Chapter 4

Software Design using Test Driven Development (TDD)

Chapter Goals

- Consider one way to develop software
- Become familiar with Test Driven Development and JUnit
- Experience refactoring

4.1 Iterative Development

This chapter introduces some techniques that help programmers develop high quality software that is easy to understand, modify, and maintain. There are many software development processes that could be presented here. For example, the “waterfall” approach suggests analysis and gathering of requirements (figuring out everything the system must do) before all else. Then the system is designed, which requires a lot of documentation. On large projects, analysis and design can take months (and even years) before a single line of code gets written. Discrete steps are imposed that require analysis to be completed before design, design before coding, and coding before testing.

This waterfall process also has some severe problems. The development team has to be almost clairvoyant, with little room for error. Waterfall places the high risk and difficult tasks toward the end of the project, when it can be too late to debug them.

Craig Larman's book[1] demonstrates that waterfall has proved to be a terrible way to develop software. In one study where 87% of such projects failed, waterfall was the "single largest contributing factor for failure being cited in 81% of the failed projects as the number one problem." The author jokes about a guy who fell off a cliff:

> *As he was hurtling down, someone at the top of the cliff yelled to the guy who was falling, "How are you doing?" The guy replied, "So far, so good!"*

Recently industry and academia have begun to adopt an *agile* approach to software development. Agile processes still have analysis, design, coding, testing, and documentation; however these methods emphasize *working software* as the primary measure of progress. Software developers work with customers—those wanting the software— in order to prioritize what is most important. Developers code and test these features. Design is employed, but throughout the whole process. Testing is done, but testing is done all along.

---

[1] Agile and Iterative Development: a Manager's Guide, Addison-Wesley Professional, 2003

4.2 Find the Objects

Some agile practices will now be demonstrated in the context of building a software system. This narrative describes what the system should be able to do.

> The student affairs office has decided to put some newfound activity fee funds toward a music jukebox in the student center. The jukebox will allow a student to play individual songs. No money will be required. Instead, a student will swipe a magnetic ID card through a card reader. Students will each be allowed to play up to 2,000 minutes of "free" jukebox music in their academic careers at the school. A student may select a song from the available collection. The user must be able to look up songs by artist and see how long any song takes to play. No user may play more than two songs on any given date. No song may be played more than five times on the same date, even if six different users try to play the same song.

One way to begin building this system is to first find the objects. Nouns are potential objects. We are looking for "things" that may model the desired system. This is not the time to worry about too many details, or even if you may be right or wrong. A first pass at finding the objects will represent a list that may end up as part of the system. Others will be eliminated. And still others may be found later.

A List of Candidate Objects

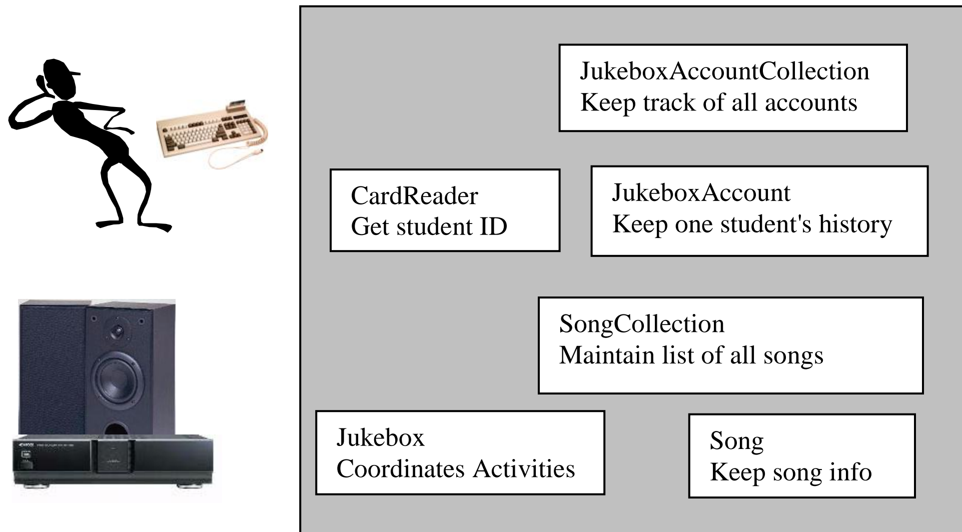
Jukebox	ID card
Student	Collection
Song	Date
Card reader	

While trying to discover useful objects, it helps to eliminate redundant entries. There is no useful purpose for writing down *student* more than once, for example. Also, you should include only those objects that have meaning within the realm of the system. Activity fee funds provide the money to pay for the Jukebox. However, fees and money need not be developed. We do not need to build the student center or keep track of the student's academic careers. The seven candidate objects listed above will do for now.

It is also important to name things as accurately as possible. Although the name Student appears many times, Student sounds more like an object that might represent the academic career. `JukeboxAccount` better represents this new type. `SongCollection` is more meaningful than collection. It is also good to recognize those things that are outside of the system that needs to be developed. The software may play a particular song over the stereo system, but we do not need to model, nor build, the stereo system—this is outside the system. It also seems like we could use a way to store all `JukeboxAccount` objects, so it is reasonable to list another candidate object named `JukeboxAccountCollection`.

The following picture represents another view of these candidate objects and their main responsibilities. It also recognizes other parts, such as a physical stereo system, a card reader, and a user, all part of the final system to be controlled by the software we develop.

Candidate Objects that Model a Cashless Jukebox (the big picture)



It helps to consider the main responsibility for each object. The responsibilities convey the purpose of an object and its role in the system. The proper assignment of responsibilities is an important skill developed over time. It can be fuzzy at first. People on the development team may disagree. It is not exact science; it is more about common sense and what makes sense to most people. While working on this system, consider that each object is responsible for two things:

1. What should an object be able to do?
2. What should an object remember?

These responsibilities can be implemented as the methods and the instance variables of a class.

4.3 Design and Test a new Type with TDD

There are still design decisions to be made, and we'll need a way to know that the classes are doing what they are supposed to be doing. To do this, we'll use test driven development (TDD).

The rules of TDD

1. Write the tests first with assertions that indicate the desired behavior.
2. Run the tests to see the new test fail.
3. Write code that makes the test pass.
4. Refactor (makes small changes to code to improve it)

The classes in this chapter will be developed with the aid of the JUnit testing framework developed by Erich Gamma and Kent Beck. JUnit is a design and testing tool that is part of virtually all Java integrated development environments (IDE’s). This includes IDEs designed for introductory course such as BlueJ or Dr Java. It also includes IDEs for professional software developers such as NetBeans, IntelliJ, and Eclipse. Other testing frameworks that aid software

development also exist for designing and testing software in other popular programming languages such as PHP, C++, C#, and VB.NET.

Develop (design and test) One Class, One Method at a Time

I will be specifying functionality as JUnit tests. The minimal code needed to write tests begins as a new class named `SongTest`. To gain access to this testing and development tool, two imports from the org.junit package are needed: `Assert` and `Test`.

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

import org.junit.Test;

public class SongTest {

}
```

Self-Check

4-1 In this chapter, it helps to type in the code as you are reading. The self-checks for this chapter are limited to typing in any new code. I indicate the new code to be added in boldface within the gray background. Begin with the unit test named `SongTest` started above and type in the code as you encounter it. In Eclipse, select **File > New > JUnit Test Case** and name it `SongTest`. At the top of the wizard, choose JUnit 4.0 and click on the link **Click here** at the bottom to add JUnit 4 to the build path (see screenshot). In the new wizard, select JUnit 4 (see screenshot).

1. Write the Test

To allow users to identify the song they want to play, we'll begin with making sure each Song can remember its own title and artist. First, let's set up the test.

The JUnit testing framework version 4.0 uses the `@Test` annotation to mark a method as a test. You must declare a method as `public`, with a return type of `void` (which means the method returns nothing). Because older versions of JUnit required that the test method name must begin with the letters `test`, the method names will begin with `test` (it's an old habit). Here is one valid test method written as a stub (no code is yet in the body of the method):

```
  @Test
  public void testGetters() {

  }
```

Self-Check

4-2 Add the test method above to `SongTest`.

The test name `testGetters` indicates that the method will ensure we can get the artist and the title from two different songs. I recommend you think of two different objects right away rather than just one. I'm thinking of "Help" by the Beatles and "Good is Good" by Sheryl Crow. The test

method begins with what we want to happen. The code has many compile-time errors that will be addressed later.

```
@Test
public void testGetters() {
  // We have not even constructed aSong or anotherSong yet. And there is no
  // getArtist or getTitle method yet, not even a Song class!
  // But here is what I think I need at this point:
  assertEquals("Beatles", aSong.getArtist());
  assertEquals("Help", aSong.getTitle());
  assertEquals("Sheryl Crow", anotherSong.getArtist ());
  assertEquals("Good is Good", anotherSong.getTitle());
}
```

Self-Check

4-3 Add the four assertions to the test method above. There will be many compile-time errors.

Consider what is needed to construct `Song` objects. We know the name of the constructor should be `Song()`, but we have to decide which (if any) values should be sent to the constructor as arguments. The test method shows that we decided to have a title and an artist. Each `Song` object will need its own artist and title, so we'll pass that data as arguments to the constructor.

```
@Test
public void testGetters() {

  Song aSong = new Song("Beatles", "Help");
  Song anotherSong = new Song("Sheryl Crow", "Good is Good");

  assertEquals("Help", aSong.getTitle());
  assertEquals("Beatles", aSong.getArtist());
  assertEquals("Good is Good", anotherSong.getTitle());
  assertEquals("Sheryl Crow", anotherSong.getArtist ());
}
```

Self-Check

4-4 Add the two object constructions to the test method above in `SongTest`. There are even more compile-time errors.

At this point, I am just writing what I think I need and not worrying about the compile-time errors because we'll fix those compile time errors next.

2. Make the test fail

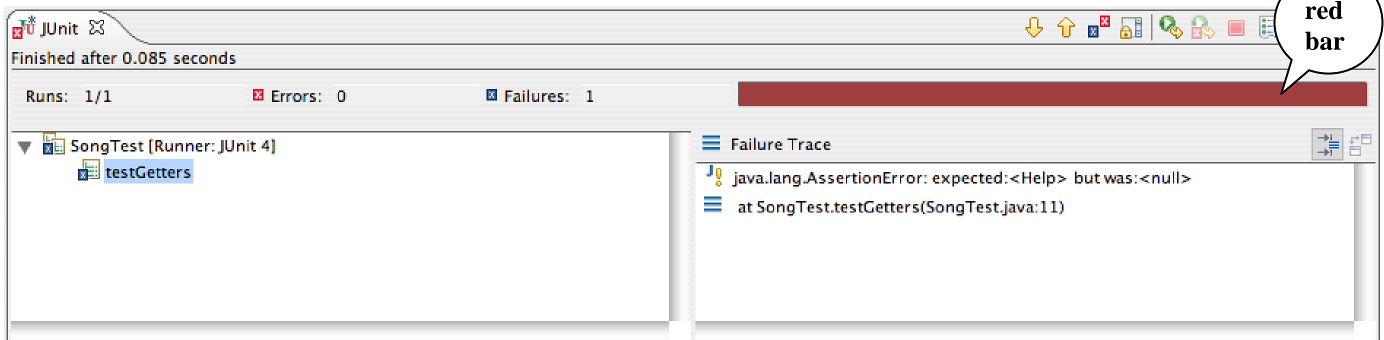
To make the test fail, the code must compile. The following minimal `Song` class removes the compile-time errors as long as a newly designed constructor is included. The method names, parameters and return type must match.

```
public class Song {

  public Song(String songArtist, String songTitle) {

  }

  public String getTitle() {
    return null;
  }

  public String getArtist() {
    return null;
  }
}
```

Self-Check

4-5 Create a new class named `Song` and add the constructor and the two methods as shown above. You can return any string for `getTitle` and `getArtist` (I chose to return `null` as would Eclipse if you let Eclipse write the code for you)

Now running the unit test will show that the `getTitle()` method does not return the correct value (no surprise). We expected the string value `"Help"`, however the actual value intentionally returned from the method is `null`.



With compile-time errors fixed and the test failing, I now want to make the tests pass.

3. Makes the test pass

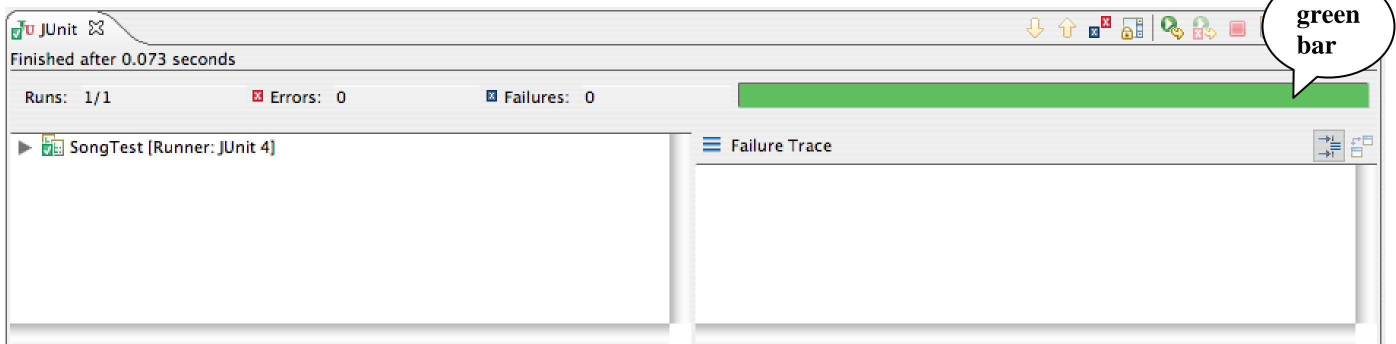
We need a way for every `Song` object to remember its own artist and title. Different instances of the same class remember their own state through instance variables. Specifically, adding a `title` instance variable to the `Song` class allows many `Song` objects to have their own titles. The constructor must also assign the argument to the correct instance variable. Additionally,

`getTitle()` must return that remembered value. These changes are shown below in the updated `Song` class that will make both assertions pass (thus, you should see the green bar in JUnit).

```
public class Song {

  // Instance variables to remember the artist and title
  private String artist;
  private String title;

  public Song(String songArtist, String songTitle) {
    // Let each object remember these values
    artist = songArtist;
    title = songTitle;
  }

  public String getTitle() {
    return title;
  }

  public String getArtist() {
    return artist;
  }
}
```

Self-Check

4-6 For the remainder of this chapter, add the highlighted code. There will be no more explicit self-checks. After adding the six highlighted lines of code to class Song as shown above, you should have the test pass.



4.4 Play a Song Only Five Times per Day

One of the requirements for this `Jukebox` system maintains that no song should be played more than five times on a given date. One way to manage this is to have each `Song` correctly respond true or false when asked if it can be played today. This means we are making the `Song` responsible for this feature.

1. Write the test

The following test method presents one way to do this. Test Driven Development is helping us to design the class. It specifies a new method, its name, the return type, and arguments (0 here).

Add this to SongTest

```
@Test
public void testCanPlayTodayAtFirst() {
  // Should be able to play this Song once
  Song aSong = new Song("Beatles", "Help");
  assertTrue(aSong.canPlayToday());
}
```

Your programming environment should have several errors since `canPlayToday` does not yet exist.

2. Make the test fail

The following additions to Song will make everything compile. Because `canPlayToday` returns false, the new test will fail.

```
public class Song {

  // ...

  public boolean canPlayToday() {
    return false;
  }

}
```

3. Make the test pass

By keeping track of how many songs can be played, `canPlayToday` can compare that to 5 to see if the song could be played again. With less than 5 plays, the method should return true. I begin by declaring a public named constant, which is a value that cannot be changed while the program runs. The code can later on use the more meaningful `MAX_PLAYS_PER_DAY` rather than 5, as a

"magic number". We also need to maintain how frequently each `Song` has been played. The instance variable `songsPlayedToday` gets initialized to 0 in the Song's constructor.

```
public class Song {
  // ...

  public static final int MAX_PLAYS_PER_DAY = 5;

  private int songsPlayedToday;

  public Song(String songArtist, String songTitle) {
    // ...
    songsPlayedToday = 0;
  }

  // ...

  public boolean canPlayToday() {
    return songsPlayedToday < MAX_PLAYS_PER_DAY;
  }
}
```

Now the new test will pass. I am obviously not done. What happens if a song has been played 5 times on the date it gets asked to `canPlayToday`? I write a test to describe what *should* happen.

1. Write the Test

Because some method is needed to let the `Song` know that it has been played, this test shows a need for a new method named `recordOnePlay`.

```
@Test
public void testCanPlayTodayWithUpTo5Plays() {
  Song aSong = new Song("Beatles", "Help");

  aSong.recordOnePlay();
  assertTrue(aSong.canPlayToday());
  aSong.recordOnePlay();
  assertTrue(aSong.canPlayToday());
  aSong.recordOnePlay();
  assertTrue(aSong.canPlayToday());
  aSong.recordOnePlay();
  assertTrue(aSong.canPlayToday());

  aSong.recordOnePlay();
  // This song has been played 5 times already
  assertFalse(aSong.canPlayToday());
}
```

The assertions above suggest what should happen during the messages: `canPlayToday` returns true until there have been five `recordOnePlay` messages. Some other object will likely use these two new methods to either help play the Song or to notify the user that the `Song` has been played its maximum number of times.

2. Make the test fail

First get the code to compile by adding method `recordOnePlay` to `Song`. Since `recordOnePlay` is a void method, it need not return a bogus value to make the test fail.

```
public void recordOnePlay() {

}
```

3. Make the test pass

Without any thought about the day upon which the `Song` was played, we can get `canPlayToday` working (test passes) by incrementing the instance variable that keeps track of how often the song has been played inside this new `recordOnePlay` method.

```
public void recordOnePlay() {
  songsPlayedToday++;
}
```

However, this will do precisely what the name implies. Keep track of how often this song has been played at any time since the `Song` was constructed. This may be good for a top ten list, but as the name suggests, `songsPlayedToday` does not yet do what its name reveal it should do. `songsPlayedToday` must maintain a number that starts at 0 at midnight and gets up to 5 maximum during the same calendar date. Just adding 1 to `songsPlayedToday` every time won't do this. We need to consider the calendar date upon which the `canPlayToday()` message is sent and also the days upon which the song has been played in the past.

One solution is to keep a list of all dates upon which the song was played. This requires a way to store the list, a way to add to the list during `recordOnePlay()` and an algorithm to find all dates in this. This is previewed with a new private helper method and a small change to `canPlayToday` that sends a `songsPlayedToday()` message to itself rather than access the instance variable of the same name (if you are adding code, don't change this just yet, the changes are shown two pages from here).

```
public boolean canPlayToday() {
  return songsPlayedToday() < MAX_PLAYS_PER_DAY;
}
private int songsPlayedToday() {
  // TODO: Search for today in the list of dates played in the past
  return 0;
}
public void recordOnePlay() {
  // TODO: Add today's date to the list of dates played
}
```

We can use an array to store all dates upon which a `Song` played. Instead of developing a whole new type for managing dates, we'll use the existing `java.util.GregorianCalendar`.

GregorianCalendar

A `GregorianCalendar` object allows programmers to store dates in a culture that uses the Gregorian calendar instituted in October 15, 1582 under Pope Gregory. To demonstrate, here is some code that constructs an empty array and adds three GregorianCalendar objects.

```
// Create an empty list that can store GregorianCalendar objects
GregorianCalendar[] datesPlayed = new GregorianCalendar[5];
int n = 0;

// Add 3 dates objects as set in the computer's clock
datesPlayed[n] = new GregorianCalendar();
n++;
datesPlayed[n] = new GregorianCalendar();
n++;
datesPlayed[n] = new GregorianCalendar();
```

The for loop prints the `getTime()` version of each `GregorianCalendar` object reference at indexes 0, 1, and 2 of `datesPlayed`. The `getTime()` method returns a string with the time and date when the `GregorianCalendar` object was contructed.

```
for (int index = 0; index < 3; index++) {
  System.out.println(datesPlayed[index].getTime());
}

Fri Aug 11 19:46:35 MST 2006
Fri Aug 11 19:46:35 MST 2006
Fri Aug 11 19:46:35 MST 2006
```

The output shows that the new `GregorianCalendar` objects remember the precise year, month, day, and even the millisecond at which they were constructed. An array of GregorianCalender objects allows all `Song` objects to remember the five most recent dates it was played.

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Song {
  // ...

  private GregorianCalendar[] datesPlayed;
  private int n;
```

```
  public Song(String songArtist, String songTitle) {
    // ...
    datesPlayed = new GregorianCalendar[MAX_PLAYS_PER_DAY];
    n = 0;
  }

  // ...

  // Add the current date to this Song's list of when it has played
  public void recordOnePlay() {
    datesPlayed[n] = new GregorianCalendar();
    n++;
  }

  public boolean canPlayToday() {
    return songsPlayedToday() < MAX_PLAYS_PER_DAY;
  }

  private int songsPlayedToday() {
    // Determine the moment at which this method begins
    GregorianCalendar today = new GregorianCalendar();
    // Find how often today equals a day in the array
    int result = 0;
    // Examine every date this Song has been played
    for (int i = 0; i < n; i++) {
      if (today.get(Calendar.YEAR) == datesPlayed[i].get(Calendar.YEAR)   &&
          today.get(Calendar.MONTH) == datesPlayed[i].get(Calendar.MONTH) &&
          today.get(Calendar.DAY_OF_MONTH) ==
                              datesPlayed[i].get(Calendar.DAY_OF_MONTH)) {
        result++;
      }
    }
    return result;
  }
}
```

The new method `songsPlayedToday` has a rather bizarre way to see if one dates "equals" another. This is necessary because the `equals` method for `GregorianCalendar` returns `false` if two different `GregorianCalendar` objects differ by as little as 1 millisecond, and it is likely that two would be constructed at different milliseconds since the Epoch (1-January-1970 00:00:00.000 GMT).

`public boolean` **`equals`**`(Object obj)`

> Compares this `GregorianCalendar` to the specified `Object`. The result is `true` if and only if the argument is a `GregorianCalendar` object that represents the same time value (millisecond offset from the Epoch) under the same `Calendar` parameters and Gregorian change date as this object.

Therefore the dates in the array are compared to current year, month, and day using `get` messages. A get message is used to return the value of one of many of fields for the object

`public int` **`get`**`(int field)`

Returns the value of the given calendar field.

The `java.util.Calendar` class declares over 60 fields declared as class constants that can be used in the `get` method such as these:

`Calendar.DAY_OF_MONTH`

Field number for `get` and `set` indicating the day of the month.

`Calendar.HOUR`

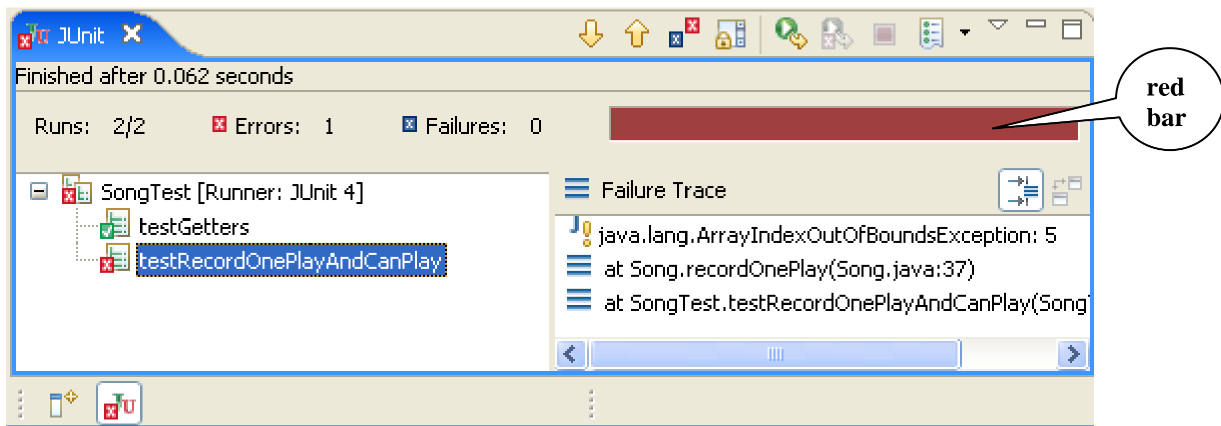
Field number for `get` and `set` indicating the hour of the morning or afternoon.

`Calendar.MILLISECOND`

Field number for `get` and `set` indicating the millisecond within the second.

If these three fields are the same as `today`'s year, month and day, then the array value "equals" the current date (actually there are simpler ways, but let's move on).

This new test still passes. This first attempt may appear to work—you get the green bar. However the list will grow to 5 larger at another `recordOnePlay` message. At that time, there will be an attempt to add to a filled array.



We could declare the array to be bigger, say 10, or maybe 100, or maybe the number of times any one song could be played. But this is difficult to say. I will arbitrarily pick 750.

```
  datesPlayed = new GregorianCalendar[750];
```

Eventually, the list will contain `GregorianCalendar` objects from many different dates (assuming users are actually selecting that song). But what happens if one song gets played more than 750 times? Do we need to really store more than the most recent 5 dates?

4.5 Refactoring

Refactoring is a term applied to making small changes to existing code that does not change its behavior. Refactorings are intended to make the code more readable; easier to understand, and easier to maintain. There currently exists a long complicated boolean expression in

`songsPlayedToday`. To maker the code more readable, we could do a refactoring known as extract method. Here is the desired change made to the method.

```
// Let someone know how many times this Song has played on the current date
public int songsPlayedToday() {
  GregorianCalendar today = new GregorianCalendar();
  int result = 0;
  for (int i = 0; i < n; i++) {
    if (sameDay(today, datesPlayed[i]))
      result++;
  }
  return result;
}
```

The complex Boolean expression is relegated to a `private` method in the `Song` class. The code is no shorter, but the loop above is easier to understand (assuming you can deduce when `sameDay` returns `true` or `false`).

```
private boolean sameDay(GregorianCalendar today,
                        GregorianCalendar dateToCompare) {
  return   (today.get(Calendar.YEAR) == dateToCompare.get(Calendar.YEAR)
        && today.get(Calendar.MONTH) == dateToCompare.get(Calendar.MONTH)
        && today.get(Calendar.DATE) == dateToCompare.get(Calendar.DATE));
}
```

Small changes like this can make code easier to understand. After a refactoring we can run all tests for `Song` to verify the behavior has not changed, that all methods still work as desired.

Refactor Again?

Another issue is the fact that the array to store dates played will keep growing and growing. We don't need any more than 5 dates. Storing a 6th or 7th date is useless. All we need to do is make sure the array always has 5 or fewer dates by removing the oldest date when appropriate (and reducing n). This will make `recordOnePlay` more complicated, but it will reduce the size of the array to 5 to save memory. It also avoids the nasty possibility that years from now, when a Song has been played 750 times, the system will crash with an array out of bounds exception. However, this solution is complex, especially compared to another way to solve this problem.

Imagine that each Song is made to remember the most recent date on which it was played. Then during a `recordOnePlay` message, the most recent date played can be compared to the current date. If these two are the same date, simply increment number of plays by 1. If these two are not the same date, this is the first play of the current date, which should be recorded for future comparisons.

```
if current date equals the most recent date played
   increment the number of plays today by 1
else {
   set most recent date played to the current date
   set songs played to 1
}
```

Now, rather than an array of dates, we just need one date. Initially it gets set to a date way in the past so it won't get confused with the day when the Jukebox first starts up. `songsPlayedToday` becomes a simple `int` again.

```
public class Song {

  private GregorianCalendar mostRecentDatePlayed;
  private int songsPlayedToday;

  public Song(String songArtist, String songTitle) {
    artist = songArtist;
    title = songTitle;
    mostRecentDatePlayed = new GregorianCalendar(1900, 1, 0);
    songsPlayedToday = 0;
  }

  // Determine if this Song has been played enough
  public boolean canPlayToday() {
    return songsPlayedToday < MAX_PLAYS_PER_DAY;
  }
```

Now the `recordOnePlay` method can keep `songsPlayedToday` as the correct value.

```
  // Maintain how often this song has been played on any date.
  // Precondition: This song can be played today.
  public void recordOnePlay() {
    GregorianCalendar today = new GregorianCalendar();
    if (sameDay(today, mostRecentDatePlayed))
      songsPlayedToday++;
    else {
      songsPlayedToday = 1;
      mostRecentDatePlayed = today;
    }
  }
```

Self-Check

4-7 Assuming you are a U of Arizona student, your instructor is using Web-Cat, and there is a project available for submission, submit your project to Web-Cat to ensure you tested all code, (all of the reference tests pass) and all of your assertions in your instructor's reference test pass.

Web-Cat instructions for Eclipse are on the next page

Using Virginia Tech's WebCat from the University of Arizona:

- View this Submission Video: http://web-cat.cs.vt.edu/WCWiki/SubmissionWalkthrough
- Read your email to get your Web-Cat password (you can change this after first login).
- Login at this url: https://web-cat.cs.vt.edu/Web-CAT/WebObjects/Web-CAT.woa
- You can submit your Eclipse Archive from this web site or get this Eclipse plugin so you can stay in Eclipse.
    - **OPTIONAL** Download the zip file from http://sourceforge.net/project/showfiles.php?group_id=142064&package_id=160235
    - Unzip and copy its 3 folders go into the Eclipse plugin folder
    - Restart Eclipse
- From Eclipse, select Window > Preferences > Electronic Submission (or with a Mac, select File > Preferences > Electronic Submission Preferences).
- Fill in the four fields using this for the Assignment definition URL (Assigment definition URL is repeated here to read easier and/or to copy and paste

https://web-cat.cs.vt.edu/Web-CAT/WebObjects/Web-CAT.woa/wa/assignments/eclipse?institution=Arizona