Flexible, Reliable Software

Additional Material

Another Example of
Test-Driven Development:
The Hotel Safe

Henrik Bærbak Christensen

Status: Released / Revision 2103

September 4, 2018

Chapter 1

The Challenge

A hotel provides a small safe in each room of the hotel. We are asked to implement the production code that handles entering the code, locking and unlocking the safe, as well as writing relevant information in a small six character display.

1.1 The Physical Safe

The safe has a display consisting of six 7-segment elements[1], see Figure 1.1. It has a numerical panel having buttons for the digits "0" to "9". Additionally it has three special buttons with a padlock symbol (marked "Lock"), a key symbol (marked "Open"), and a broken arrow symbol (marked "Set New PIN"). From the factory the safe is preprogrammed with the pin code "123456" as code to open the safe.

Figure 1.1: The display and key panel of the safe.

Users can program their own pin codes to open the safe, however, it must always be a 6 digit pin code.

[1] 7-segment elements can display all digits as well as subset of letters, those that do not contain any oblique lines.

1.2 Stories

The safe is provided with a very short guide for operating the safe intended for the hotel room visitor, as seen in Figure 1.2.

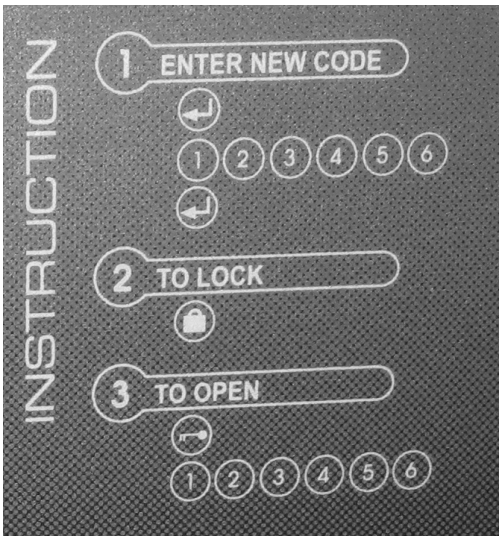


Figure 1.2: The safe's instructions.

The company defines a set of stories to drive the implementation effort. The following stories assume the safe has the factory default pin code "123456" as the 6-digit proper pin code.

Story 1: Unlock Safe The user approaches the safe whose door is locked. The display is empty, which means it contains 6 spaces/blanks. The user hits the key-symbol button. The user enters his previously stored pin code by pressing the buttons one at the time: "1", "2", "3", "4", "5", "6". The display reacts by writing each digit as it is pressed. After the final "6" button press, the display clears and displays "OPEN ". The safe door unlocks and can be opened.

Story 2: Lock Safe The safe door is unlocked. The display reads "OPEN ". The user closes the door and presses the lock button. The door locks. The display reads "CLOSED".

Story 3: Forgetting key Button The safe is locked. The user forgets to hit the key button first and hits "1". The display reads "ERROR ". All following button hits result in the display reading "ERROR ", unless the key botton is pressed.

Story 4: Wrong Code The safe is locked. The user hits key followed by 1 2 4 3 5 6. The display is cleared. The safe remains locked.

Story 5: Set New Code The safe is open/unlocked. The user hits the pin button, enters a new six digit pin code, "777333", and finally hits the pin again. The safe's display reads "CODE ". It remains unlocked. After locking, the safe can only be unlocked (see story 1) by entering the new pin code "777333".

Note: The scenarios above are not quite consistent: After locking the safe, the display reads "CLOSED" (and entering new pin "CODE ") but then how does it get to the

state where the display is cleared? In the real safe there will be a timer clearing the display after a short period but I will ignore this feature in the following discussion.

Note also that story 2 is actually not complete as it only discusses the behavior of the lock button if the safe is unlocked, not what happens if it is pressed while the safe is already locked. This, however, will be discovered in iteration 8.

1.3 Given Interfaces

The Safe interface is given and is simplified for the purpose of our process, a real safe would interact with various hardware components which would require the introduction of test stubs which is a topic not yet introduced.

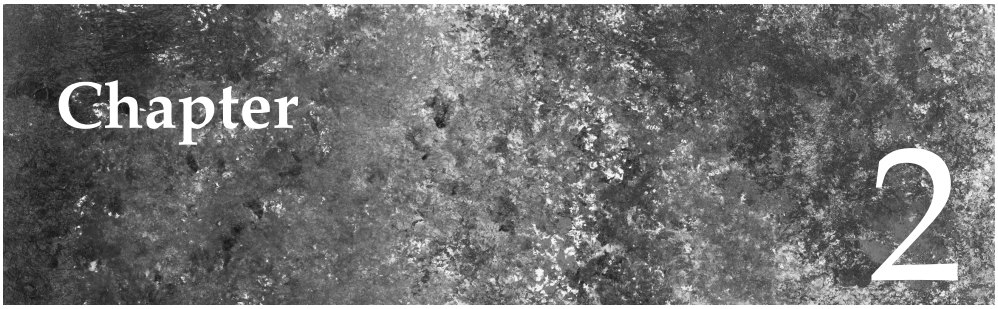
Listing: examples/safe/iteration-0/Safe.java

```
/** The specification of a simple safe.
*/
public interface Safe {
  /** Enter a button press on the safe. */
  public void enter(Button button);

  /** Read the output of the display on the safe.
   * POSTCONDITON: It is always a non-null string of
   * exactly 6 characters that can be printed on
   * a 7-segment display.
   * @return: the output on the display
   */
  public String readDisplay();

  /** Get the state of the safe: is it locked or not.
   * @return true iff the safe is locked.
   */
  public boolean isLocked();
}
```

The Button class is just a simple enumeration.

Listing: examples/safe/iteration-0/Button.java

```
/** The buttons on the safe.
*/
public enum Button  {
  D0, D1, D2, D3, D4, D5,
    D6, D7, D8, D9, LOCK, KEY, PIN
}
```



TDD of Safe

In this chapter I will develop an implementation of the Safe interface in a class SafeImpl. However, I will only develop the code for the first four stories (no entering of a new pin code) in this *small release* or sprint.

The first shot at a test list looks like this.

- Initial: Display reads 6 spaces. Safe is locked.
- Enter (key,1,2,3) gives "123 " as output. Safe is locked.
- Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.
- Enter (1,2) gives "ERROR ". Safe locked.
- Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.
- Unlocked safe: Enter (lock) gives empty display. Safe locked.

2.1 Iteration 1: Setup

As most of the items on the test list assume the safe is initially in the initial/locked state it makes sense to make it our *One Step Test—Initial: Display reads 6 spaces. Safe is locked*.

Step 1: Quickly add a test:

Listing: examples/safe/iteration-1/TestSafe.java

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

/** Test cases for Safe.
*/
public class TestSafe {

  @Test
  public void shouldInitiallyBeLockedAndCleanDisplay() {
    Safe safe = new SafeImpl();
    assertEquals("Display must be empty",
                 "      ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }
}
```

leads to *Step 2: Run all tests and see the new one fail*:

```
JUnit version 4.4
.E
Time: 0,015
There was 1 failure:
1) shouldInitiallyBeLockedAndCleanDisplay(TestSafe)
java.lang.AssertionError:
  Display must be empty expected:<      > but was:<null>
```

Step 3: Make a little change is of course heavy use of *Fake It*.

Listing: examples/safe/iteration-1/SafeImpl.java

```
/** Implementation of the Safe.
*/
public class SafeImpl implements Safe {
  public void enter(Button button) {
  }

  public boolean isLocked() {
    return true;
  }

  public String readDisplay() {
    return "      ";
  }

}
```

Which gives me *Step 4: Run all tests and see them all succeed*. No need for *Step 5: Refactor to remove duplication*.

2.2 Iteration 2: Half Code

Step 1: Quickly add a test—Enter (key,1,2,3) gives "123 " as output. Safe is locked.

Listing: examples/safe/iteration-2/TestSafe.java

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

/** Test cases for Safe.
*/
public class TestSafe {
  private Safe safe;

  @Before
  public void setup() {
    safe = new SafeImpl();
  }

  @Test
  public void shouldInitiallyBeLockedAndCleanDisplay() {
    assertEquals("Display must be empty",
```

```
                "      ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }

  @Test
  public void shouldDisplayCodeAsEntered() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    assertEquals("Display must be 123",
                 "123   ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }
}
```

(Note! I do not just enter "123" but the real and proper sequence as outlined in the story. At the moment the production code could be implemented without pressing the key button, but once I enter the production code to ensure it is pressed this test case would fail and I would have to recode it. See sidebar 5.3 on page 63 in FRS.)

Leads to *Step 2: Run all tests and see the new one fail*. But how to proceed? Perhaps this test case is a bit too complex? Do I have to introduce code to translate from Button.D1 to "1" etc. at this point in my process? No, I can actually *Step 3: Make a little change Fake It* and still *Triangulate* something sensible into the implementation, namely the introduction of an instance variable to keep the display contents.

Listing: examples/safe/iteration-2/SafeImpl.java

```
/** Implementation of the Safe.
*/
public class SafeImpl implements Safe {
  private String displayContents;

  public SafeImpl() {
    displayContents = "      ";
  }

  public void enter(Button button) {
    displayContents = "123   ";
  }

  public boolean isLocked() {
    return true;
  }

  public String readDisplay() {
    return displayContents;
  }

}
```

Remember, keep focus and take (very) small steps. *Step 4: Run all tests and see them all succeed*.

Step 5: Refactor to remove duplication—I can refactor the test case class to use the before method.

Listing: examples/safe/iteration-2/TestSafe.java

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

/** Test cases for Safe.
*/
public class TestSafe {
  private Safe safe;

  @Before
  public void setup() {
    safe = new SafeImpl();
  }

  @Test
  public void shouldInitiallyBeLockedAndCleanDisplay() {
    assertEquals("Display must be empty",
                 "      ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }

  @Test
  public void shouldDisplayCodeAsEntered() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    assertEquals("Display must be 123",
                 "123   ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }
}
```

- ~~Initial: Display reads 6 spaces. Safe is locked.~~
- ~~Enter (key,1,2,3) gives "123 " as output. Safe is locked.~~
- Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.
- Enter (1,2) gives "ERROR ". Safe locked.
- Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.
- Unlocked safe: Enter (lock) gives empty display. Safe locked.

2.3 Iteration 3: Unlock Safe

I look at the next item on my list *Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.* Seems like a nice *One Step Test* and easy to code as a test case.

Listing: examples/safe/iteration-3/TestSafe.java

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

/** Test cases for Safe.
*/
public class TestSafe {
  private Safe safe;

  @Before
```

```
  public void setup() {
    safe = new SafeImpl();
  }

  @Test
  public void shouldInitiallyBeLockedAndCleanDisplay() {
    assertEquals("Display must be empty",
                 "      ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }

  @Test
  public void shouldDisplay123CodeAsEntered() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    assertEquals("Display must be 123",
                 "123   ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }

  @Test
  public void shouldUnlockSafeOnCorrectCode() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    safe.enter(Button.D4);
    safe.enter(Button.D5);
    safe.enter(Button.D6);
    assertEquals("Display must be 123456",
                 "123456", safe.readDisplay() );
    assertTrue("Safe must be unlocked", !safe.isLocked());
  }
}
```

Of course I get *Step 2: Run all tests and see the new one fail*. Now the question is if I can get *Step 3: Make a little change*? It seems I need some design and quite some code to *Triangulate* the fake-it code away? The test case both require unlocking (which is faked) as well as comparing entered code with stored correct code (which is not even faked, it is non-existing)—and even accumulating digits to be shown in the display. The bottom line is that the test case is easy but the production code will not be *taking small steps.* A principle not mentioned in the FRS book, but given in the overview on the inner cover is:

TDD Principle: **Child Test**

How do you get a test case running that turns out to be too big? Write a smaller test case that represents the broken part of the bigger test case. Get the smaller test case running. Reintroduce the larger test case.

So, what should I take first? It seems accumulating the entered code seems natural: having a string and adding each pressed digit to this seems rather simple. But—how do I translate from the Button enumeration values to characters? A large switch in SafeImpl? A public method in Button? A map lookup in SafeImpl? I decide to keep Button simple and generally prefer using to look up in data structures to large

switches. To force this design into action I comment my iteration 3 test case out and turn my attention to the child tests.

2.4 Iteration 4: Child Test/Accumulating Digits

OK, press "1" and see the display showing "1", press "7" and see the display showing "17", etc.

Listing: examples/safe/iteration-4/TestSafe.java

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

/** Test cases for Safe.
*/
public class TestSafe {
  private Safe safe;

  @Before
  public void setup() {
    safe = new SafeImpl();
  }

  @Test
  public void shouldInitiallyBeLockedAndCleanDisplay() {
    assertEquals("Display must be empty",
                 "      ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }

  @Test
  public void shouldDisplay123CodeAsEntered() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    assertEquals("Display must be 123",
                 "123   ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }
  /*
  @Test
  public void shouldUnlockSafeOnCorrectCode() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    safe.enter(Button.D4);
    safe.enter(Button.D5);
    safe.enter(Button.D6);
    assertEquals("Display must be 123456",
                 "123456", safe.readDisplay() );
    assertTrue("Safe must be unlocked", !safe.isLocked());
  }
  */
```

```
  @Test
  public void shouldAccumulateDigitsInDisplay() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    assertEquals("Display must be 1",
                 "1     ", safe.readDisplay() );
    safe.enter(Button.D7);
    assertEquals("Display must be 17",
                 "17    ", safe.readDisplay() );
    safe.enter(Button.D9);
    assertEquals("Display must be 179",
                 "179   ", safe.readDisplay() );
  }

}
```

Step 3: Make a little change: introduce storage for entered digits, and prepare the lookup. A bit of coding quickly shows that handling trailing spaces using a String type is ackward, so I turn my attention to keeping a array of six chars. The requirement that pushing the key, lock, and pin buttons also require some extra logic. I arrive at:

Listing: examples/safe/iteration-4/SafeImpl.java

```
import java.util.*;

/** Implementation of the Safe.
*/
public class SafeImpl implements Safe {
  private char[] displayContents;
  private int index;
  private Map<Button,Character> map;

  public SafeImpl() {
    displayContents = new char[6];
    for (int i = 0; i < 6; i++) displayContents[i] = ' ';
    index = 0;
    map = new HashMap<Button,Character>();
    map.put(Button.D1, '1');
    map.put(Button.D2, '2');
    map.put(Button.D3, '3');
    map.put(Button.D7, '7');
    map.put(Button.D9, '9');
  }

  public void enter(Button button) {
    Character c = map.get(button);
    if ( c != null ) {
      displayContents[index] = c.charValue();
      index = (index+1) % 6;
    }
  }

  public boolean isLocked() {
    return true;
  }

  public String readDisplay() {
    return String.valueOf(displayContents);
```

```
  }

}
```

which *Step 4: Run all tests and see them all succeed*. I need a test case to triangulate the rest of the key symbols into the production code.

- ~~Initial: Display reads 6 spaces. Safe is locked.~~
- ~~Enter (key,1,2,3) gives "123 " as output. Safe is locked.~~
- Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.
- Enter (1,2) gives "ERROR ". Safe locked.
- Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.
- Unlocked safe: Enter (lock) gives empty display. Safe locked.
- ~~Accumulate digits in the display when pressing 179.~~
- *Enter codes with digits 0, 4, 5, 6, and 8.*

2.5 Iteration 5: Unlock Safe—Take Two

Now I can return to the main test—unlocking the safe. I simply uncomment the test case from iteration 3.

Listing: examples/safe/iteration-5/TestSafe.java

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

/** Test cases for Safe.
*/
public class TestSafe {
  private Safe safe;

  @Before
  public void setup() {
    safe = new SafeImpl();
  }

  @Test
  public void shouldInitiallyBeLockedAndCleanDisplay() {
    assertEquals("Display must be empty",
                 "      ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }

  @Test
  public void shouldDisplay123CodeAsEntered() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    assertEquals("Display must be 123",
                 "123   ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }

  @Test
  public void shouldUnlockSafeOnCorrectCode() {
```

```
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    safe.enter(Button.D4);
    safe.enter(Button.D5);
    assertEquals("Display must be 12345",
                 "12345 ", safe.readDisplay() );
    safe.enter(Button.D6);
    assertEquals("Display must be OPEN",
                 "OPEN  ", safe.readDisplay() );

    assertTrue("Safe must be unlocked", !safe.isLocked());
  }

  @Test
  public void shouldAccumulateDigitsInDisplay() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    assertEquals("Display must be 1",
                 "1     ", safe.readDisplay() );
    safe.enter(Button.D7);
    assertEquals("Display must be 17",
                 "17    ", safe.readDisplay() );
    safe.enter(Button.D9);
    assertEquals("Display must be 179",
                 "179   ", safe.readDisplay() );
  }

}
```

Step 2: Run all tests and see the new one fail:

```
1) shouldUnlockSafeOnCorrectCode(TestSafe)
org.junit.ComparisonFailure: Display must be 123456
  expected:<123[456]> but was:<123[   ]>
```

which force me to triangulate more of the mapping between buttons and characters into place.

```
  public SafeImpl() {
    displayContents = new char[6];
    for (int i = 0; i < 6; i++) displayContents[i] = ' ';
    index = 0;
    map = new HashMap<Button,Character>();
    map.put(Button.D1, '1');
    map.put(Button.D2, '2');
    map.put(Button.D3, '3');
    map.put(Button.D4, '4');
    map.put(Button.D5, '5');
    map.put(Button.D6, '6');
    map.put(Button.D7, '7');
    map.put(Button.D9, '9');
  }
```

Next only the last assert complains:

```
1) shouldUnlockSafeOnCorrectCode(TestSafe)
java.lang.AssertionError: Safe must be unlocked
```

Step 3: Make a little change involves introducing a boolean and the code matching algorithm:

Listing: examples/safe/iteration-5/SafeImpl.java

```
import java.util.*;

/** Implementation of the Safe.
*/
public class SafeImpl implements Safe {
  private char[] displayContents;
  private int index;
  private boolean locked;
  private Map<Button,Character> map;

  public SafeImpl() {
    displayContents = new char[6];
    for (int i = 0; i < 6; i++) displayContents[i] = ' ';

    locked = true;

    index = 0;
    map = new HashMap<Button,Character>();
    map.put(Button.D1, '1');
    map.put(Button.D2, '2');
    map.put(Button.D3, '3');
    map.put(Button.D4, '4');
    map.put(Button.D5, '5');
    map.put(Button.D6, '6');
    map.put(Button.D7, '7');
    map.put(Button.D9, '9');
  }

  public void enter(Button button) {
    Character c = map.get(button);
    if ( c != null ) {
      displayContents[index] = c.charValue();
      index = (index+1) % 6;
    }
    if ( readDisplay().equals("123456") ) {
      locked = false;
      displayContents[0]='O';
      displayContents[1]='P';
      displayContents[2]='E';
      displayContents[3]='N';
      displayContents[4]=displayContents[5]=' ';
    }
  }

  public boolean isLocked() {
    return locked;
  }

  public String readDisplay() {
    return String.valueOf(displayContents);
  }
```

```
}
```

Ups—no test for the display reading "OPEN"! Actually the test case is wrong! I have not been careful enough when I converted the items on my test list into test cases. This is regretable and probably because I have done this work without doing pair programming. Never the less this blunder carries a lesson—we do make mistakes, and when we do so—well—we have to correct them as soon as possible.

So—I quickly adjust the test case to the proper specification.

```
@Test
public void shouldUnlockSafeOnCorrectCode() {
  safe.enter(Button.KEY);
  safe.enter(Button.D1);
  safe.enter(Button.D2);
  safe.enter(Button.D3);
  safe.enter(Button.D4);
  safe.enter(Button.D5);
  assertEquals("Display must be 12345",
               "12345 ", safe.readDisplay() );
  safe.enter(Button.D6);
  assertEquals("Display must be OPEN",
               "OPEN  ", safe.readDisplay() );

  assertTrue("Safe must be unlocked", !safe.isLocked());
}
```

Which of course leads to *Step 2: Run all tests and see the new one fail*. The *Step 3: Make a little change*:

```
public void enter(Button button) {
  Character c = map.get(button);
  if ( c != null ) {
    displayContents[index] = c.charValue();
    index = (index+1) % 6;
  }
  if ( readDisplay().equals("123456") ) {
    locked = false;
    displayContents[0]='O';
    displayContents[1]='P';
    displayContents[2]='E';
    displayContents[3]='N';
    displayContents[4]=displayContents[5]=' ';
  }
}
```

Step 4: Run all tests and see them all succeed. I note that I do not really like the tedious way to set a completely new contents of the display. However, this iteration gives me no excuse to change this code—but looking over the test list I see that some of the later iterations will make duplicated code (setting the display to CLOSED and ERROR) and thus refactor it later.

- ~~Initial: Display reads 6 spaces. Safe is locked.~~
- ~~Enter (key,1,2,3) gives "123 " as output. Safe is locked.~~
- ~~Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.~~
- Enter (1,2) gives "ERROR ". Safe locked.
- Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.
- Unlocked safe: Enter (lock) gives empty display. Safe locked.
- ~~Accumulate digits in the display when pressing 179.~~
- Enter codes with digits 0, ~~4, 5, 6,~~ and 8.

2.6 Iteration 6: Wrong Code

It is time to enter a wrong code: *Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.*

Step 1: Quickly add a test.

Listing: examples/safe/iteration-6/TestSafe.java

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

/** Test cases for Safe.
*/
public class TestSafe {
  private Safe safe;

  @Before
  public void setup() {
    safe = new SafeImpl();
  }

  @Test
  public void shouldInitiallyBeLockedAndCleanDisplay() {
    assertEquals("Display must be empty",
                 "      ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }

  @Test
  public void shouldDisplay123CodeAsEntered() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    assertEquals("Display must be 123",
                 "123   ", safe.readDisplay() );
    assertTrue("Safe must be locked", safe.isLocked());
  }

  @Test
  public void shouldUnlockSafeOnCorrectCode() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    safe.enter(Button.D4);
    safe.enter(Button.D5);
    assertEquals("Display must be 12345",
```

```
                "12345 ", safe.readDisplay() );
    safe.enter(Button.D6);
    assertEquals("Display must be OPEN",
                "OPEN  ", safe.readDisplay() );

    assertTrue("Safe must be unlocked", !safe.isLocked());
  }

  @Test
  public void shouldAccumulateDigitsInDisplay() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    assertEquals("Display must be 1",
                "1     ", safe.readDisplay() );
    safe.enter(Button.D7);
    assertEquals("Display must be 17",
                "17    ", safe.readDisplay() );
    safe.enter(Button.D9);
    assertEquals("Display must be 179",
                "179   ", safe.readDisplay() );
  }

  @Test
  public void shouldKeepLockedForWrongCode() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D4);
    safe.enter(Button.D3);
    safe.enter(Button.D5);
    safe.enter(Button.D6);
    assertEquals("Display must be CLOSED",
                "CLOSED", safe.readDisplay() );

    assertTrue("Safe must be stay locked", safe.isLocked());
  }
}
```

My first attempt at *Step 3: Make a little change* makes 4 out of 5 testcases fail!

```
  public void enter(Button button) {
    Character c = map.get(button);
    if ( c != null ) {
      displayContents[index] = c.charValue();
      index = (index+1) % 6;
    }
    if ( readDisplay().equals("123456") ) {
      locked = false;
      displayContents[0]='O';
      displayContents[1]='P';
      displayContents[2]='E';
      displayContents[3]='N';
      displayContents[4]=displayContents[5]=' ';
    } else if ( index == 0 ) { // 6 digits entered
      displayContents[0]='C';
      displayContents[1]='L';
      displayContents[2]='O';
      displayContents[3]='S';
      displayContents[4]='E';
      displayContents[5]='D';
```

```
    }
  }
```

The typical complaint from JUnit is

```
1) shouldDisplay123CodeAsEntered(TestSafe)
org.junit.ComparisonFailure: Display must be 123
  expected:<123[   ]> but was:<123[SED]>
        at org.junit.Assert.assertEquals(Assert.java:99)
```

Ahh—the key button does not advance index and thus "CLOSED" is set into the displayContents. Making the first `if` emcompass the whole block solves the problem and I get *Step 4: Run all tests and see them all succeed*.

Step 5: Refactor to remove duplication: it is obvious to make a private method to set the contents of the character array, setDisplayContents, which can be used three places in the production code. The code is reduced in size and duplication is avoided.

Listing: examples/safe/iteration-6/SafeImpl.java

```
import java.util.*;

/** Implementation of the Safe.
*/
public class SafeImpl implements Safe {
  private char[] displayContents;
  private int index;
  private boolean locked;
  private Map<Button,Character> map;

  public SafeImpl() {
    displayContents = new char[6];
    setDisplayContents("      ");

    locked = true;

    index = 0;
    map = new HashMap<Button,Character>();
    map.put(Button.D1, '1');
    map.put(Button.D2, '2');
    map.put(Button.D3, '3');
    map.put(Button.D4, '4');
    map.put(Button.D5, '5');
    map.put(Button.D6, '6');
    map.put(Button.D7, '7');
    map.put(Button.D9, '9');
  }

  public void enter(Button button) {
    Character c = map.get(button);
    if ( c != null ) {
      displayContents[index] = c.charValue();
      index = (index+1) % 6;

      if ( readDisplay().equals("123456") ) {
        locked = false;
        setDisplayContents("OPEN  ");
        displayContents[4]=displayContents[5]=' ';
```

```
      } else if ( index == 0 ) { // 6 digits entered
        setDisplayContents("CLOSED");
      }
    }
  }

  public boolean isLocked() {
    return locked;
  }

  public String readDisplay() {
    return String.valueOf(displayContents);
  }

  /** PRECONDITION: string must be exactly 6 charaters long */
  private void setDisplayContents(String sixCharString) {
    for (int i = 0; i < 6; i++)
      displayContents[i] = sixCharString.charAt(i);
  }
}
```

- ~~Initial: Display reads 6 spaces. Safe is locked.~~
- ~~Enter (key,1,2,3) gives "123 " as output. Safe is locked.~~
- ~~Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.~~
- Enter (1,2) gives "ERROR ". Safe locked.
- ~~Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.~~
- Unlocked safe: Enter (lock) gives empty display. Safe locked.
- ~~Accumulate digits in the display when pressing 179.~~
- Enter codes with digits 0, ~~4, 5, 6,~~ and 8.

2.7 Iteration 7: Getting 0 and 8 to work.

I use *Representative Data*: Each button requires its own processing (as each button is mapped into a specific character) and the item *Enter codes with digits 0, 4, 5, 6, and 8* highlights that I still need to test with button 0 and 8. *Step 1: Quickly add a test*:

```
  @Test
  public void shouldDisplay908CodeAsEntered() {
    safe.enter(Button.KEY);
    safe.enter(Button.D9);
    safe.enter(Button.D0);
    safe.enter(Button.D8);
    assertEquals("Display must be 908",
                 "908   ", safe.readDisplay() );
  }
```

Step 3: Make a little change is trivial and omitted here—look into the code.

- ~~Initial: Display reads 6 spaces. Safe is locked.~~
- ~~Enter (key,1,2,3) gives "123 " as output. Safe is locked.~~
- ~~Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.~~
- Enter (1,2) gives "ERROR ". Safe locked.
- ~~Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.~~
- Unlocked safe: Enter (lock) gives empty display. Safe locked.
- ~~Accumulate digits in the display when pressing 179.~~
- ~~Enter codes with digits 0, 4, 5, 6, and 8.~~

2.8 Iteration 8: Forgetting the KEY Button

Almost finished. Let us try to get the "ERROR" situation. *Step 1: Quickly add a test*:

```
@Test
  public void shouldDisplayERRORWhenForGettingKeyButton() {
  safe.enter(Button.D1);
  assertEquals("Display must be ERROR",
               "ERROR ", safe.readDisplay() );
  safe.enter(Button.D2);
  assertEquals("Display must be ERROR",
               "ERROR ", safe.readDisplay() );
  assertTrue("Safe must be stay locked", safe.isLocked());
}
```

which of course *Step 2: Run all tests and see the new one fail*. To *Step 3: Make a little change* I introduce a boolean enterCodeState that must be true if the user is entering a new code. It is of course set to false in the constructor. I massage the production code somewhat, using classic switching for the state machine to end up in *Step 4: Run all tests and see them all succeed* with the following code:

```
public void enter(Button button) {
  Character c = map.get(button);
  // Statemachine
  if ( enterCodeState ) { // Safe is expecting a code
    if ( c != null ) {
      displayContents[index] = c.charValue();
      index = (index+1) % 6;

      if ( readDisplay().equals("123456") ) {
        locked = false;
        setDisplayContents("OPEN   ");
      } else if ( index == 0 ) { // 6 digits entered
        setDisplayContents("CLOSED");
      }
    }
  } else { // Safe is not expecting a code
    if ( c != null ) {
      setDisplayContents("ERROR ");
    }
    if ( button == Button.KEY ) {
      enterCodeState = true;
      setDisplayContents("      ");
    }
  }
}
```

One thing nags me: when is the enterCodeState set to false? Never! And the stories does not really tell me when the safe should not be in the *enter a code* state. So—I have to talk to the users or customers. They tell me that if you press the lock key then the safe should revert to the initial/waiting state. Also the safe should enter this state after unlocking it or if the entered pin code was incorrect. This can be formulated as a stories.

Story 1a: Regret Opening Safe The user starts entering the proper pin code, "1", "2", "3", but regrets to open the safe. The user hits the lock button, the display clears, and the safe remains locked.

Story 4a: Completed Pin Code The user enters a 6-digit pin code (valid or invalid). The safe returns to the initial/waiting state.

I formulate this as items on the test list *Safe goes to initial state if lock button entered* and *Safe returns to initial state after 6 digits entered.*

While I achieve *Step 4: Run all tests and see them all succeed*, the analyzability of this code is not good: nested `if`s are notoriously difficult to analyze. So *Step 5: Refactor to remove duplication*, chopping up the structure using private methods. Another approach would be to refactor the design to use the STATE pattern.

Listing: examples/safe/iteration-8/SafeImpl.java

```
import java.util.*;

/** Implementation of the Safe.
*/
public class SafeImpl implements Safe {
  private char[] displayContents;
  private int index;
  private boolean locked, enterCodeState;
  private Map<Button,Character> map;

  public SafeImpl() {
    displayContents = new char[6];
    setDisplayContents("      ");

    locked = true; enterCodeState = false;

    index = 0;
    map = new HashMap<Button,Character>();
    map.put(Button.D1, '1'); map.put(Button.D2, '2');
    map.put(Button.D3, '3'); map.put(Button.D4, '4');
    map.put(Button.D5, '5'); map.put(Button.D6, '6');
    map.put(Button.D7, '7'); map.put(Button.D8, '8');
    map.put(Button.D9, '9'); map.put(Button.D0, '0');
  }

  public void enter(Button button) {
    // Statemachine
    if ( enterCodeState ) { // Safe is expecting a code
      handleButtonInEnterCodeState(button);
    } else { // Safe is not expecting a code
      handleButtonInInitialState(button);
    }
  }

  public boolean isLocked() {
    return locked;
  }

  public String readDisplay() {
    return String.valueOf(displayContents);
  }

  /** PRECONDITION: string must be exactly 6 charaters long */
  private void setDisplayContents(String sixCharString) {
    for (int i = 0; i < 6; i++)
      displayContents[i] = sixCharString.charAt(i);
  }
```

```
  private void handleButtonInEnterCodeState(Button button) {
    Character c = map.get(button);
    if ( c != null ) {
      displayContents[index] = c.charValue();
      index = (index+1) % 6;

      if ( readDisplay().equals("123456") ) {
        locked = false;
        setDisplayContents("OPEN  ");
      } else if ( index == 0 ) { // 6 digits entered
        setDisplayContents("CLOSED");
      }
    }
  }

  private void handleButtonInInitialState(Button button) {
    Character c = map.get(button);
    if ( c != null ) {
      setDisplayContents("ERROR ");
    }
    if ( button == Button.KEY ) {
      enterCodeState = true;
      setDisplayContents("      ");
    }
  }
}
```

Reviewing the final code I wonder if it is correct that the key button is processed when in initial state. Story 3 states that whenever I hit the key button the safe should begin to accept a new code, but this aspect has not really been captured by any item on the test list, and the production code may not handle it as far as I can see. Thus one more for the test list.

- ~~Initial: Display reads 6 spaces. Safe is locked.~~
- ~~Enter (key,1,2,3) gives "123 " as output. Safe is locked.~~
- ~~Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.~~
- ~~Enter (1,2) gives "ERROR ". Safe locked.~~
- ~~Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.~~
- Unlocked safe: Enter (lock) gives empty display. Safe locked.
- ~~Accumulate digits in the display when pressing 179.~~
- ~~Enter codes with digits 0, 4, 5, 6, and 8.~~
- *Safe goes to initial state if lock button entered.*
- *Safe returns to initial state after 6 digits entered.*
- *Enter (1,2,key,1,2,3,4,5,6) gives unlocked safe.*

2.9 Iteration 9: Entering Key In The Middle Of Things

The last observation nags me so this is the next one step test. *Step 1: Quickly add a test*:

```
  @Test
    public void shouldOpenSafeAfterError() {
    safe.enter(Button.D1);
```

```
    safe.enter(Button.D2);
    assertEquals("Display must be ERROR",
                 "ERROR ", safe.readDisplay() );
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    safe.enter(Button.D4);
    safe.enter(Button.D5);
    safe.enter(Button.D6);
    assertEquals("Display must be OPEN",
                 "OPEN  ", safe.readDisplay() );

    assertTrue("Safe must be unlocked", !safe.isLocked());
  }
```

which pass. No need to have worried about the production code but the test is important as we now have the requirement by the story covered by our test suite and are thus guarded against failures if we later have to modify the code.

The test code contains duplicated code now. *Step 5: Refactor to remove duplication*.

```
  @Test
    public void shouldOpenSafeAfterError() {
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    assertEquals("Display must be ERROR",
                 "ERROR ", safe.readDisplay() );
    enterCorrectCode();
    assertEquals("Display must be OPEN",
                 "OPEN  ", safe.readDisplay() );
    assertTrue("Safe must be unlocked", !safe.isLocked());
  }
```

and

```
  private void enterCorrectCode() {
    safe.enter(Button.KEY);
    safe.enter(Button.D1);
    safe.enter(Button.D2);
    safe.enter(Button.D3);
    safe.enter(Button.D4);
    safe.enter(Button.D5);
    assertEquals("Display must be 12345",
                 "12345 ", safe.readDisplay() );
    safe.enter(Button.D6);
  }
```

- ~~Initial: Display reads 6 spaces. Safe is locked.~~
- ~~Enter (key,1,2,3) gives "123 " as output. Safe is locked.~~
- ~~Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.~~
- ~~Enter (1,2) gives "ERROR ". Safe locked.~~
- ~~Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.~~
- Unlocked safe: Enter (lock) gives empty display. Safe locked.
- ~~Accumulate digits in the display when pressing 179.~~
- ~~Enter codes with digits 0, 4, 5, 6, and 8.~~
- Safe goes to initial state if lock button entered.
- Safe returns to initial state after 6 digits entered.
- ~~Enter (1,2,key,1,2,3,4,5,6) gives unlocked safe.~~

2.10 Iteration 10: Locking the safe.

Let us try to lock it again. *Step 1: Quickly add a test*.

```
@Test
public void shouldLockTheSafe() {
  enterCorrectCode();
  assertTrue("Safe must be unlocked", !safe.isLocked());

  safe.enter(Button.LOCK);
  assertTrue("Safe must be locked again", safe.isLocked());
  assertEquals("Display must be cleared",
               "      ", safe.readDisplay() );
}
```

Step 2: Run all tests and see the new one fail. *Step 3: Make a little change* is relatively simple.

```
public void enter(Button button) {
  // Statemachine
  if ( button == Button.LOCK ) {
    enterCodeState = false;
    locked = true;
    setDisplayContents("      ");
  }
  if ( enterCodeState ) { // Safe is expecting a code
    handleButtonInEnterCodeState(button);
  } else { // Safe is not expecting a code
    handleButtonInInitialState(button);
  }
}
```

Step 4: Run all tests and see them all succeed. However, I wish that my code more clearly express the fact that the safe is a state machine that changes state every time the user presses a button. I therefore like the enter method to contain the button event processing. At the moment the code to handle pressing the key button is hidden within the handleButtonInInitialState method.

Again, I like that I have very free hands in experimenting with my code as the test cases developed so far will have a high probability of catching any mistakes in my refactoring. Remember that refactoring means that no changes are made to external behaviour of the system and the test cases are really tests of this external behaviour.

Working a bit with the system, moving code around and making up new helper methods, I end up with code that better express the button event processing nature of the state machine.

```
public void enter(Button button) {
  // Handle state changing buttons
  if ( button == Button.KEY ) {          // KEY
    enterCodeState = true;
    setDisplayContents("      ");
  } else if ( button == Button.LOCK ) {  // LOCK
    enterCodeState = false;
    setDisplayContents("      ");
    locked = true;
  } else {                               // DIGIT
    Character c = map.get(button);
```

```
      if ( enterCodeState ) {
        addCharacterToDisplay(c);
      } else {
        flagError();
      }
    }
  }

  private void addCharacterToDisplay(char c) {
    displayContents[index] = c;
    index = (index+1) % 6;

    if ( readDisplay().equals("123456") ) {
      locked = false;
      setDisplayContents("OPEN  ");
    } else if ( index == 0 ) { // 6 digits entered
      setDisplayContents("CLOSED");
    }
  }

  private void flagError() {
    setDisplayContents("ERROR ");
  }
```

which pass all nine tests.

At this point, I am not satisifed yet, however. The state that exactly six digits have been entered is also a special state, so I would like to show this more precisely in the code as well. Thus I again rework the code to introduce an Enum for the states of the safe. This process took me over one hour—and ended up in *Do Over*. The code became longer, did not pass the test code, and became more complex. Refactoring should increase maintainability, not reduce it. Therefore I threw it away and asked SubVersion to retrieve the snapshot I comitted just before setting commencing the last unsuccessful refactoring adventure.

- ~~Initial: Display reads 6 spaces. Safe is locked.~~
- ~~Enter (key,1,2,3) gives "123 " as output. Safe is locked.~~
- ~~Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.~~
- ~~Enter (1,2) gives "ERROR ". Safe locked.~~
- ~~Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.~~
- ~~Unlocked safe: Enter (lock) gives empty display. Safe locked.~~
- ~~Accumulate digits in the display when pressing 179.~~
- ~~Enter codes with digits 0, 4, 5, 6, and 8.~~
- Safe goes to initial state if lock button entered.
- Safe returns to initial state after 6 digits entered.
- ~~Enter (1,2,key,1,2,3,4,5,6) gives unlocked safe.~~

2.11 Iteration 11: Go to Initial State On Lock Button.

Only a few items on the test list. I pick *Safe goes to initial state if lock button entered.* That is, if the user hits key, 7, 0, 3, and then lock, the safe will switch from 'enter code' state to the initial/waiting/locked state.

```
@Test
public void shouldAbortUnlockingWhenLockButtonHit() {
  safe.enter(Button.KEY);
  safe.enter(Button.D7);
  safe.enter(Button.D0);
  safe.enter(Button.D3);
  assertEquals("Display must be 703",
               "703   ", safe.readDisplay() );
  safe.enter(Button.LOCK);
  assertEquals("Display is cleared",
               "      ", safe.readDisplay() );
  assertTrue("Safe must be stay locked", safe.isLocked());
}
```

This test case actually pass right away. But what if I enter a new proper code right after the above sequence? I better try...

```
@Test
public void shouldAbortUnlockingWhenLockButtonHit() {
  safe.enter(Button.KEY);
  safe.enter(Button.D7);
  safe.enter(Button.D0);
  safe.enter(Button.D3);
  assertEquals("Display must be 703",
               "703   ", safe.readDisplay() );
  safe.enter(Button.LOCK);
  assertEquals("Display is cleared",
               "      ", safe.readDisplay() );
  assertTrue("Safe must be stay locked", safe.isLocked());
  enterCorrectCode();
  assertTrue("Safe must be unlocked", !safe.isLocked());
}
```

And now it fails! The output gives me a strong hint as to what is wrong:

```
1) shouldAbortUnlockingWhenLockButtonHit(TestSafe)
org.junit.ComparisonFailure: Display must be 12345
expected:<[12345 ]> but was:<[45OSED]>
```

The first three digits entered, "703", has advanced our index counter and it is not reset when the user hits the lock button. *Step 3: Make a little change*.

```
public void enter(Button button) {
  // Handle state changing buttons
  if ( button == Button.KEY ) {          // KEY
    enterCodeState = true;
    setDisplayContents("      ");
  } else if ( button == Button.LOCK ) {  // LOCK
    enterCodeState = false;
    setDisplayContents("      ");
    locked = true;
    index = 0;
  } else {                               // DIGIT
    Character c = map.get(button);
    if ( enterCodeState ) {
      addCharacterToDisplay(c);
    } else {
      flagError();
```

```
      }
    }
  }
```

But should I not do the same thing when the key button is hit? Reset the index? Reviewing the test code it seems that test case shouldOpenSafeAfterError test this, but only after an error situation. Better put it onto the test list.

- ~~Initial: Display reads 6 spaces. Safe is locked.~~
- ~~Enter (key,1,2,3) gives "123 " as output. Safe is locked.~~
- ~~Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.~~
- ~~Enter (1,2) gives "ERROR ". Safe locked.~~
- ~~Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.~~
- ~~Unlocked safe: Enter (lock) gives empty display. Safe locked.~~
- ~~Accumulate digits in the display when pressing 179.~~
- ~~Enter codes with digits 0, 4, 5, 6, and 8.~~
- ~~Safe goes to initial state if lock button entered.~~
- Safe returns to initial state after 6 digits entered.
- ~~Enter (1,2,key,1,2,3,4,5,6) gives unlocked safe.~~
- *Safe goes to enter code state whenever key button entered.*

2.12 Iteration 12: Go to Enter Key Code State

I persue the last observation. *Step 1: Quickly add a test*

```
@Test
public void shouldAlwaysGoToEnterCodeState() {
  safe.enter(Button.KEY);
  safe.enter(Button.D7);
  safe.enter(Button.D7);
  enterCorrectCode();
  assertTrue("Safe must be unlocked", !safe.isLocked());
}
```

My intuition was correct:

```
1) shouldAlwaysGoToEnterCodeState(TestSafe)
org.junit.ComparisonFailure: Display must be 12345
  expected:<[12345 ]> but was:<[5LOSED]>
```

The solution is resetting the index.

```
public void enter(Button button) {
  // Handle state changing buttons
  if ( button == Button.KEY ) {          // KEY
    enterCodeState = true;
    setDisplayContents("      ");
    index = 0;
  } else if ( button == Button.LOCK ) {  // LOCK
    enterCodeState = false;
    setDisplayContents("      ");
    locked = true;
    index = 0;
  } else {                               // DIGIT
```

```
      Character c = map.get(button);
      if ( enterCodeState ) {
        addCharacterToDisplay(c);
      } else {
        flagError();
      }
    }
  }
```

This *Step 4: Run all tests and see them all succeed* - but contains duplicated code: the "set index to zero and clear the display contents" appear three places in the code. It would be nice to *Step 5: Refactor to remove duplication* define a method to reset the safe: reset().

Listing: examples/safe/iteration-12/SafeImpl.java

```
import java.util.*;

/** Implementation of the Safe.
*/
public class SafeImpl implements Safe {
  private char[] displayContents;
  private int index;
  private boolean locked, enterCodeState;
  private Map<Button,Character> map;

  public SafeImpl() {
    displayContents = new char[6];
    reset();
    locked = true; enterCodeState = false;

    map = new HashMap<Button,Character>();
    map.put(Button.D1, '1'); map.put(Button.D2, '2');
    map.put(Button.D3, '3'); map.put(Button.D4, '4');
    map.put(Button.D5, '5'); map.put(Button.D6, '6');
    map.put(Button.D7, '7'); map.put(Button.D8, '8');
    map.put(Button.D9, '9'); map.put(Button.D0, '0');
  }

  public void enter(Button button) {
    // Handle state changing buttons
    if ( button == Button.KEY ) {          // KEY
      reset();
      enterCodeState = true;
    } else if ( button == Button.LOCK ) {  // LOCK
      reset();
      enterCodeState = false;
      locked = true;
    } else {                               // DIGIT
      Character c = map.get(button);
      if ( enterCodeState ) {
        addCharacterToDisplay(c);
      } else {
        flagError();
      }
    }
  }

  private void addCharacterToDisplay(char c) {
    displayContents[index] = c;
```

```
    index = (index+1) % 6;

    if ( readDisplay().equals("123456") ) {
      locked = false;
      setDisplayContents("OPEN  ");
    } else if ( index == 0 ) { // 6 digits entered
      setDisplayContents("CLOSED");
    }
  }

  private void flagError() {
    setDisplayContents("ERROR ");
  }

  /** resets the machine to the
   * initial state.
   */
  private void reset() {
    setDisplayContents("      ");
    index = 0;
  }

  public boolean isLocked() {
    return locked;
  }

  public String readDisplay() {
    return String.valueOf(displayContents);
  }

  /** PRECONDITION: string must be exactly 6 charaters long */
  private void setDisplayContents(String sixCharString) {
    for (int i = 0; i < 6; i++)
      displayContents[i] = sixCharString.charAt(i);
  }



}
```

As I cannot find any more tests to put on the test list:

- ~~Initial: Display reads 6 spaces. Safe is locked.~~
- ~~Enter (key,1,2,3) gives "123 " as output. Safe is locked.~~
- ~~Enter (key,1,2,3,4,5,6) gives "OPEN ", safe unlocked.~~
- ~~Enter (1,2) gives "ERROR ". Safe locked.~~
- ~~Enter (key,1,2,4,3,5,6) gives "CLOSED". Safe locked.~~
- ~~Unlocked safe: Enter (lock) gives empty display. Safe locked.~~
- ~~Accumulate digits in the display when pressing 179.~~
- ~~Enter codes with digits 0, 4, 5, 6, and 8.~~
- ~~Safe goes to initial state if lock button entered.~~
- Safe returns to initial state after 6 digits entered.
- ~~Enter (1,2,key,1,2,3,4,5,6) gives unlocked safe.~~
- ~~Safe goes to enter code state whenever key button entered.~~

One left...

2.13 Iteration 13: Entered Code Returns to Initial State

Let us push six digits and test what state the safe is in: enterCodeState or not. The question is how to test it? There is no accessor method for this boolean. Three proposals come to my mind.

- *Modify the* Safe *interface.* Never! Client code has no use for knowing the state of the safe, it breaks encapsulation, and putting additional methods in the interface just for testing purposes lowers cohesion of the abstraction.
- *Adding an accessor method in the* SafeImpl *class.* A bit better but it would mean casting in the testing code along the lines of assertTrue(((SafeImpl) safe). isInEnterCodeState()). This would make the testing code fragile as it is more tightly coupled to the concrete name of the subclass.
- *Use a sideeffect of the state.* I can use my knowledge from the stories that if I enter a digit in the initial/waiting state then the display writes "ERROR" while it does not do so in the enter code state. Thus I can make a test case that enters seven digits and test that the display reads "ERROR."

I choose the last approach. Though the "evidence" is indirect, I do not change any interfaces but only rely on specifications by the users. The *Step 1: Quickly add a test*

```
@Test
public void shouldEnterWaitingStateAfterSixDigitsEntered() {
  // First test with a valid code
  enterCorrectCode();
  safe.enter(Button.D9);
  assertEquals("ERROR ", safe.readDisplay() );
  // Next with an invalid one
  safe.enter(Button.KEY);
  safe.enter(Button.D7); safe.enter(Button.D6);
  safe.enter(Button.D7); safe.enter(Button.D6);
  safe.enter(Button.D7); safe.enter(Button.D6);
  safe.enter(Button.D9);
  assertEquals("ERROR ", safe.readDisplay() );
}
```

This leads to *Step 2: Run all tests and see the new one fail*

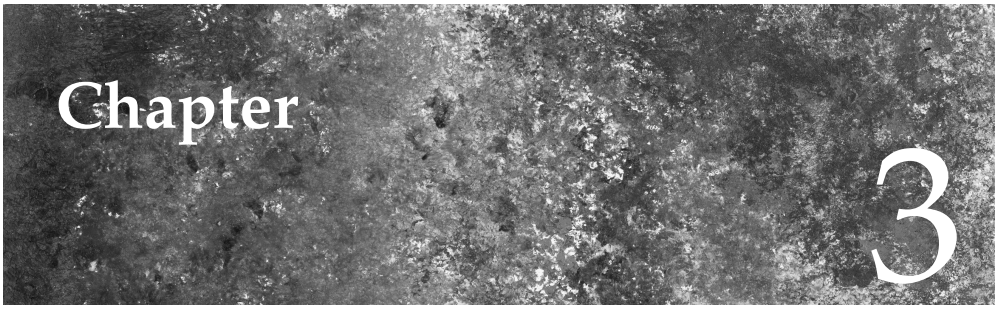
```
There was 1 failure:
1) shouldEnterWaitingStateAfterValidCodeEntered(TestSafe)
org.junit.ComparisonFailure: expected:<[ERROR] > but was:<[9PEN ] >
```

OK, I note that I have actually detected two anomalies in the production code. First the missing behaviour I was hunting—the safe is still in the enter code state. But second the display is not cleared: the "9" appears concatenated with the "PEN" from the open safe statement. (This aspect has actually not been expressed by the stories. As described in Chapter 1 it is envisioned there is some kind of timer that clears the display after some period of inactivity.)

Step 3: Make a little change is easy as I simply add three lines in the end of the enter method.

```
public void enter(Button button) {
  // Handle state changing buttons
  if ( button == Button.KEY ) {          // KEY
    reset();
    enterCodeState = true;
  } else if ( button == Button.LOCK ) {  // LOCK
    reset();
    enterCodeState = false;
    locked = true;
  } else {                               // DIGIT
    Character c = map.get(button);
    if ( enterCodeState ) {
      addCharacterToDisplay(c);
    } else {
      flagError();
    }
    // after 6 digits entered, go to wait state
    if ( index == 0 ) {
      enterCodeState = false;
    }
  }
}
```

And *Step 4: Run all tests and see them all succeed*—all 12 test cases pass.



Discussion

3.1 Is it complete?

TDD is heavily focused on adding features. So one may question if the derived implementation is sound and complete? To get an answer let us look at a state diagram, see Figure 3.1, for the machine and do a more classic up-front analysis. As the state diagram shows, there are basically two states of the safe: "CodeEntering" and "Wait". The wait state is what I called the initial state in the TDD presentation. In any given state the user may hit three types of buttons: key, lock, and digit buttons[1]. Thus the diagram is complete: each state has all three types of state changes associated. In the CodeEntering state there is one additional complication namely that different actions are required depending on if six digits have been entered or not.

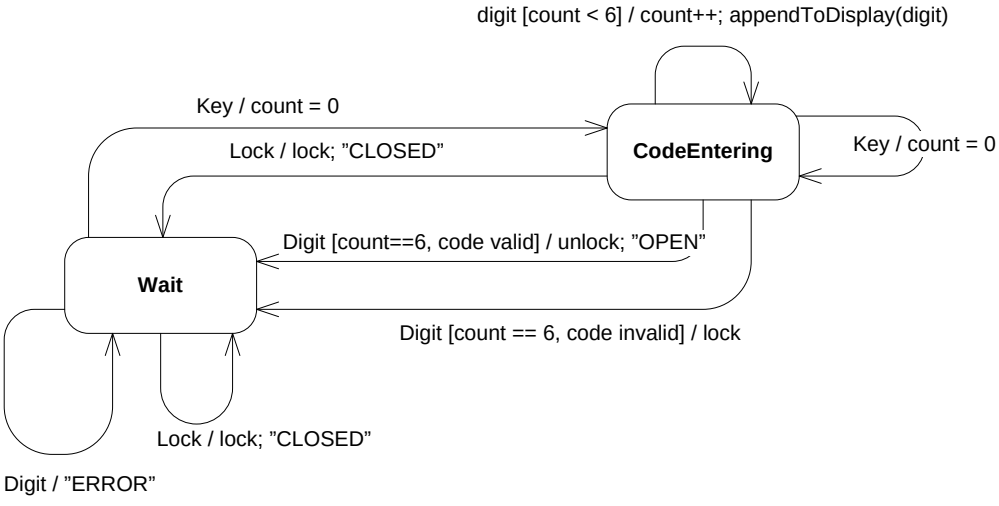


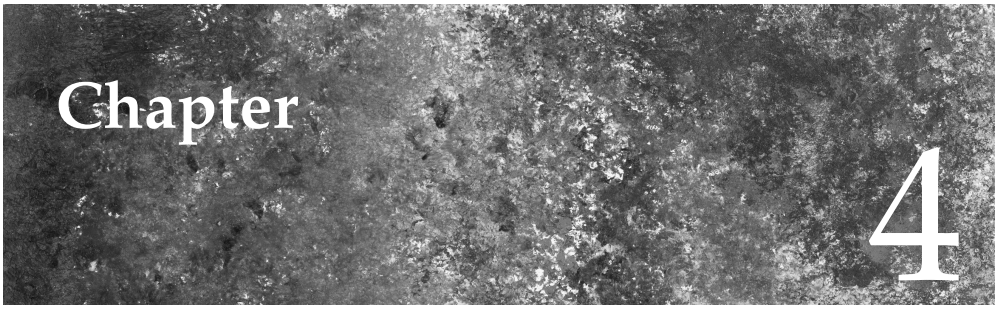
Figure 3.1: State chart (UML) for the safe.

[1] count is the count of digits in the display, and equal to what I called index in the TDD code

Table 3.1: Transitions

State - Event	Test method
Wait - Key	shouldDisplay123CodeAsEntered shouldUnlockSafeOnCorrectCode shouldAccumulateDigitsInDisplay shouldKeepLockedForWrongCode shouldDisplay908CodeAsEntered shouldOpenSafeAfterError shouldLockTheSafe shouldAbortUnlockingWhenLockButtonHit shouldAlwaysGoToEnterCodeState
Wait - Lock	shouldLockTheSafe
Wait - Digit	shouldDisplayERRORWhenForGettingKeyButton shouldOpenSafeAfterError
CodeEntering - Key	shouldAlwaysGoToEnterCodeState
CodeEntering - Lock	shouldAbortUnlockingWhenLockButtonHit
CodeEntering - Digit/count < 6	shouldDisplay123CodeAsEntered (and a lot more)
CodeEntering - Digit/count = 6, code valid	shouldUnlockSafeOnCorrectCode shouldEnterWaitingStateAfterSixDigitsEntered() (and a lot more)
CodeEntering - Digit/count = 6, code invalid	shouldKeepLockedForWrongCode shouldEnterWaitingStateAfterSixDigitsEntered()

Now I can review my test case methods to see if all transitions are covered. As Table 3.1 shows, it seems that my TDD process has actually covered all possible transition within the diagram.



Small Release Review

How can a small release be tested by the users/customer? I consider writing a small Java Swing application resembling Figure 1.1 but find I can make an interaktive prototype much faster that is workable though it is a bit more clumsy. It is run from the shell/prompt and you simply type in digits and either “k” (key symbol) or “l” (lock symbol) followed by “enter”.

Listing: examples/safe/interaktive/Interaktive.java

```
import java.io.IOException;

public class Interaktive {

  public static void main(String[] args) throws IOException {

    Safe safe = new SafeImpl();

    int character;
    System.out.println("Enter digits 0-9; k=key button; l=lock button.");
    System.out.println("Hit Ctrl-C to stop application.");

    while ( (character = System.in.read()) != -1 ) {
      char c = (char) character;
      boolean legal = true;
      Button b = null;
      switch (c) {
      case '0': b = Button.D0; break;
      case '1': b = Button.D1; break;
      case '2': b = Button.D2; break;
      case '3': b = Button.D3; break;
      case '4': b = Button.D4; break;
      case '5': b = Button.D5; break;
      case '6': b = Button.D6; break;
      case '7': b = Button.D7; break;
      case '8': b = Button.D8; break;
      case '9': b = Button.D9; break;
      case 'l': b = Button.LOCK; break;
      case 'k': b = Button.KEY; break;

      default: legal = false;
```

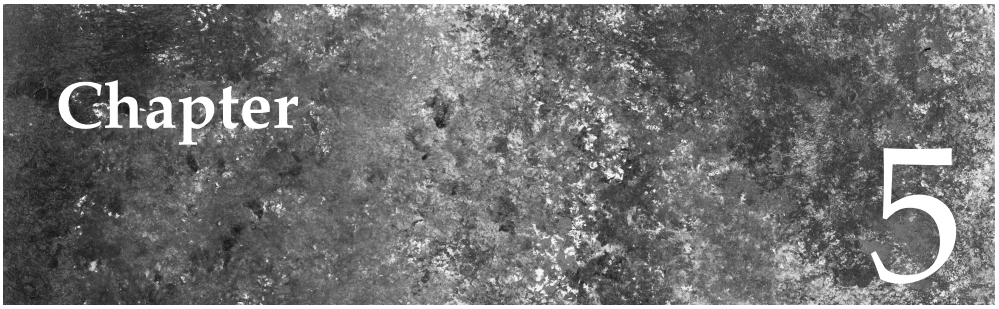
```
      }
      if ( legal ) {
        safe.enter(b);
        System.out.println( "Display: "+safe.readDisplay()+
                            " Locked: "+safe.isLocked() );
      }

    }
  }
}
```

A session using the interaktive prototype may look like this.

```
Enter digits 0-9; k=key button; l=lock button.
Hit Ctrl-C to stop application.
5
Display: ERROR  Locked: true
k
Display:        Locked: true
7
Display: 7      Locked: true
6
Display: 76     Locked: true
5
Display: 765    Locked: true
4
Display: 7654   Locked: true
3
Display: 76543  Locked: true
2
Display: CLOSED Locked: true
k
Display:        Locked: true
1
Display: 1      Locked: true
2
Display: 12     Locked: true
3
Display: 123    Locked: true
4
Display: 1234   Locked: true
5
Display: 12345  Locked: true
6
Display: OPEN   Locked: false
8
Display: ERROR  Locked: false
l
Display:        Locked: true
```

First I test the error situation, next an invalid pin code followed by a legal one, to conclude by locking the safe again.

In this case, the review passes but I have often experience that when you do TDD you may overlook some important test cases that only appears at integration testing time or, as here, when you do some functional/system testing with the end users. If you run into this (and I bet you will) do not consider it a major flaw of your TDD proces but rather remember that once a defect is detected that is not covered by your unit test cases, then start by adding a test case that reproduces the defect *before* you correct it in the production code.



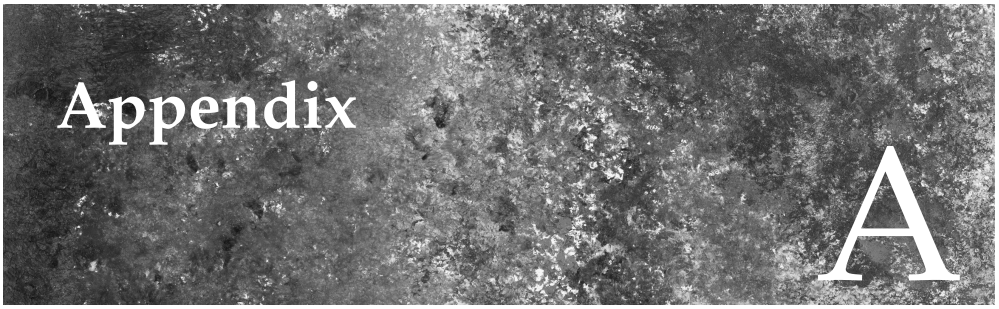
Further Exercises

Exercise 5.1:

Implement Story 5 using TDD.

Exercise 5.2:

Rewrite the implementation to use the STATE pattern to implement the state machine.



All Stories

See also comments in Chapter 1.

Story 1: Unlock Safe The user approaches the safe whose door is locked. The display is empty, which means it contains 6 spaces/blanks. The user hits the key-symbol button. The user enters his previously stored pin code by pressing the buttons one at the time: “1”, “2”, “3”, “4”, “5”, “6”. The display reacts by writing each digit as it is pressed. After the final “6” button press, the display clears and displays “OPEN ”. The safe door unlocks and can be opened.

Story 1a: Regret Opening Safe The user starts entering the proper pin code, “1”, “2”, “3”, but regrets to open the safe. The user hits the lock button, the display clears, and the safe remains locked.

Story 2: Lock Safe The safe door is unlocked. The display reads “OPEN ”. The user closes the door and presses the lock button. The door locks. The display reads “CLOSED”.

Story 3: Forgetting key Button The safe is locked. The user forgets to hit the key button first and hits “1”. The display reads “ERROR ”. All following button hits result in the display reading “ERROR ”, unless the key botton is pressed.

Story 4: Wrong Code The safe is locked. The user hits key followed by 1 2 4 3 5 6. The display is cleared. The safe remains locked.

Story 4a: Completed Pin Code The user enters a 6-digit pin code (valid or invalid). The safe returns to the initial/waiting state.

Story 5: Set New Code The safe is open/unlocked. The user hits the pin button, enters a new six digit pin code, “777333”, and finally hits the pin again. The safe’s display reads “CODE ”. It remains unlocked. After locking, the safe can only be unlocked (see story 1) by entering the new pin code “777333”.