
  Card peek()
  {
    if( isEmpty() ) { throw new EmptyStackException(); }
    ...
  }}
```

In this case, calling `peek` on an empty pile should result in an `EmptyStackException`. This is part of the specified, expected behavior, and should be tested.

- `expected` property of the @Test annotation (This idiom is replaced with the static method `assertThrows` in JUnit 5)

```
@Test(expected = EmptyStackException.class)
public void testPeek_Empty()
{
   new FoundationPile().peek();
}
```

Limit: Exceptional behavior must be the last thing to happen in the test.

- More flexibility

```
@Test
public void testPeek_Empty()
{
  FoundationPile pile = new FoundationPile();
  try
  {
    pile.peek();
    fail();
  }
  catch(EmptyStackException e ) {}
  assertTrue(pile.isEmpty());
}
```

If the unit under test (the `peek` method here) is faulty in the sense that it does not raise an exception when it should, the code will keep executing normally and reach the following statement, which will force a test failure using JUnit's `fail()` method. If the unit under test is (at least partially) correct in that it does raise the exception when it should, control flow will immediately jump to the `catch` clause, thereby skipping the `fail();` statement. It is then possible to add additional code below the `catch` clause, as illustrated.

5.7 Encapsulation and Unit Testing

A typical solution when an interface does not include a method that would be convenient for testing, is to provide the desired functionality in the form of a *helper method* in the testing class instead.

```
public class TestFoundationPile
{
  private FoundationPile aPile;

  private int size()
  {
    List<Card> temp = new ArrayList<>();
    int size = 0;

    while( !aPile.isEmpty() ) { size++; temp.add(aPile.pop()); }

    while( !temp.isEmpty() )
    { aPile.push(temp.remove(temp.size() - 1)); }

    return size;
  }

  @Before
  public void setUp() { aPile = new FoundationPile(); }
}
```

[Test Private Method or not]

- Private methods are internal elements of other, accessible methods, and therefore are not really "units" that should be tested. Following this logic, the code in private methods should be tested indirectly through the execution of the accessible methods that call them;
- The **private** access modifier is a tool to help us structure the project code, and tests can ignore it.

In cases where it is judged desirable to test a private method, we need to bypass the method's access restriction. This can be done using metaprogramming. For sake of discussion, let us assume that class FoundationPile also has a method:

```
private Optional<Card> getPreviousCard(Card pCard) { ... }
```

```
public class TestFoundationPile
{
  private FoundationPile aPile;

  @Before
  public void setUp() { aPile = new FoundationPile(); }

  private Optional<Card> getPreviousCard(Card pCard)
  {
    try
    {
      Method method = FoundationPile.class.
        getDeclaredMethod("getPreviousCard", Card.class);
      method.setAccessible(true);
      return (Optional<Card>) method.invoke(aPile, pCard);
    }
    catch( ReflectiveOperationException exception )
    {
      fail();
      return Optional.empty();
    }
  }

  @Test
  public void testGetPreviousCard_empty()
  {
    assertFalse(getPreviousCard(Card.get(Rank.ACE, Suit.CLUBS)).
      isPresent());
  }
}
```

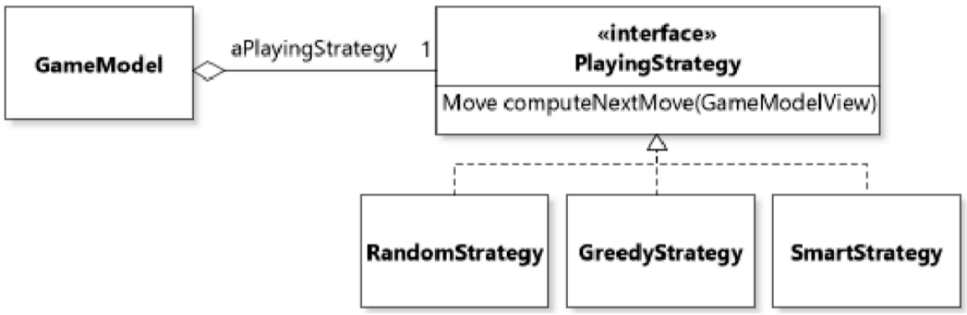
the call to `getPreviousCard` is NOT a call to the unit under tests. Instead, the call is to a *helper method* (of the same name) that uses metaprogramming to call the unit under test while bypassing the access restriction of the **private** keyword.

5.8 Testing with Stubs

The key to unit testing is to test small parts of the code *in isolation*. In some cases, however, factors can make it difficult to test a piece of code in isolation, for example, when the part we want to test:

- triggers the execution of a large chunk of other code;
- includes sections whose behavior depends on the environment (e.g., system fonts);
- involves non-deterministic behavior (e.g., randomness).

e.g. The `GameModel` class has a `tryToAutoPlay()` method that triggers the computation of the next move by dynamically delegating the task to a strategy



- Calling the `tryToAutoPlay` method on an instance of `GameModel` will delegate the call to `computeNextMove` on a strategy object, which will involve the execution of presumably complex behavior to realize the strategy. This does not align well with the concept of unit testing, where we want to test small pieces of code in isolation.
- The implementation of the strategy may involve some randomness.
- How can we know which strategy would be used by the game engine? Presumably we need to determine an oracle for the results.
- How is this different from testing the strategies individually?

The way out of this conundrum is the realization that the responsibility of `GameModel.tryToAutoPlay(...)` is NOT to compute the next move, but rather to delegate this to a strategy. So, to write a unit test that tests that the UUT does what it is expected to do, we only need to verify that it properly relays the request to compute a move to a strategy. This can be achieved with the writing of a stub.
A stub is a greatly simplified version of an object that mimics its behavior sufficiently to support the testing of a UUT that uses this object. Using stubs is heavily dependent on types and polymorphism.

```
public class TestGameModel
{
  static class StubStrategy implements PlayingStrategy
  {
    private boolean aExecuted = false;

    public boolean hasExecuted()
    { return aExecuted; }

    public Move computeNextMove(GameModelView pModelView)
    {
      aExecuted = true;
      return new NullMove();
    }
  }
}
```

This strategy does nothing except remember that its `computeNextMove` method has been called, and returns a dummy `Move` object called `NullMove` (an application of the NULL OBJECT pattern). We can then use an instance of this stub instead of a "real" strategy in the rest of the test. To inject the stub into the game model, we can rely on metaprogramming:

```
@Test
public void testTryToAutoPlay()
{
  Field strategyField =
    GameModel.class.getDeclaredField("aPlayingStrategy");
  strategyField.setAccessible(true);
  StubStrategy strategy = new StubStrategy();
  GameModel model = GameModel.instance();
  strategyField.set(model, strategy);
  ...
}
```

at which point completing the test is simply a matter of calling the UUT `tryToAutoPlay` and verifying that it did properly call the strategy:

```
@Test
public void testTryToAutoPlay()
{
  ...
  model.tryToAutoPlay();
  assertTrue(strategy.hasExecuted());
}
```

The use of stubs in unit testing can get very sophisticated, and frameworks exist to support this task if necessary.

5.9 Test Coverage

Up to now this chapter covered how to define and structure unit tests from a practical standpoint, but avoided the question of what inputs to provide to the unit under test. Despite the example of exhaustive testing in Section 5.1, it should be clear that for the vast majority of UUTs it is not even physically possible to exhaustively test the input space. For example, as discussed in Section 4.2, the number of different arrangements of cards that an instance of class `Deck` can take is astronomical (2.2×10^{68}). Even with a cutting-edge CPU, testing any of the methods of class `Deck` for all possible inputs (i.e., possible states of the implicit parameter) would take an amount of time greater than many times the age of the universe. This is quite incompatible with the requirement that unit tests execute "quickly" (see Section 5.5).

Clearly we need to select some input out of all the possibilities. This is a problem known as *test case selection*, where *test case* can be considered to be a set of input values for an assumed UUT. For example, an instance of `Deck` with a single ace of clubs in the deck is a test case of the method `Deck.draw()`. The basic challenge of the test case selection problem is to test *efficiently*, meaning to find a minimal set of test cases that provides us a maximal amount of "testing" for our code. Unfortunately, while it is fairly obvious what a minimal number of test cases is, there is no natural or even agreed-upon definition of what an "amount of testing" is. However, there is a large body of research results and practical experience on the topic of test case selection: enough for many books (see Further Reading for a recommendation). In this section, I only summarize the key theoretical tenets and practical insights necessary to get started with test case selection. There are two basic ways to approach the selection of test cases:

- **Functional (or black-box) testing** tries to cover as much of the *specified* behavior of a UUT as possible, based on some external specification of what the UUT should do. For the `Deck.draw()` method, this specification is that the method should result in the top card of the deck being removed and returned. There are many advantages to black-box testing, including that it is not necessary to access the code of the UUT, that tests can reveal problems with the specification, and that tests can reveal missing logic.
- **Structural (or white-box) testing** tries to cover as much of the *implemented* behavior of the UUT as possible, based on an analysis of the source code of the UUT. An example is provided below. The main advantage of white-box testing is that it can reveal problems caused by low-level implementation details that are invisible at the level of the specification.

Coverage of functional testing techniques is outside of the scope of this book, so the remainder of this section provides a review of the main concepts of structural testing. Let us consider again the implementation of the `canMoveTo` method of class `FoundationPile` (see Section 5.5, the **`assert`** statement was removed to simplify the discussion).

```
boolean canMoveTo(Card pCard)
{
  if( isEmpty() )
  { return pCard.getRank() == Rank.ACE; }
  else
  {
    return pCard.getSuit() == peek().getSuit() &&
      pCard.getRank().ordinal() ==
        peek().getRank().ordinal()+1;
  }
}
```

Here we can intuitively see that the code structure can be partitioned into different "pieces" that might be good to test. First, there is the case where the pile is empty (the **true** part of the **if** statement), and the case where it is not empty (the **else** block). But then, each of these "pieces" can also be partitioned into different subpieces, for example to cover the case where the cards are in the correct sequence, but of different suits. In the general case, things can get hairy, and it is easy to get lost without a systematic way to understand the code.

One classic method for determining what to test is based on the concept of *coverage*. Informally, a *test coverage metric* is a number (typically a percentage) that determines how much of the code executes when we run our tests. Test coverage metrics can then be computed by code coverage tools that keep track of the code that gets executed when we run unit tests. This sounds simple, but the catch is that there are different definitions of what we can mean by "code", in the context of testing. Each definition is a different way to compute "how much testing" is done. Certain software development organizations may have well-defined *test adequacy criteria* whereby test suites must meet certain coverage thresholds, but in many other cases, the insights provided by coverage metrics are used more generally to help determine where to invest future testing efforts. The following are three well-known coverage metrics (there are many others, see Further Reading).

Statement Coverage

Let us start with the simplest coverage metric: *statement coverage*. Statement coverage is simply the number of statements executed by a test or test suite, divided by the number of statements in the code of interest. The following test:

```
@Test
public void testCanMoveTo_Empty()
{
  assertTrue(aPile.canMoveTo(ACE_CLUBS));
  assertFalse(aPile.canMoveTo(THREE_CLUBS));
}
```

achieves $2/3 = 67\%$ coverage, because the conditional statement predicate and the single statement in the **true** branch are executed, and the single statement in the **false** branch is not. The logic behind statement coverage is simply that if a fault

is present in a statement that is never executed, the tests are not going to help find it. Although this logic may seem appealing, statement coverage is actually a poor coverage metric. A first reason is that it depends heavily on the detailed structure of the code. We could rewrite the `canMoveTo` method as follows, and achieve 100% test coverage with exactly the same tests.

```
boolean canMoveTo(Card pCard)
{
  boolean result = pCard.getSuit() == peek().getSuit() &&
      pCard.getRank().ordinal() ==
       peek().getRank().ordinal()+1;
  if( isEmpty() )
  { result = pCard.getRank() == Rank.ACE; }
  return result;
}
```

The second reason is that not all statements are created equally, and there can be quite a bit that goes on in a statement, especially if this statement involves a compound Boolean expression (as is the case of the first statement in the last example).

Branch Coverage

Branch coverage is the number of program "branches" (decision points) executed by the test(s) divided by the total number of branches in the code of interest. In this context, a branch is one outcome of a condition. Branch coverage is a stronger metric than statement coverage in the sense that for the same coverage result, more of the possible program executions will have been tested. Unfortunately, the concept of branch coverage is a bit ambiguous in the literature, due to the different possible interpretations of the term "branch". In the code of `canMoveTo`, there are only two branches if we only consider the single **`if`** statement. However, both the **`true`** and the **`false`** branches lead to statements that consist of Boolean expressions, which are themselves another type of branch. To be consistent with popular coverage analysis tools, I adopt the definition that Boolean expressions within statements also introduce branches. Although more complex to determine, this definition is also more useful. With this definition, the original code of `canMoveTo` exhibits eight branches: the **`if`**, the first **`return`** statement, the first comparison in the second return statement, and the second comparison in the second return statement. Each of these branches has a **`true`** and **`false`** outcome. The only test written so far thus has only $3/8 = 37.5\%$ branch coverage. If we add the other test shown in Section 5.5 to the test suite:

```
@Test
public void testCanMoveTo_NotEmptyAndSameSuit()
{
  aPile.push(ACE_CLUBS);
  assertTrue(aPile.canMoveTo(TWO_CLUBS));
  assertFalse(aPile.canMoveTo(THREE_CLUBS));
}
```

we get $7/8 = 87.5\%$. This is pretty good, but our systematic coverage analysis points out that we are actually missing one branch, which corresponds to the case where the pile is not empty, and the card at the top of the pile and the card passed as argument are of different suits. Generally speaking, branch coverage is one of the most useful test coverage criteria. It is well supported by testing tools and relatively easy to interpret, and also *subsumes* statement coverage, meaning that achieving complete branch coverage always implies complete statement coverage.

Path Coverage

There are other coverage metrics stronger than branch coverage. For example, one could, in principle, compute a *path coverage* metric as the number of execution paths actually executed over all possible execution paths in the code of interest. Path coverage subsumes almost all other coverage metrics, and is a very close approximation of the "entire behavior that is possible to test". Unfortunately, in almost all cases, the number of paths through a piece of code will be unbounded, so it will not be possible to compute this metric. For this reason, path coverage is considered a theoretical metric, useful for reasoning about test coverage in the abstract, but without any serious hope of general practical applicability. Interestingly, the number of paths in the code of `canMoveTo` is actually only five, so, less than the number of branches! The path coverage of the two-test test suite above can thus be computed, at $4/5 = 80\%$. Because the structure of the code is simple, and without loops, this is not overly surprising. However, as soon as loops enter the picture, reasoning about paths becomes problematic.

Insights

This chapter described techniques to structure and implement unit tests for a project, and argued that unit tests can provide valuable feedback on the design of production code. The following insights assume you have decided to adopt unit testing and are using a unit testing framework.

- Every unit test includes the execution of unit under test (UUT), some input data passed to the UUT, an oracle that describes what the result of executing the UUT should be, and one or more assertions that compare the result of the execution with the oracle;
- Design your unit tests so that they are focused, that is, that they isolate and test a small and well-defined amount of behavior;
- Design your unit tests to run fast, be independent from each other, and be repeatable in any computing environment;
- Design your unit tests to be readable: consider using the name of the test and local variables to add clarity about what you are testing and to describe the oracle;