Test Driven Development (TDD)

Arjumand Ateeq - last updated October 2016

Any time we design a new product, be it physical or digital in nature, we start by listing out the requirements. What do we want the product to do? In formal engineering, there are elaborate processes that capture these requirements, then trickle these down to development engineers who fulfill those requirements, and testing engineers who ensure the requirements were, in fact, fulfilled. A consistent problem across disciplines is the clarity lost between communication cycles. Requirements and test management tools help to manage the complexity of this communication, but the truth about how the system actually behaves still lies in one place, and one place only - the source code.

If requirements drive how the system should behave, and tests ensure the system behaves as expected, immense clarity and reliability can be achieved if the code is driven directly by the tests that ensure the requirements have been met. That is, instead of the **Requirements <-- Code <-- Test** chain of events, we are establishing a different kind of link between these phases: **Requirements <-- Test <-- Code**. When we think along these lines, we are engaging in behaviour-driven development, guided by tests, or simply, Test Driven Development.

In the Agile world, acceptance criteria for stories is often written using Given/When/Then clauses. For example, Given user A likes item X and user B likes item Y, When user A and user B are similar, Then recommend item Y to user A. The complete set of Givens, Whens and Thens form the states, events and resulting scenarios that the new product must fulfill[1]. If these scenarios are translated into tests using the same kind of language, the tests become the specifications for the system; they describe the behaviour that system is expected to exhibit. All that is left to do, then, is write just enough code to have the system actually exhibit this behaviour. Working in this way solves a number of problems:

1. Testing no longer requires leaps of faith to determine whether or not we have written enough tests to satisfy all the requirements. The tests simply list out all the Given/When/Then combinations as scenarios.
2. The availability of requirements right inside the code helps keep the implementation focused. The amount of code written is just enough to fulfill the requirements - no more, and no less. This helps to maintain a clean code base, and minimizes development time.
3. The system's design is often the simplest one possible. Rather than designing upfront, we let the design emerge as we fulfill more and more requirements.
4. The system's design is influenced by the expected user experience, rather than the system design driving the user experience. Assuming the high-level requirements define the expected user experience, working from these high-level requirements in a top-down manner naturally evolves the design based on the expected user experience.

The clarity with which the code can now be written brings about a sense of joy you can only understand when you experience it for yourself, by practicing TDD in a behaviour-driven way. In the rest of this post, I will evolve a simple recommendation system, using unit tests to specify the behaviour and guide my implementation. My goal is to help you see that behaviour-driven TDD is how software development should really be done. The simple rule to follow for TDD is to work in the cycle: Add a little test, run all tests and fail, make a little change, run the tests and succeed, refactor to remove duplication. I will assume you already know the basics of how to work this way. If you need more practice with this, a great resource is Kent Beck's book, *Test-Driven Development*[51].

Development Environment

I have used the following setup to work through this example.

Programming language: Java 8
Unit testing library: JUnit 4.12, and Hamcrest 1.3
IDE: Eclipse Neon
Dependency management: Apache Maven 3.3.9.
Version control: GitHub

The complete source code is available on Github at:
https://github.com/ethereality/RecommendationSystem/tree/v1.0

Using your IDE, you can start with a new maven project, with a pom file similar to the one below:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.recommender</groupId>
  <artifactId>recommend</artifactId>
  <version>0.1</version>
  <packaging>jar</packaging>

  <name>recommend</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
      <!-- https://mvnrepository.com/artifact/junit/junit -->
      <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.12</version>
      </dependency>
      <!-- https://mvnrepository.com/artifact/org.hamcrest/hamcrest-all -->
      <dependency>
```

```
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-all</artifactId>
            <version>1.3</version>
        </dependency>
    </dependencies>
</project>
```

All the source code classes are added to src/main/java, and all the test classes are added to src/test/java, in a package structure that is identical to the source code classes. For example, if a source class called *Engine* is added as src/main/java/recommender/main/Engine.java, then the corresponding test class exists as src/test/java/recommender/main/EngineTest.java.

Thinking Through the Problem

Imagine asking a close friend for a book recommendation. Your friend will likely know your taste well and therefore recommend books he thinks you will like. But how does he know what you will like? If he knows what genres you like to read, he will think of the books he has read that fall into that genre. Of these books, he may have liked some, and not liked others. He will likely only recommend the books he has liked. But how likely are you to also like a book that your friend likes? A gross generalization could be that the more similar your tastes, the more likely you are to like the same books. In fact, the more similar your tastes, the more likely you may be to dislike the same books. Let's list these thoughts out more generically.

1. An item should be recommended to a person if there is a high enough probability that person will like it.
2. The probability of a person liking an item depends on the similarity of that person to other people who have liked or disliked that item.
3. Two people are similar to each other if they have more liked items in common than disliked items.
4. In order to get recommendations, a person must find other similar people. To find other similar people, the person must declare her own likes and dislikes for some number of items.

At this point, we can start to see some of the concepts needed to implement this recommendation system. For example, we need a way to capture likes and dislikes, and a way to calculate similarity between users. We could start by listing all the concepts we can think of, together with the dependencies between them. But this design will inevitably change as we understand the current requirements better, and as those requirements change. Rather than trying to keep our design document up to date, let's try to focus on what we really want to do - fulfill requirements. We will let the design emerge as we specify our requirements, and then write just enough code to fulfill those requirements.

Listing High-Level Requirements

Having thought through the problem a little, we can come up with the high-level requirements for our recommendation system.

Provide Ability to Add Users and Items

1. Given a user identified by name, When the user is added for the first time, Then user gets added successfully.
2. Given two users identified by the same name, When both users are added, Then user is recorded only once.
3. Given an item identified by name, When item is added for the first time, Then item gets added successfully.
4. Given two items identified by the same name, When both items are added, Then item is recorded only once.

Provide Ability to View Users and Items

1. Given no users exist in the system, When system is queried for users, Then no users are returned.
2. Given one or more users exist in the system, When system is queried for users, Then all users in system are returned.
3. Given no items exist in the system, When system is queried for items, Then no items are returned.
4. Given one or more items exist in the system, When system is queried for items, Then all items in the system are returned.

Provide Ability for Users to Rate or Unrate an Item

1. Given an item not rated by a user, When the user likes that item, Then a like is recorded for that user-item pair.
2. Given an item not rated by a user, When the user dislikes that item, Then a dislike is recorded for that user-item pair.
3. Given an item liked by a user, When the user dislikes that item, Then only a dislike is recorded for that user-item pair.
4. Given an item disliked by a user, When the user likes that item, Then only a like is recorded for that user-item pair.
5. Given a user who has liked or disliked an item, When the user unrates that item, Then no rating record exists for that user-item pair.

Provide Ability to Get Recommendations

1. Given no items exist in the system, When a user asks for recommendations, Then no recommendations are provided.
2. Given only one user exists in the system, When that user asks for recommendations, Then no recommendations are provided.
3. Given a user who has not rated any items, When that user asks for recommendations, Then no recommendations are provided.
4. Given user A is similar to user B and user C, When user A asks for recommendations, Then all items liked by user B and by user C and not yet rated by user A are recommended.
5. Given user A is similar to user B and user C, When user A asks for recommendations, Then any items disliked by user B or by user C are not recommended.

6. Given user A is dissimilar to user B and user C, When user A asks for recommendations, Then any items liked by user B or by user C are not recommended.
7. Given user A is dissimilar to user B and user C, When user A asks for recommendations, Then any items disliked by user B or by user C are not recommended.
8. Given user A is similar to user B and dissimilar to user C, and user B and user C both like an item, When user A asks for recommendations, Then that item is not recommended.
9. Given user A is similar to user B and dissimilar to user C, and user B and user C both dislike an item, When user A asks for recommendations, Then that item is not recommended.
10. Given user A is more similar to user B than to user C, and user B likes an item but user C dislikes that same item, When User A asks for recommendations, Then that item is recommended.
11. Given user A is more similar to user B than to user C, and user B dislikes an item but user C likes that same item, When User A asks for recommendations, Then that item is not recommended.

These high-level requirements specify the direct user-facing interactions with the system. Each of these interactions will require a number of sub-decisions to produce the expected output. But since we are not designing the whole system upfront, we do not have to worry about those decisions just yet.

Expressing Requirements As Tests

We are building the core of the recommendation system, which would require an application interface - the main class with which the outside world interacts. This API class will house our high-level requirements. Let's call it *Engine*. When practicing TDD, we always start with a test, never the implementation. So we start with *EngineTest*, where each high-level requirement is written as an independent test.

```
public class EngineTest {

        @Test
        public void givenANamedUserWhenUserIsAddedThenUserGetsAddedSuccessfully()
        {
                //Given a user identified by name,
                //When the user is added for the first time,
                //Then user gets added successfully.
        }

        @Test
        public void givenTwoUsersWithSameNameWhenUsersAreAddedThenOnlyOneUserIsRecorded()
        {
                //Given two users identified by the same name,
                //When both users are added,
                //Then user is recorded only once.
        }

        @Test
        public void givenANamedItemWhenItemIsAddedThenItemGetsAddedSuccessfully()
        {
                //Given an item identified by name,
                //When item is added for the first time,
                //Then item gets added successfully.
        }

        @Test
        public void givenTwoItemsWithSameNameWhenItemsAreAddedThenOnlyOneItemIsRecorded()
        {
```

```
                //Given two items identified by the same name,
                //When both items are added,
                //Then item is recorded only once.
        }

        @Test
        public void givenNoUsersExistWhenUsersAreQueriedThenNoUsersAreReturned()
        {
                //Given no users exist in the system,
                //When system is queried for users,
                //Then no users are returned.
        }

        @Test
        public void givenUsersExistWhenUsersAreQueriedThenAllUsersAreReturned()
        {
                //Given one or more users exist in the system,
                //When system is queried for users,
                //Then all users in system are returned.
        }

        @Test
        public void givenNoItemsExistWhenItemsAreQueriedThenNoItemsAreReturned()
        {
                //Given no items exist in the system,
                //When system is queried for items,
                //Then no items are returned.
        }

        @Test
        public void givenItemsExistWhenItemsAreQueriedThenAllItemsAreReturned()
        {
                //Given one or more items exist in the system,
                //When system is queried for items,
                //Then all items in the system are returned.
        }

        @Test
        public void givenAnItemNotRatedByUserWhenUserLikesItemThenLikeIsRecordedForThatUserItemPair()
        {
                //Given an item not rated by a user,
                //When the user likes that item,
                //Then a like is recorded for that user-item pair.
        }

        @Test
        public void
givenAnItemNotRatedByUserWhenUserDislikesItemThenDislikeIsRecordedForThatUserItemPair()
        {
                //Given an item not rated by a user,
                //When the user dislikes that item,
                //Then a dislike is recorded for that user-item pair.
        }

        @Test
        public void givenItemLikeByUserWhenUserDislikesItemThenOnlyDislikeIsRecordedForThatUserItemPair()
        {
                //Given an item liked by a user,
                //When the user dislikes that item,
                //Then only a dislike is recorded for that user-item pair.
        }

        @Test
        public void givenItemDislikeByUserWhenUserLikesItemThenOnlyLikeIsRecordedForThatUserItemPair()
        {
                //Given an item disliked by a user,
                //When the user likes that item,
                //Then only a like is recorded for that user-item pair.
        }
```

```
        @Test
        public void
givenAUserWhoLikesOrDislikesAnItemWhenTheUserUnratesThatItemThenNoRatingExistsForThatUserItemPair()
        {
                //Given a user who has liked or disliked an item,
                //When the user unrates that item,
                //Then no rating record exists for that user-item pair.
        }

        @Test
        public void givenNoItemsExistWhenAnyUserAsksForRecommendationsThenNoRecommendationsAreGiven()
        {
                //Given no items exist in the system,
                //When a user asks for recommendations,
                //Then no recommendations are provided.
        }

        @Test
        public void
givenOnlyOneUserExistsWhenThatUserAsksForRecommendationsThenNoRecommendationsAreGiven()
        {
                //Given only one user exists in the system,
                //When that user asks for recommendations,
                //Then no recommendations are provided.
        }

        @Test
        public void
givenAUserWhoHasNotRatedAnyItemsWhenThatUserAsksForRecommendationsThenNoRecommendationsAreGiven()
        {
                //Given a user who has not rated any items,
                //When that user asks for recommendations,
                //Then no recommendations are provided.
        }

        @Test
        public void
givenAUserWhoHasRatedSomeItemsWhenThatUserAsksForRecommendationsThenItemsNotYetRatedByUserAndLikedByAllSim
ilarUsersAreReturned()
        {
                //Given user A is similar to user B and user C,
                //When user A asks for recommendations,
                //Then all items liked by user B and by user C
                //      and not yet rated by user A are recommended.
        }

        @Test
        public void
givenAUserWithTwoSimilarUsersWhenThatUserAsksForRecommendationThenRecommendationsExcludeItemsDislikedByEit
herUser()
        {
                //Given user A is similar to user B and user C,
                //When user A asks for recommendations,
                //Then any items disliked by user B or by user C are not recommended.
        }

        @Test
        public void
givenAUserWithTwoDissimilarUsersWhenThatUserAsksForRecommendationsThenRecommendationsExcludeItemsLikedByEi
therUser()
        {
                //Given user A is dissimilar to user B and user C,
                //When user A asks for recommendations,
                //Then any items liked by user B or by user C are not recommended.
        }

        @Test
```

```
        public void
givenAUserWithTwoDissimilarUsersWhenThatUserAsksForRecommendationsThenRecommendationsIncludeItemsDislikedB
yEitherUser()
        {
                //Given user A is dissimilar to user B and user C,
                //When user A asks for recommendations,
                //Then any items disliked by user B or by user C are recommended.
        }

        @Test
        public void
givenAUserWithOneSimilarAndOneDissimilarUserWhenThatUserAsksForRecommendationsThenRecommendationsExcludeIt
emsLikedByBothUsers()
        {
                //Given user A is similar to user B and dissimilar to user C,
                //      and user B and user C both like an item,
                //When user A asks for recommendations,
                //Then that item is not recommended.
        }

        @Test
        public void
givenAUserWithOneSimilarAndOneDissimilarUserWhenThatUserAsksForRecommendationsThenRecommendationsExcludeIt
emsDislikedByBothUsers()
        {
                //Given user A is similar to user B and dissimilar to user C,
                //      and user B and user C both dislike an item,
                //When user A asks for recommendations,
                //Then that item is not recommended.
        }

        @Test
        public void
givenAUserWithOneSimilarUserWhoLikesAnItemAndALessSimilarUserWhoDislikesTheItemWhenThatUserAsksForRecommen
dationsThenTheItemIsRecommended()
        {
                //Given user A is more similar to user B than to user C,
                //      and user B likes an item but user C dislikes that same item,
                //When User A asks for recommendations,
                //Then that item is recommended.
        }

        @Test
        public void
givenAUserWithOneSimilarUserWhoDislikesAnItemAndALessSimilarUserWhoLikesTheItemWhenThatUserAsksForRecommen
dationsThenTheItemIsNotRecommended()
        {
                //Given user A is more similar to user B than to user C,
                //      and user B dislikes an item but user C likes that same item,
                //When User A asks for recommendations,
                //Then that item is not recommended.
        }
}
```

The really long test names are sure to raise more than a few eyebrows, but when you read those names, it should be immediately clear what behaviour is being tested. There is no prescription for naming tests, but the main goal is to make the test intention clear. So you can try to make the name as succinct as possible without losing sight of this goal. Here are some basic guidelines:

1. The closer to natural language the name is, the easier it will be to understand its intent.
2. The name should indicate the starting state, the transition that changes that state, and the final state caused by the transition. This draws parallels with the Given/When/Then construct of the

requirements, making it easier to see which requirement is being tested. But you do not necessarily have to write the test names in this format.

With the requirements transferred into tests, we can refer to them directly inside the code. Let's now work through these requirements one at a time.

Working in a TDD Cycle

The requirements-encoded empty tests serve as a great ToDo list, and we can work our way through this list, one step at a time. Here are the basic steps we will follow to work through this list:

1. Pick the next test in the list which can get us to a green bar as quickly as possible.
2. Write test code describing the starting state, transition, and our expectation of the final state. We will write the code in the way we *expect* to exercise the system.
3. Write enough production code that will allow our tests to compile, even if the tests fail.
4. Write enough production code that will allow our tests to pass.
5. Refactor the production code to remove any duplication, and so that new tests written to test similar behaviour will not require any changes to production code.
6. Run all tests in the system to make sure you still get a green bar, indicating all tests pass.

Now let's make sense of this through practice. We'll start with `givenANamedUserWhenUserIsAddedThenUserGetsAddedSuccessfully`.

```
public class EngineTest {

	@Test
	public void givenANamedUserWhenUserIsAddedThenUserGetsAddedSuccessfully()
	{
		Engine engine = new Engine();
		User Alice = new User("Alice");

		engine.addUsers(Alice);

		assertThat(engine.getUsers(), contains(Alice));
	}

	...

}
```

To get this test to compile, we need to add new classes, *Engine* and *User*, with some basic code.

```
public class Engine {

	public void addUser(User userToAdd)
	{
	}

	public Set<User> getUsers()
	{
		return Collections.emptySet();
	}

}

public class User {
```

```
        private String name;

        public User(String name)
        {
                this.name = name;
        }

        @Override
        public String toString()
        {
                return name;
        }
}
```

The *toString()* method has been added to help with debugging; testing it is not very useful since it does not implement any meaningful logic.

Our test fails, so we'll have to do some work to make it pass.

```
public class Engine {

        private Set<User> users = new HashSet<>();

        public void addUser(User userToAdd)
        {
                users.add(userToAdd);
        }

        public Set<User> getUsers()
        {
                return Collections.unmodifiableSet(users);
        }
}
```

The test passes, and we get our first green bar! Now, as soon as we start adding users to a hashset, we know the world of users is unsteady until we override the *equals* and *hashcode* methods in *User* to match our expectations. And since, in the world of TDD, tests always precede code, let's add a new *UserTest* class which describes how you expect *User* to behave with respect to equality and hashes, and then write the appropriate implementation to make these tests compile and pass.

```
public class UserTest {

        @Test
        public void aUserIsEqualToHerself()
        {
                User Alice = new User("Alice");

                assertThat(Alice, is(equalTo(Alice)));
        }

        @Test
        public void twoUsersAreEqualIfTheyHaveSameName()
        {
                User Alice = new User("Alice");
                User anotherAlice = new User("Alice");

                assertThat(Alice, is(equalTo(anotherAlice)));
        }

        @Test
        public void twoUsersAreNotEqualIfTheyDoNotHaveSameName()
        {
                User Alice = new User("Alice");
```

```
                User Bob = new User("Bob");

                assertThat(Alice, is(not(equalTo(Bob))));
        }

        @Test
        public void hashRemainsSameIfUserDoesNotChange()
        {
                User Alice = new User("Alice");

                int firstHash = Alice.hashCode();
                int secondHash = Alice.hashCode();
                int thirdHash = Alice.hashCode();

                assertThat(firstHash, is(equalTo(secondHash)));
                assertThat(firstHash, is(equalTo(thirdHash)));
        }

        @Test
        public void hashOfTwoEqualUsersIsSame()
        {
                User firstUser = new User("Alice");
                User secondUser = new User("Alice");

                assertThat(firstUser, is(equalTo(secondUser)));
                assertThat(firstUser.hashCode(), is(equalTo(secondUser.hashCode())));
        }
}

public class User {

        @Override
        public boolean equals(Object toCompare)
        {
                if(!(toCompare instanceof User)) {
                        return false;
                }

                User otherUser = (User) toCompare;
                return otherUser.name.equals(this.name);
        }

        @Override
        public int hashCode()
        {
                return Objects.hash(name);
        }

        ...
}
```

On to the next requirement!

```
public class EngineTest {

        @Test
        public void givenTwoUsersWithSameNameWhenUsersAreAddedThenOnlyOneUserIsRecorded()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User anotherAlice = new User("Alice");

                engine.addUser(Alice);
                engine.addUser(anotherAlice);

                assertThat(engine.getUsers().size(), is(1));
        }
```

```
        ...

}
```

This test passes without any work. Great! But let's use this opportunity to make a small improvement. It can be cumbersome to add users one at a time, so let's change Engine to accept multiple users at a time. Start by changing the test.

```
public class EngineTest {

        @Test
        public void givenANamedUserWhenUserIsAddedThenUserGetsAddedSuccessfully()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");

                engine.addUsers(Alice);

                assertThat(engine.getUsers(), contains(Alice));
        }

        @Test
        public void givenTwoUsersWithSameNameWhenUsersAreAddedThenOnlyOneUserIsRecorded()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User anotherAlice = new User("Alice");

                engine.addUsers(Alice, anotherAlice);

                assertThat(engine.getUsers().size(), is(1));
        }

        ...

}

public class Engine {

        public void addUsers(User... usersToAdd)
        {
                users.addAll(Arrays.asList(usersToAdd));
        }

        ...
}
```

The next two tests, for adding items, are similar to the tests for adding users, so we should be able to fly through these.

```
public class EngineTest {

        @Test
        public void givenANamedItemWhenItemIsAddedThenItemGetsAddedSuccessfully()
        {
                Engine engine = new Engine();
                Item aerobics = new Item("Aerobics");

                engine.addItems(aerobics);

                assertThat(engine.getItems(), contains(aerobics));
        }

        @Test
```

```
        public void givenTwoItemsWithSameNameWhenItemsAreAddedThenOnlyOneItemIsRecorded()
        {
                Engine engine = new Engine();
                Item aerobics = new Item("Aerobics");
                Item moreAerobics = new Item("Aerobics");

                engine.addItems(aerobics, moreAerobics);

                assertThat(engine.getItems().size(), is(1));
        }

        ...

}

public class Engine {

        private Set<Item> items = new HashSet<>();

        public void addItems(Item... itemsToAdd)
        {
                items.addAll(Arrays.asList(itemsToAdd));
        }

        public Set<Item> getItems()
        {
                return Collections.unmodifiableSet(items);
        }

        ...
}

public class ItemTest {

        @Test
        public void anItemIsEqualToItself()
        {
                Item item = new Item("item");

                assertThat(item, is(equalTo(item)));
        }

        @Test
        public void twoItemsAreEqualIfTheyHaveSameName()
        {
                Item item1 = new Item("item");
                Item item2 = new Item("item");

                assertThat(item1, is(equalTo(item2)));
        }

        @Test
        public void twoItemsAreNotEqualIfTheyDoNotHaveSameID()
        {
                Item item1 = new Item("item1");
                Item item2 = new Item("item2");

                assertThat(item1, is(not(equalTo(item2))));
        }

        @Test
        public void hashRemainsSameIfItemDoesNotChange()
        {
                Item item = new Item("item");

                int firstHash = item.hashCode();
                int secondHash = item.hashCode();
                int thirdHash = item.hashCode();
```

```
                assertThat(firstHash, is(equalTo(secondHash)));
                assertThat(firstHash, is(equalTo(thirdHash)));
        }

        @Test
        public void hashOfTwoEqualItemsIsSame()
        {
                Item firstItem = new Item("some name");
                Item secondItem = new Item("some name");

                assertThat(firstItem, is(equalTo(secondItem)));
                assertThat(firstItem.hashCode(), is(equalTo(secondItem.hashCode())));
        }
}

public class Item {

        private String name;

        public Item(String name)
        {
                this.name = name;
        }

        @Override
        public boolean equals(Object toCompare)
        {
                if(!(toCompare instanceof Item)) {
                        return false;
                }

                Item otherItem = (Item)toCompare;
                return otherItem.name.equals(this.name);
        }

        @Override
        public int hashCode()
        {
                return Objects.hash(name);
        }

        @Override
        public String toString()
        {
                return name;
        }
}
```

The tests for viewing users and items can now be verified without any production code changes.

```
public class EngineTest {

        @Test
        public void givenNoUsersExistWhenUsersAreQueriedThenNoUsersAreReturned()
        {
                Engine engine = new Engine();

                assertThat(engine.getUsers(), is(empty()));
        }

        @Test
        public void givenUsersExistWhenUsersAreQueriedThenAllUsersAreReturned()
        {
                Engine engine = new Engine();
```

```
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                engine.addUsers(Alice, Bob);

                assertThat(engine.getUsers(), containsInAnyOrder(Alice, Bob));
        }

        @Test
        public void givenNoItemsExistWhenItemsAreQueriedThenNoItemsAreReturned()
        {
                Engine engine = new Engine();

                assertThat(engine.getItems(), is(empty()));
        }

        @Test
        public void givenItemsExistWhenItemsAreQueriedThenAllItemsAreReturned()
        {
                Engine engine = new Engine();

                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                engine.addItems(aerobics, basketball);

                assertThat(engine.getItems(), containsInAnyOrder(aerobics, basketball));
        }

        ...

}
```

Letting the User Experience Drive Implementation

To tackle the next set of tests, we have to introduce the idea of ratings. There are many different ways this concept can be implemented, but the habit of writing tests first can steer us in a direction that accompishes a noble goal - prioritize user experience over all else. How would a user of our API prefer expressing a positive or negative rating?

```
public class EngineTest {

        @Test
        public void givenAnItemNotRatedByUserWhenUserLikesItemThenLikeIsRecordedForThatUserItemPair()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                Item aerobics = new Item("Aerobics");

                engine.like(Alice, aerobics);

                assertThat(engine.getRating(Alice, aerobics), is(RATING.LIKE));
        }

        ...

}
```

There are a number of choices we have made when writing this test.

1. Likes and dislikes could have been expressed as `engine.addLike(Alice, aerobics)` or as `engine.addRating(Alice, "LIKE", aerobics)`. However, `engine.like(Alice, aerobics)` seems more natural.

2. To obtain a rating, we have chosen to use `engine.getRating(Alice, aerobics)` instead of something like `engine.getLikes(Alice)`. This reflects our theory that users want to know how an item is rated, rather than iterate over the various types of ratings and then determine whether or not an item was rated as that type.
3. The type of rating returned should be both intuitive and easy to interpret. Hence, we have chosen a constrained variable, `RATING.LIKE`, rather than something a little harder to parse, such as the string "LIKE".

These choices demonstrate that we are letting the user expectation drive the implementation, rather than the implementation drive the user experience. Instead of being consumed by the many implementation choices we could make, we simply write our test code in the way we envision users of the system interacting with the entities we create. The implementation then follows naturally. Note that, when writing tests for the API layer, users are the end-users of the entire system. When writing tests for lower layers, users are the software developers who will use the entities to build and grow the system.

To compile and have our test pass, we will need to make some changes to *Engine*.

```
public class Engine {

	public enum RATING {LIKE};
	private Map<User, Set<Item>> likes = new HashMap<>();

	public void like(User user, Item item)
	{
		if(!likes.containsKey(user)) {
			likes.put(user, new HashSet<>());
		}

		likes.get(user).add(item);
	}

	public RATING getRating(User user, Item item)
	{
		if(likes.containsKey(user) && likes.get(user).contains(item)) {
			return RATING.LIKE;
		} else {
			return null;
		}
	}

	...
}
```

Disliking an item is very similar.

```
public class EngineTest {

	@Test
	public void
givenAnItemNotRatedByUserWhenUserDislikesItemThenDislikeIsRecordedForThatUserItemPair()
	{
		Engine engine = new Engine();
		User Alice = new User("Alice");
		Item basketball = new Item("basketball");

		engine.dislike(Alice, basketball);
```

```
                assertThat(engine.getRating(Alice, basketball), is(RATING.DISLIKE));
        }
        ...
}

public class Engine {

        public enum RATING {LIKE, DISLIKE, UNSPECIFIED};
        private Map<User, Set<Item>> likes = new HashMap<>();
        private Map<User, Set<Item>> dislikes = new HashMap<>();

        public void dislike(User user, Item item)
        {
                if(!dislikes.containsKey(user)) {
                        dislikes.put(user, new HashSet<>());
                }

                dislikes.get(user).add(item);
        }

        public RATING getRating(User user, Item item)
        {
                if(likes.containsKey(user) && likes.get(user).contains(item)) {
                        return RATING.LIKE;
                } else if(dislikes.containsKey(user) &&
                        dislikes.get(user).contains(item)) {
                        return RATING.DISLIKE;
                } else {
                        return RATING.UNSPECIFIED;
                }
        }

}
```

Refactoring With Confidence

Our system now has the capability to register likes and dislikes, but *Engine* has started to look a little crowded. In addition to this, likes and dislikes seem to be a unified concept, a rating, so we would like to encapsulate those in a new *Ratings* object. This *Ratings* object would have, as its consumer, the Engine class, so we want to make sure Ratings behaves as Engine expects it to. Engine's expectations are, in turn, driven by the high-level requirements, which are now all nicely encoded into tests. When moving code from *Engine* to *Ratings*, we can rest assured that if the move is unsuccessful in any way, the Engine tests will alert us right away. Hence, we refactor with confidence.

```
public class Ratings {

        public enum RATING {LIKE, DISLIKE, UNSPECIFIED};

        private Map<User, Set<Item>> likes = new HashMap<>();
        private Map<User, Set<Item>> dislikes = new HashMap<>();

        public void addLike(User user, Item item)
        {
                if(!likes.containsKey(user)) {
                        likes.put(user, new HashSet<>());
                }

                likes.get(user).add(item);
        }

        public void addDislike(User user, Item item)
```

```
        {
                if(!dislikes.containsKey(user)) {
                        dislikes.put(user, new HashSet<>());
                }

                dislikes.get(user).add(item);
        }

        public RATING get(User user, Item item)
        {
                if(likes.containsKey(user) && likes.get(user).contains(item)) {
                        return RATING.LIKE;
                } else if(dislikes.containsKey(user) && dislikes.get(user).contains(item)) {
                        return RATING.DISLIKE;
                } else {
                        return RATING.UNSPECIFIED;
                }
        }

}

public class Engine {

        private Ratings ratings = new Ratings();

        public void like(User user, Item item)
        {
                ratings.addLike(user, item);
        }

        public void dislike(User user, Item item)
        {
                ratings.addDislike(user, item);
        }

        public RATING getRating(User user, Item item)
        {
                return ratings.get(user, item);
        }

        ...

}
```

When we run all our tests, the bar is still green, giving us confidence that the world is the same after our move, as it was before the move.

Test Independent Objects Independently

Right now, *Ratings* conforms only to the behaviour expected by *Engine*, so the tests in *EngineTests* are sufficient to verify its behaviour. But as *Engine* grows, it may become harder for us to draw parallels between the behaviour expected from *Engine* and the behaviour expected from *Ratings*. In time, other objects may start to depend on *Ratings*, and we would then have to ensure *Ratings* publishes, fulfills, and abides by an independent contract that other entities can depend on. So let's add *RatingsTest*, which highlights and verifies the current capabilities of the *Ratings* object.

```
public class RatingsTest {

        @Test
        public void givenAUserAndAnItemThenTheUserCanHaveNoRatingForThatItem() {}
```

```
        @Test
        public void givenAUserWhoHasNotRatedAnItemThenTheUserCanLikeThatItem() {}

        @Test
        public void givenAUserWhoHasNotRatedAnItemThenTheUserCanDislikeThatItem() {}

        @Test
        public void givenMultipleUsersRateTheSameItemThenTheirRatingsDoNotConflictWithEachOther() {}

}
```

Once again, the empty tests serve as a ToDo list while we think through the behaviour exhibited by the *Ratings* object. Note how we are not listing all the behaviour the *Ratings* object will ever be capable of. Instead, we are only describing its current capabilities, and we will add to these as more is expected from the *Ratings* object. It is natural that most of the tests for *Ratings* are very similar to, or the same as, the ratings tests in *EngineTest*. However, the requirements in Engine could change, and fulfilling those requirements may require the same behaviour from *Ratings* in addition to some behaviour from other objects. The independent tests for Ratings allow us to focus on the behaviour of one unit, and one unit only. If a ratings related test in *EngineTest* fails, but all tests in *RatingsTest* pass, then we know immediately where to focus our attention when trying to diagnose the problem.

Another very important point to note is that we are not dogmatically testing each method of the *Ratings* class; rather, we are testing independent scenarios the *Ratings* object will handle. While this may worry some of you who are now wondering if it is not important to test each code path, realize that every code path is either detecting a state or state transition, or it is producing an output. So if your tests list all the scenarios your code handles, all the code paths will get tested. And if you drive implementation with scenario-based tests, all your code will have been written to fulfill those scenarios. You can also use code coverage tools, such as Emma, to further put your mind at ease, or to help find dead code paths as your system evolves. When you see the clarity with which behaviour-driven tests allow you to see the bigger picture, the use cases your system solves, I hope you will realize there is no other way in which tests should be written.

We can now define the tests in *RatingsTest*. We should be able to verify these using the code we already have in place.

```
public class RatingsTest {

        @Test
        public void givenAUserAndAnItemThenTheUserCanHaveNoRatingForThatItem()
        {
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");

                assertThat(ratings.get(Alice, aerobics), is(RATING.UNSPECIFIED));
        }

        @Test
        public void givenAUserWhoHasNotRatedAnItemThenTheUserCanLikeThatItem()
        {
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
```

```
                Item aerobics = new Item("aerobics");

                ratings.addLike(Alice, aerobics);

                assertThat(ratings.get(Alice, aerobics), is(RATING.LIKE));
        }

        @Test
        public void givenAUserWhoHasNotRatedAnItemThenTheUserCanDislikeThatItem()
        {
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");

                ratings.addDislike(Alice, aerobics);

                assertThat(ratings.get(Alice, aerobics), is(RATING.DISLIKE));
        }

        @Test
        public void givenMultipleUsersRateTheSameItemThenTheirRatingsDoNotConflictWithEachOther()
        {
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                Item aerobics = new Item("aerobics");

                ratings.addLike(Alice, aerobics);
                ratings.addDislike(Bob, aerobics);

                assertThat(ratings.get(Alice, aerobics), is(RATING.LIKE));
                assertThat(ratings.get(Bob, aerobics), is(RATING.DISLIKE));
        }
}
```

Let's get back to the next test for *Engine*.

```
public class EngineTest {

        @Test
        public void givenItemLikedByUserWhenUserDislikesItemThenOnlyDislikeIsRecordedForThatUserItemPair()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                engine.like(Alice, aerobics);

                engine.dislike(Alice, aerobics);

                assertThat(engine.getRating(Alice, aerobics), is(RATING.DISLIKE));
        }

        ...

}
```

This test fails. Since the Engine code for rating is really just delegating the work to the *Ratings* object, we need to ensure *Ratings* agrees to provide this behaviour.

```
public class RatingsTest {

        @Test
        public void givenAUserHasLikedAnItemWhenTheUserDislikesThatItemThenThatItemIsDislikedByTheUser()
        {
                Ratings ratings = new Ratings();
```

```
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                ratings.addLike(Alice, aerobics);

                ratings.addDislike(Alice, aerobics);

                assertThat(ratings.get(Alice, aerobics), is(RATING.DISLIKE));
        }
        ...
}
```

This test also fails. Good! To have this test pass, we need to make sure we remove any likes for an item before it gets disliked.

```
public class Ratings {

        public void addDislike(User user, Item item)
        {
                if(likes.containsKey(user)) {
                        likes.get(user).remove(item);
                }

                if(!dislikes.containsKey(user)) {
                        dislikes.put(user, new HashSet<>());
                }

                dislikes.get(user).add(item);
        }

        ...
}
```

Now the Engine test will also pass. The next Engine test is very similar.

```
public class EngineTest {

        @Test
        public void givenItemDislikedByUserWhenUserLikesItemThenOnlyLikeIsRecordedForThatUserItemPair()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                engine.dislike(Alice, aerobics);

                engine.like(Alice, aerobics);

                assertThat(engine.getRating(Alice, aerobics), is(RATING.LIKE));
        }

        ...
}

public class RatingsTest {

        @Test
        public void givenAUserHasDislikedAnItemWhenTheUserLikesThatItemThenThatItemIsLikedByTheUser()
        {
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                ratings.addDislike(Alice, aerobics);

                ratings.addLike(Alice, aerobics);

                assertThat(ratings.get(Alice, aerobics), is(RATING.LIKE));
        }
```

```
        ...
}

public class Ratings {

        public void addLike(User user, Item item)
        {
                if(dislikes.containsKey(user)) {
                        dislikes.get(user).remove(item);
                }

                if(!likes.containsKey(user)) {
                        likes.put(user, new HashSet<>());
                }

                likes.get(user).add(item);
        }

        ...
}
```

The final ratings related test in *Engine* adds the ability to remove a rating. When we add this capability to *Ratings*, it also lends us the opportunity to remove some code duplication, so in addition to a new *remove* method, we will also add an *add* method, that adds the specified rating.

```
public class EngineTest {

        @Test
        public void givenItemDislikedByUserWhenUserLikesItemThenOnlyLikeIsRecordedForThatUserItemPair()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                engine.dislike(Alice, aerobics);

                engine.like(Alice, aerobics);

                assertThat(engine.getRating(Alice, aerobics), is(RATING.LIKE));
        }

        @Test
        public void
givenAUserWhoLikesOrDislikesAnItemWhenTheUserUnratesThatItemThenNoRatingExistsForThatUserItemPair()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                engine.like(Alice, aerobics);
                engine.dislike(Alice, basketball);

                engine.removeRating(Alice, aerobics);
                engine.removeRating(Alice, basketball);

                assertThat(engine.getRating(Alice, aerobics), is(RATING.UNSPECIFIED));
                assertThat(engine.getRating(Alice, basketball), is(RATING.UNSPECIFIED));
        }
        ...
}

public class Engine {

        public void like(User user, Item item)
        {
                ratings.add(user, item, RATING.LIKE);
```

```
        }

        public void dislike(User user, Item item)
        {
                ratings.add(user, item, RATING.DISLIKE);
        }

        public void removeRating(User user, Item item)
        {
                ratings.remove(user, item);
        }
        ...
}

public class RatingsTest {

        @Test
        public void givenAUserHasDislikedAnItemWhenTheUserLikesThatItemThenThatItemIsLikedByTheUser()
        {
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                ratings.add(Alice, aerobics, RATING.DISLIKE);

                ratings.add(Alice, aerobics, RATING.LIKE);

                assertThat(ratings.get(Alice, aerobics), is(RATING.LIKE));
        }

        @Test
        public void
givenAUserHasLikedAnItemWhenTheUserUnratesThatItemThenNoRatingExistsForThatUserItemPair()
        {
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                ratings.add(Alice, aerobics, RATING.LIKE);

                ratings.remove(Alice, aerobics);

                assertThat(ratings.get(Alice, aerobics), is(RATING.UNSPECIFIED));
        }

        @Test
        public void
givenAUserHasDislikedAnItemWhenTheUserUnratesThatItemThenNoRatingExistsForThatUserItemPair()
        {
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                ratings.add(Alice, aerobics, RATING.DISLIKE);

                ratings.remove(Alice, aerobics);

                assertThat(ratings.get(Alice, aerobics), is(RATING.UNSPECIFIED));
        }

        ...

}

public class Ratings {

        public void add(User user, Item item, RATING rating)
        {
                remove(user, item);

                if(rating.equals(RATING.LIKE)) {
```

```
                addLike(user, item);
            } else if(rating.equals(RATING.DISLIKE)) {
                addDislike(user, item);
            }
        }

        private void addLike(User user, Item item)
        {
            if(!likes.containsKey(user)) {
                likes.put(user, new HashSet<>());
            }

            likes.get(user).add(item);
        }

        private void addDislike(User user, Item item)
        {
            if(!dislikes.containsKey(user)) {
                dislikes.put(user, new HashSet<>());
            }

            dislikes.get(user).add(item);
        }

        public void remove(User user, Item item)
        {
            if(likes.containsKey(user)) {
                likes.get(user).remove(item);
            }

            if(dislikes.containsKey(user)) {
                dislikes.get(user).remove(item);
            }
        }
        ...

}
```

We're now ready to tackle the recommendations tests.

Let the Design Naturally Emerge

By now we're quite familiar with the TDD cycle and can independently pace ourselves to work in small increments. So I will start to move faster, unless there is something specific to point out.

The first three recommendation tests are trivial to fulfill, since they are not asking for any recommendations.

```
public class EngineTest {

    @Test
    public void givenNoItemsExistWhenAnyUserAsksForRecommendationsThenNoRecommendationsAreGiven()
    {
        Engine engine = new Engine();
        User Alice = new User("Alice");
        User Bob = new User("Bob");
        engine.addUsers(Alice, Bob);

        assertThat(engine.getRecommendations(Alice), is(empty()));
        assertThat(engine.getRecommendations(Bob),is(empty()));
    }
```

```
        @Test
        public void
givenOnlyOneUserExistsWhenThatUserAsksForRecommendationsThenNoRecommendationsAreGiven()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                engine.addUsers(Alice);
                engine.addItems(aerobics, basketball);

                assertThat(engine.getRecommendations(Alice), is(empty()));
        }

        @Test
        public void
givenAUserWhoHasNotRatedAnyItemsWhenThatUserAsksForRecommendationsThenNoRecommendationsAreGiven()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                engine.addUsers(Alice, Bob);
                engine.addItems(aerobics, basketball);

                engine.like(Bob, basketball);

                assertThat(engine.getRecommendations(Alice), is(empty()));
        }

        ...

}

public class Engine {

        public Set<Item> getRecommendations(User user)
        {
                return Collections.emptySet();
        }

        ...

}
```

It is only when we get to the fourth recommendation test that we need to do some real work.

```
public class EngineTest {

        @Test
        public void
givenAUserWhoHasRatedSomeItemsWhenThatUserAsksForRecommendationsThenItemsNotYetRatedByUserAndLikedByAllSim
ilarUsersAreReturned()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cory = new User("Cory");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                engine.addUsers(Alice, Bob, Cory);
                engine.addItems(aerobics, basketball, curling);

                engine.like(Alice, aerobics);
```

```
                engine.like(Bob, aerobics);
                engine.like(Bob, basketball);

                engine.dislike(Cory, aerobics);
                engine.like(Cory, curling);

                assertThat(engine.getRecommendations(Alice), contains(basketball));
        }

        ...

}
```

Now we really need to figure out how to get recommendations. Let's set ourselves up by creating a new Recommendations object which will be responsible for calculating the recommendations for a user.

```
public class Recommendations {

        public Set<Item> get(User user)
        {
                return Collections.emptySet();
        }

}

public class Engine {

        private Recommendations recommendations = new Recommendations();

        public Set<Item> getRecommendations(User user)
        {
                return recommendations.get(user);
        }

}
```

How do we expect this *Recommendations* object to behave, given how similar two users are and what they like or dislike? Here are all the tests I can think of for getting recommendations:

```
public class RecommendationsTest {

        @Test
        public void givenAUserWithNoRatedItemsThenTheUserGetsNoRecommendations() {}

        @Test
        public void givenAUserWithNoUnratedItemsThenTheUserGetsNoRecommendations() {}

        @Test
        public void givenAUserWithNoSimilarUsersOtherThanHerselfThenTheUserGetsNoRecommendations() {}

        @Test
        public void
givenUserAIsSimilarToUserBThenItemsLikedByUserBAndNotYetRatedByUserAAreRecommendedToUserA() {}

        @Test
        public void givenUserAIsSimilarToUserBThenItemsDisLikedByUserBAreNotRecommendedToUserA() {}

        @Test
        public void givenUserAIsDissmilarToUserBThenItemsLikedByUserBAreNotRecommendedToUserA() {}

        @Test
        public void givenUserAIsDissmilarToUserBThenItemsDislikedByUserBAreRecommendedToUserA() {}
```

```
        @Test
        public void
givenUserAIsMoreSimilarToUserBThanUserCAndAnItemIsLikedByBothUsersThenUserAIsRecommendedThatItem() {}

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanUserCAndAnItemIsDislikedByBothUsersThenUserAIsNotRecommendedThatItem()
{}

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanUserCAndAnItemIsLikedByUserBButDislikedByUserCThenAIsRecommendedThatItem
() {}

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanUserCAndAnItemIsDislikedByUserBButLikedByUserCThenAIsNotRecommendedThatI
tem() {}

        @Test
        public void
givenUserAIsDissimilarToBothUserBAndUserCThenUserAIsNotRecommendedAnyItemsLikedByUserBOrUserC() {}

        @Test
        public void
givenUserAIsDissimilarToBothUserBAndUserCThenUserAIsRecommendedAnyItemsLikedByUserBOrUserC() {}

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanDissimilarToUserCAndAnItemIsLikedByBothUsersThenUserAIsRecommendedThatIt
em() {}

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanDissimilarToUserCAndAnItemIsDislikedByBothUsersThenUserAIsNotRecommended
ThatItem() {}

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanDissimilarToUserCAndAnItemIsLikedByUserBButDislikedByUserCThenUserAIsRec
ommendedThatItem() {}

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanDissimilarToUserCAndAnItemIsDislikedByUserBButLikedByUserCThenUserAIsNot
RecommendedThatItem() {}
```

Let's start with the first test, which appears to be the simplest.

```
public class RecommendationsTest {
        @Test
        public void givenAUserWithNoRatedItemsThenTheUserGetsNoRecommendations()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                Ratings ratings = new Ratings();

                assertThat(recommendations.get(Alice), is(empty()));
        }

        ...

}
```

We need to let Recommendations know about ratings, even if they're empty. We also anticipate that calculating the recommendations would be an expensive operation, and that it does not necessarily need to be done every time we ask for recommendations. So let's separate out the two operations.

```
public class RecommendationsTest {

        @Test
        public void givenAUserWithNoRatedItemsThenTheUserGetsNoRecommendations()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                Ratings ratings = new Ratings();

                recommendations.update(Alice, ratings);

                assertThat(recommendations.get(Alice), is(empty()));
        }

        ...

}

public class Recommendations {

        private Map<User, Set<Item>> recommendations = new HashMap<>();

        public Set<Item> get(User user)
        {
                return Optional.ofNullable(recommendations.get(user))
                                .orElse(Collections.emptySet());
        }

        public void update(User user, Ratings ratings)
        {

        }
}
```

Our simple test will now pass. We know we haven't done any real work, but we don't need to yet. So let's move on to the next test.

```
public class RecommendationsTest {

        @Test
        public void givenAUserWithNoUnratedItemsThenTheUserGetsNoRecommendations()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");

                Ratings ratings = new Ratings();
                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);

                recommendations.update(Alice, ratings);

                assertThat(recommendations.get(Alice), is(empty()));
        }
        ...
}
```

This will also pass without more work. On to the next one! With the next test, things get interesting. We somehow need to indicate the similarity between users. Figuring out similarity is a separate responsibility from calculating recommendations, so let's have a new object, *Similarity*, take up that responsibility, and pass it on to *Recommendations*. Also, as in the case of recommendations, let's keep the calculation of similar users separate from retrieval of similar users.

```
public class RecommendationsTest {

        @Test
        public void givenAUserWithNoRatedItemsThenTheUserGetsNoRecommendations()
        {
                ...
                recommendations.update(Alice, new Similarity(), ratings);
                ...
        }

        @Test
        public void givenAUserWithNoUnratedItemsThenTheUserGetsNoRecommendations()
        {
                ...

                recommendations.update(Alice, new Similarity(), ratings);
                ...
        }

        @Test
        public void givenAUserWithNoSimilarUsersOtherThanHerselfThenTheUserGetsNoRecommendations()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");

                Ratings ratings = new Ratings();
                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);

                recommendations.update(Alice, new Similarity(), ratings);

                assertThat(recommendations.get(Alice), is(empty()));
        }

        ...

}

public class Similarity {

        private Map<User, Set<User>> similarUsers = new HashMap<>();

        public Set<User> getSimilarUsers(User user)
        {
                return Optional.ofNullable(similarUsers.get(user))
                                .orElse(Collections.emptySet());
        }

        public void update(Set<User> allUsers, Ratings ratings)
        {
        }

}
```

This gets us a green bar, but before we can go any further, we need to make sure we understand what makes two users similar to each other.

```
public class SimilarityTest {

	@Test
	public void givenAUserWhoHasNotRatedAnyItemsThenTheUserIsSimilarToHimself() {}

	@Test
	public void givenAUserWhoHasRatedSomeItemsThenTheUserIsSimilarToHimself() {}

	@Test
	public void givenNeitherUserANorUserBHasRatedAnyItemsThenUserAAndUserBAreNotSimilar() {}

	@Test
	public void
givenUserAAndUserBLikeAllTheSameItemsAndDislikeAllTheSameItemsThenUserAAndUserBAreSimilar() {}

	@Test
	public void
givenUserANeitherLikesAnyItemsUserBLikesNorDislikesAnyItemsUserBDislikesThenUserAAndUserBAreDissimilar()
{}

	@Test
	public void
givenUserAAndUserBHaveSameNumberOfLikesAndDislikesInCommonAsNotInCommonThenUserAAndUserBAreNotSimilar() {}

	@Test
	public void
givenUserAAndUserBHaveMoreLikesAndDislikesInCommonThanUncommonLikesAndDislikesThenUserAAndUserBAreSimilar(
) {}

	@Test
	public void
givenUserAAndUserBHaveLessLikesAndDislikesInCommonThanUncommonLikesAndDislikesThenUserAAndUserBAreNotSimil
ar() {}

}
```

By now, you should be able to see how objects like *Ratings*, *Recommendations* and *Similarity*, and the relationship these objects have with each other, have emerged naturally from the simple act of us fulfilling behaviour specified in tests. If we had tried to design the system upfront, we would most likely have overdesigned it.

Don't couple tests to implementation

To find the similarity between two users, we will calculate their similarity index using a variation of the Jaccard Index. For a detailed explanation of this, refer to Mahmud Ridwan's article on a simple collaborative filtering recommendation engine[3].

Given two users, U_1 and U_2, the similarity index between them is calculated as:

$S(U_1, U_2) = \frac{|L_1 \cap L_2| + |D_1 \cap D_2| - |L_1 \cap D_2| - |L_2 \cap D_1|}{|L_1 \cup L_2 \cup D_1 \cup D_2|}$, where L_1 and L_2 are the set of items liked by U_1 and U_2 respectively, and D_1 and D_2 are the set of items disliked by U_1 and U_2 respectively.

```
public class SimilarityTest {
```

```
	@Test
	public void givenAUserWhoHasNotRatedAnyItemsThenTheUserIsSimilarToHimself()
	{
		Similarity similarity = new Similarity();
		User Alice = new User("Alice");
		Set<User> allUsers = new HashSet<>(Arrays.asList(Alice));

		Ratings ratings = new Ratings();
		similarity.update(allUsers, ratings);

		assertThat(similarity.getSimilarUsers(Alice), contains(Alice));
	}

	@Test
	public void givenAUserWhoHasRatedSomeItemsThenTheUserIsSimilarToHimself()
	{
		Similarity similarity = new Similarity();
		User Alice = new User("Alice");
		Item aerobics = new Item("aerobics");
		Item basketball = new Item("basketball");
		Set<User> allUsers = new HashSet<>(Arrays.asList(Alice));

		Ratings ratings = new Ratings();
		ratings.add(Alice, aerobics, RATING.LIKE);
		ratings.add(Alice, basketball, RATING.DISLIKE);
		similarity.update(allUsers, ratings);

		assertThat(similarity.getSimilarUsers(Alice), contains(Alice));
	}

	@Test
	public void givenNeitherUserANorUserBHasRatedAnyItemsThenUserAAndUserBAreNotSimilar()
	{
		Similarity similarity = new Similarity();
		User Alice = new User("Alice");
		User Bob = new User("Bob");
		Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));

		Ratings ratings = new Ratings();
		similarity.update(allUsers, ratings);

		assertThat(similarity.getSimilarUsers(Alice), not(hasItem(Bob)));
		assertThat(similarity.getSimilarUsers(Bob), not(hasItem(Alice)));
	}

	@Test
	public void
givenUserAAndUserBLikeAllTheSameItemsAndDislikeAllTheSameItemsThenUserAAndUserBAreSimilar()
	{
		Similarity similarity = new Similarity();
		Ratings ratings = new Ratings();

		User Alice = new User("Alice");
		User Bob = new User("Bob");
		Item aerobics = new Item("aerobics");
		Item basketball = new Item("basketball");
		Item curling = new Item("curling");

		ratings.add(Alice, aerobics, RATING.LIKE);
		ratings.add(Alice, basketball, RATING.LIKE);
		ratings.add(Alice, curling, RATING.DISLIKE);

		ratings.add(Bob, aerobics, RATING.LIKE);
		ratings.add(Bob, basketball, RATING.LIKE);
		ratings.add(Bob, curling, RATING.DISLIKE);

		Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
		similarity.update(allUsers, ratings);
```

```
                assertThat(similarity.getSimilarUsers(Alice), hasItem(Bob));
                assertThat(similarity.getSimilarUsers(Bob), hasItem(Alice));
        }

        @Test
        public void
givenUserANeitherLikesAnyItemsUserBLikesNorDislikesAnyItemsUserBDislikesThenUserAAndUserBAreDissimilar()
        {
                Similarity similarity = new Similarity();
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.DISLIKE);
                ratings.add(Bob, basketball, RATING.LIKE);
                ratings.add(Bob, curling, RATING.DISLIKE);

                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
                similarity.update(allUsers, ratings);

                assertThat(similarity.getSimilarUsers(Alice), not(hasItem(Bob)));
                assertThat(similarity.getSimilarUsers(Bob), not(hasItem(Alice)));
        }

        @Test
        public void
givenUserAAndUserBHaveSameNumberOfLikesAndDislikesInCommonAsNotInCommonThenUserAAndUserBAreNotSimilar()
        {
                Similarity similarity = new Similarity();
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
                User Bob = new User("Bobr");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                Item dodgeball = new Item("dodgeball");

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.DISLIKE);
                ratings.add(Alice, dodgeball, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.DISLIKE);
                ratings.add(Bob, curling, RATING.LIKE);
                ratings.add(Bob, dodgeball, RATING.DISLIKE);

                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
                similarity.update(allUsers, ratings);

                assertThat(similarity.getSimilarUsers(Alice), not(hasItem(Bob)));
                assertThat(similarity.getSimilarUsers(Bob), not(hasItem(Alice)));
        }

        @Test
        public void
givenUserAAndUserBHaveMoreLikesAndDislikesInCommonThanUncommonLikesAndDislikesThenUserAAndUserBAreSimilar(
)
        {
                Similarity similarity = new Similarity();
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
```

```
                User Bob = new User("Bob");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.DISLIKE);
                ratings.add(Bob, curling, RATING.DISLIKE);

                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
                similarity.update(allUsers, ratings);

                assertThat(similarity.getSimilarUsers(Alice), hasItem(Bob));
                assertThat(similarity.getSimilarUsers(Bob), hasItem(Alice));
        }

        @Test
        public void
givenUserAAndUserBHaveLessLikesAndDislikesInCommonThanUncommonLikesAndDislikesThenUserAAndUserBAreNotSimil
ar()
        {
                Similarity similarity = new Similarity();
                Ratings ratings = new Ratings();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.LIKE);
                ratings.add(Bob, curling, RATING.DISLIKE);

                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
                similarity.update(allUsers, ratings);

                assertThat(similarity.getSimilarUsers(Alice), not(hasItem(Bob)));
                assertThat(similarity.getSimilarUsers(Bob), not(hasItem(Alice)));
        }

}

public class Similarity {

        private static final double SIMILARITY_THRESHOLD = 0.0;
        private Map<User, Set<User>> similarUsers = new HashMap<>();

        public Set<User> getSimilarUsers(User user)
        {
                return Optional.ofNullable(similarUsers.get(user))
                                .orElse(Collections.emptySet());
        }

        public void update(Set<User> allUsers, Ratings ratings)
        {
                for(User user : allUsers) {
                        Set<User> similars = getSimilarUsers(user, allUsers, ratings);
                        similarUsers.put(user, similars);
                }
        }
```

```
	private Set<User> getSimilarUsers(User user, Set<User> allUsers, Ratings ratings)
	{
		Set<User> similars = new HashSet<>();
		for(User otherUser: allUsers) {
			double similarityIndex = similarityIndexOf(user, otherUser, ratings);
			if(similarityIndex > SIMILARITY_THRESHOLD) {
				similars.add(otherUser);
			}
		}
		return similars;
	}

	private double similarityIndexOf(User firstUser, User secondUser, Ratings ratings)
	{
		if(firstUser.equals(secondUser)) {
			return 1.0;
		} else if(ratings.itemsRatedByUser(firstUser).isEmpty() &&
				ratings.itemsRatedByUser(secondUser).isEmpty()) {
			return 0.0;
		} else {
			return calculateSimilarityIndexOf(firstUser, secondUser, ratings);

		}
	}

	@SuppressWarnings("unchecked")
	private double calculateSimilarityIndexOf(User firstUser, User secondUser, Ratings ratings)
	{
		Set<Item> firstUserLikes = ratings.likes(firstUser);
		Set<Item> secondUserLikes = ratings.likes(secondUser);
		Set<Item> firstUserDislikes = ratings.dislikes(firstUser);
		Set<Item> secondUserDislikes = ratings.dislikes(secondUser);

		double intersection = intersection(firstUserLikes, secondUserLikes).size()
				+ intersection(firstUserDislikes, secondUserDislikes).size()
				- intersection(firstUserLikes, secondUserDislikes).size()
				- intersection(secondUserLikes, firstUserDislikes).size();
		double union = union(firstUserLikes, secondUserLikes, firstUserDislikes,
secondUserDislikes).size();
		double similarityIndex = intersection/union;

		return similarityIndex;
	}

	private Set<Item> intersection(Set<Item> firstSet, Set<Item> secondSet)
	{
		return firstSet.stream()
				.filter(secondSet::contains)
				.collect(Collectors.toSet());
	}

	@SuppressWarnings("unchecked")
	private Set<Item> union(Set<Item>... sets)
	{
		return Stream.of(sets)
				.flatMap(Collection::stream)
				.collect(Collectors.toSet());
	}

}
```

While implementing Similarity, I have had to add some supporting methods to *Ratings*. *Ratings* can now report on what items are liked or disliked by a particular user, which items are rated by a given user, which items are not rated by a given user, and which users have rated a particular item. These helper methods do not represent new scenarios, so rather than add new tests to specifically test these

methods, I have added new verification statements to existing tests. This adds to the credibility and clarity of those scenarios, while ensuring these methods are working as expected. Here is one example of how I have modified the existing tests:

```
public class RatingsTest {

	@Test
	public void givenAUserWhoHasNotRatedAnItemThenTheUserCanLikeThatItem()
	{
		Ratings ratings = new Ratings();
		User Alice = new User("Alice");
		Item aerobics = new Item("aerobics");

		ratings.add(Alice, aerobics, RATING.LIKE);

		assertThat(ratings.get(Alice, aerobics), is(RATING.LIKE));
		assertThat(ratings.likes(Alice), hasItem(aerobics));
		assertThat(ratings.usersWhoRatedItem(aerobics, RATING.LIKE), contains(Alice));
	}

	...

}
```

Now that we know how the similarity between users is determined, we can move our attention back to *Recommendations*. To find recommendations for a user, we will calculate the probability of a user liking an item. Once again, for a detailed explanation of this, refer to Mahmud Ridwan's article on a simple collaborative filtering recommendation engine[3].

Given a user U, and an item I, the probability that U will like I is :

$P(U,I) = \frac{Z_L - Z_D}{|I_L| + |I_D|}$, where Z_L and Z_D are, respectively, the similarities of U with other users who have liked I and other users who have disliked I, and I_L and I_D are, respectively, the total number of users who have liked I and the total number of users who have disliked I.

```
public class RecommendationsTest {

	@Test
	public void givenAUserWithNoRatedItemsThenTheUserGetsNoRecommendations()
	{
		Recommendations recommendations = new Recommendations();
		User Alice = new User("Alice");
		User Bob = new User("Bob");
		Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
		Set<Item> allItems = Collections.singleton(new Item("aerobics"));

		Ratings ratings = new Ratings();
		Similarity similarity = new Similarity();
		similarity.update(allUsers, ratings);

		recommendations.update(Alice, allItems, similarity, ratings);

		assertThat(ratings.itemsRatedByUser(Alice), is(empty()));
		assertThat(recommendations.get(Alice), is(empty()));
	}

	@Test
	public void givenAUserWithNoUnratedItemsThenTheUserGetsNoRecommendations()
	{
		Recommendations recommendations = new Recommendations();
```

```
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
                Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball));

                Ratings ratings = new Ratings();
                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(ratings.itemsNotRatedByUser(Alice, allItems), is(empty()));
                assertThat(recommendations.get(Alice), is(empty()));
        }

        @Test
        public void givenAUserWithNoSimilarUsersOtherThanHerselfThenTheUserGetsNoRecommendations()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
                Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball));

                Ratings ratings = new Ratings();
                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.DISLIKE);
                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), contains(Alice));
                assertThat(recommendations.get(Alice), is(empty()));
        }

        @Test
        public void
givenUserAIsSimilarToUserBThenItemsLikedByUserBAndNotYetRatedByUserAAreRecommendedToUserA()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
                Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball));

                Ratings ratings = new Ratings();
                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.LIKE);
                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), hasItem(Bob));
                assertThat(recommendations.get(Alice), contains(basketball));
        }

        @Test
        public void givenUserAIsSimilarToUserBThenItemsDisLikedByUserBAreNotRecommendedToUserA()
```

```
{
        Recommendations recommendations = new Recommendations();
        User Alice = new User("Alice");
        User Bob = new User("Bob");
        Item aerobics = new Item("aerobics");
        Item basketball = new Item("basketball");
        Item curling = new Item("curling");
        Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
        Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball, curling));

        Ratings ratings = new Ratings();
        ratings.add(Alice, aerobics, RATING.LIKE);
        ratings.add(Bob, aerobics, RATING.LIKE);
        ratings.add(Bob, basketball, RATING.DISLIKE);
        ratings.add(Bob, curling, RATING.LIKE);
        Similarity similarity = new Similarity();
        similarity.update(allUsers, ratings);

        recommendations.update(Alice, allItems, similarity, ratings);

        assertThat(similarity.getSimilarUsers(Alice), hasItem(Bob));
        assertThat(recommendations.get(Alice), not(hasItem(basketball)));
}

@Test
public void givenUserAIsDissmilarToUserBThenItemsLikedByUserBAreNotRecommendedToUserA()
{
        Recommendations recommendations = new Recommendations();
        User Alice = new User("Alice");
        User Bob = new User("Bob");
        Item aerobics = new Item("aerobics");
        Item basketball = new Item("basketball");
        Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
        Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball));

        Ratings ratings = new Ratings();
        ratings.add(Alice, aerobics, RATING.LIKE);
        ratings.add(Bob, aerobics, RATING.DISLIKE);
        ratings.add(Bob, basketball, RATING.LIKE);
        Similarity similarity = new Similarity();
        similarity.update(allUsers, ratings);

        recommendations.update(Alice, allItems, similarity, ratings);

        assertThat(similarity.getSimilarUsers(Alice), not(hasItem(Bob)));
        assertThat(recommendations.get(Alice), not(hasItem((basketball))));
}

@Test
public void givenUserAIsDissmilarToUserBThenItemsDislikedByUserBAreRecommendedToUserA()
{
        Recommendations recommendations = new Recommendations();
        User Alice = new User("Alice");
        User Bob = new User("Bob");
        Item aerobics = new Item("aerobics");
        Item basketball = new Item("basketball");
        Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob));
        Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball));

        Ratings ratings = new Ratings();
        ratings.add(Alice, aerobics, RATING.LIKE);
        ratings.add(Bob, aerobics, RATING.DISLIKE);
        ratings.add(Bob, basketball, RATING.DISLIKE);
        Similarity similarity = new Similarity();
        similarity.update(allUsers, ratings);

        recommendations.update(Alice, allItems, similarity, ratings);

        assertThat(similarity.getSimilarUsers(Alice), not(hasItem(Bob)));
```

```
                assertThat(recommendations.get(Alice), hasItem((basketball)));
        }

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanUserCAndAnItemIsLikedByBothUsersThenUserAIsRecommendedThatItem()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cam = new User("Cam");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                Item dodgeball = new Item("dodgeball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob, Cam));
                Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball, curling,
dodgeball));

                Ratings ratings = new Ratings();

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.DISLIKE);
                ratings.add(Bob, curling, RATING.LIKE);
                ratings.add(Bob, dodgeball, RATING.LIKE);

                ratings.add(Cam, aerobics, RATING.LIKE);
                ratings.add(Cam, basketball, RATING.DISLIKE);
                ratings.add(Cam, curling, RATING.DISLIKE);
                ratings.add(Cam, dodgeball, RATING.LIKE);

                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), hasItems(Bob, Cam));
                assertThat(recommendations.get(Alice), hasItem((dodgeball)));
        }

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanUserCAndAnItemIsDislikedByBothUsersThenUserAIsNotRecommendedThatItem()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cam = new User("Cam");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                Item dodgeball = new Item("dodgeball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob, Cam));
                Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball, curling,
dodgeball));

                Ratings ratings = new Ratings();

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.DISLIKE);
                ratings.add(Bob, curling, RATING.LIKE);
```

```
                ratings.add(Bob, dodgeball, RATING.DISLIKE);

                ratings.add(Cam, aerobics, RATING.LIKE);
                ratings.add(Cam, basketball, RATING.DISLIKE);
                ratings.add(Cam, curling, RATING.DISLIKE);
                ratings.add(Cam, dodgeball, RATING.DISLIKE);

                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), hasItems(Bob, Cam));
                assertThat(recommendations.get(Alice), not(hasItem((dodgeball))));
        }

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanUserCAndAnItemIsLikedByUserBButDislikedByUserCThenAIsRecommendedThatItem
()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cam = new User("Cam");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                Item dodgeball = new Item("dodgeball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob, Cam));
                Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball, curling,
dodgeball));

                Ratings ratings = new Ratings();

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.DISLIKE);
                ratings.add(Bob, curling, RATING.LIKE);
                ratings.add(Bob, dodgeball, RATING.LIKE);

                ratings.add(Cam, aerobics, RATING.LIKE);
                ratings.add(Cam, basketball, RATING.DISLIKE);
                ratings.add(Cam, curling, RATING.DISLIKE);
                ratings.add(Cam, dodgeball, RATING.DISLIKE);

                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), hasItems(Bob, Cam));
                assertThat(recommendations.get(Alice), hasItem((dodgeball)));
        }

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanUserCAndAnItemIsDislikedByUserBButLikedByUserCThenAIsNotRecommendedThatI
tem()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cam = new User("Cam");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
```

```
                Item curling = new Item("curling");
                Item dodgeball = new Item("dodgeball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob, Cam));
                Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball, curling,
dodgeball));

                Ratings ratings = new Ratings();

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.DISLIKE);
                ratings.add(Bob, curling, RATING.LIKE);
                ratings.add(Bob, dodgeball, RATING.DISLIKE);

                ratings.add(Cam, aerobics, RATING.LIKE);
                ratings.add(Cam, basketball, RATING.DISLIKE);
                ratings.add(Cam, curling, RATING.DISLIKE);
                ratings.add(Cam, dodgeball, RATING.LIKE);

                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), hasItems(Bob, Cam));
                assertThat(recommendations.get(Alice), not(hasItem((dodgeball))));
        }

        @Test
        public void
givenUserAIsDissimilarToBothUserBAndUserCThenUserAIsNotRecommendedAnyItemsLikedByUserBOrUserC()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cam = new User("Cam");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                Item dodgeball = new Item("dodgeball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob, Cam));
                Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball, curling,
dodgeball));

                Ratings ratings = new Ratings();

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.DISLIKE);
                ratings.add(Bob, basketball, RATING.LIKE);
                ratings.add(Bob, curling, RATING.DISLIKE);
                ratings.add(Bob, dodgeball, RATING.LIKE);

                ratings.add(Cam, aerobics, RATING.DISLIKE);
                ratings.add(Cam, basketball, RATING.LIKE);
                ratings.add(Cam, curling, RATING.DISLIKE);

                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), not(hasItems(Bob, Cam)));
                assertThat(recommendations.get(Alice), not(hasItem((dodgeball))));
```

```
	}

	@Test
	public void
givenUserAIsDissimilarToBothUserBAndUserCThenUserAIsRecommendedAnyItemsLikedByUserBOrUserC()
	{
		Recommendations recommendations = new Recommendations();
		User Alice = new User("Alice");
		User Bob = new User("Bob");
		User Cam = new User("Cam");
		Item aerobics = new Item("aerobics");
		Item basketball = new Item("basketball");
		Item curling = new Item("curling");
		Item dodgeball = new Item("dodgeball");
		Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob, Cam));
		Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball, curling,
dodgeball));

		Ratings ratings = new Ratings();

		ratings.add(Alice, aerobics, RATING.LIKE);
		ratings.add(Alice, basketball, RATING.DISLIKE);
		ratings.add(Alice, curling, RATING.LIKE);

		ratings.add(Bob, aerobics, RATING.DISLIKE);
		ratings.add(Bob, basketball, RATING.LIKE);
		ratings.add(Bob, curling, RATING.DISLIKE);
		ratings.add(Bob, dodgeball, RATING.DISLIKE);

		ratings.add(Cam, aerobics, RATING.DISLIKE);
		ratings.add(Cam, basketball, RATING.LIKE);
		ratings.add(Cam, curling, RATING.DISLIKE);

		Similarity similarity = new Similarity();
		similarity.update(allUsers, ratings);

		recommendations.update(Alice, allItems, similarity, ratings);

		assertThat(similarity.getSimilarUsers(Alice), not(hasItems(Bob, Cam)));
		assertThat(recommendations.get(Alice), hasItem((dodgeball)));
	}

	@Test
	public void
givenUserAIsMoreSimilarToUserBThanDissimilarToUserCAndAnItemIsLikedByBothUsersThenUserAIsRecommendedThatIt
em()
	{
		Recommendations recommendations = new Recommendations();
		User Alice = new User("Alice");
		User Bob = new User("Bob");
		User Cam = new User("Cam");
		Item aerobics = new Item("aerobics");
		Item basketball = new Item("basketball");
		Item curling = new Item("curling");
		Item dodgeball = new Item("dodgeball");
		Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob, Cam));
		Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball, curling,
dodgeball));

		Ratings ratings = new Ratings();

		ratings.add(Alice, aerobics, RATING.LIKE);
		ratings.add(Alice, basketball, RATING.DISLIKE);
		ratings.add(Alice, curling, RATING.LIKE);

		ratings.add(Bob, aerobics, RATING.LIKE);
		ratings.add(Bob, basketball, RATING.DISLIKE);
		ratings.add(Bob, curling, RATING.LIKE);
		ratings.add(Bob, dodgeball, RATING.LIKE);
```

```
                ratings.add(Cam, aerobics, RATING.LIKE);
                ratings.add(Cam, basketball, RATING.LIKE);
                ratings.add(Cam, curling, RATING.DISLIKE);
                ratings.add(Cam, dodgeball, RATING.LIKE);

                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), not(hasItems(Bob, Cam)));
                assertThat(recommendations.get(Alice), hasItem((dodgeball)));
        }

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanDissimilarToUserCAndAnItemIsDislikedByBothUsersThenUserAIsNotRecommended
ThatItem()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cam = new User("Cam");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                Item dodgeball = new Item("dodgeball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob, Cam));
                Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball, curling,
dodgeball));

                Ratings ratings = new Ratings();

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.DISLIKE);
                ratings.add(Bob, curling, RATING.LIKE);
                ratings.add(Bob, dodgeball, RATING.DISLIKE);

                ratings.add(Cam, aerobics, RATING.LIKE);
                ratings.add(Cam, basketball, RATING.LIKE);
                ratings.add(Cam, curling, RATING.DISLIKE);
                ratings.add(Cam, dodgeball, RATING.DISLIKE);

                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), not(hasItems(Bob, Cam)));
                assertThat(recommendations.get(Alice), not(hasItem((dodgeball))));
        }

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanDissimilarToUserCAndAnItemIsLikedByUserBButDislikedByUserCThenUserAIsRec
ommendedThatItem()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cam = new User("Cam");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
```

```
                Item dodgeball = new Item("dodgeball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob, Cam));
                Set<Item> allItems = new HashSet<>(Arrays.asList(aerobics, basketball, curling,
dodgeball));

                Ratings ratings = new Ratings();

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.DISLIKE);
                ratings.add(Bob, curling, RATING.LIKE);
                ratings.add(Bob, dodgeball, RATING.LIKE);

                ratings.add(Cam, aerobics, RATING.LIKE);
                ratings.add(Cam, basketball, RATING.LIKE);
                ratings.add(Cam, curling, RATING.DISLIKE);
                ratings.add(Cam, dodgeball, RATING.DISLIKE);

                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), not(hasItems(Bob, Cam)));
                assertThat(recommendations.get(Alice), hasItem((dodgeball)));
        }

        @Test
        public void
givenUserAIsMoreSimilarToUserBThanDissimilarToUserCAndAnItemIsDislikedByUserBButLikedByUserCThenUserAIsNot
RecommendedThatItem()
        {
                Recommendations recommendations = new Recommendations();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cam = new User("Cam");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                Item dodgeball = new Item("dodgeball");
                Set<User> allUsers = new HashSet<>(Arrays.asList(Alice, Bob, Cam));
                Set<Item> allItems = new HashSet<>(Arrays.asList(
                        aerobics, basketball, curling, dodgeball));

                Ratings ratings = new Ratings();

                ratings.add(Alice, aerobics, RATING.LIKE);
                ratings.add(Alice, basketball, RATING.DISLIKE);
                ratings.add(Alice, curling, RATING.LIKE);

                ratings.add(Bob, aerobics, RATING.LIKE);
                ratings.add(Bob, basketball, RATING.DISLIKE);
                ratings.add(Bob, curling, RATING.LIKE);
                ratings.add(Bob, dodgeball, RATING.DISLIKE);

                ratings.add(Cam, aerobics, RATING.LIKE);
                ratings.add(Cam, basketball, RATING.LIKE);
                ratings.add(Cam, curling, RATING.DISLIKE);
                ratings.add(Cam, dodgeball, RATING.LIKE);

                Similarity similarity = new Similarity();
                similarity.update(allUsers, ratings);

                recommendations.update(Alice, allItems, similarity, ratings);

                assertThat(similarity.getSimilarUsers(Alice), not(hasItems(Bob, Cam)));
```

```
                assertThat(recommendations.get(Alice), not(hasItem((dodgeball))));
        }

}

public class Recommendations {

        private static final double LIKING_PROBABILITY_THRESHOLD = 0.0;
        private Map<User, Set<Item>> suggestions = new HashMap<>();

        public Set<Item> get(User user)
        {
                return Optional.ofNullable(suggestions.get(user))
                                .orElse(Collections.emptySet());
        }

        public void update(User user, Set<Item> allItems,
                        Similarity similarity, Ratings ratings)
        {
                Set<Item> newSuggestions = new HashSet<>();

                for(Item unratedItem : ratings.itemsNotRatedByUser(user, allItems)) {
                        Set<User> usersWhoLikedThis = ratings.usersWhoRatedItem(
                                                        unratedItem, RATING.LIKE);
                        Set<User> usersWhoDislikedThis = ratings.usersWhoRatedItem(
                                                        unratedItem, RATING.DISLIKE);
                        double similarityWithUsersWhoLikedThis = similarity.similarityIndexOf(
                                                        user, usersWhoLikedThis, ratings);
                        double similarityWithUsersWhoDislikedThis = similarity.similarityIndexOf(
                                                        user, usersWhoDislikedThis, ratings);

                        double probabilityOfLikingThis =
                                (similarityWithUsersWhoLikedThis - similarityWithUsersWhoDislikedThis)
                                        /(usersWhoLikedThis.size() + usersWhoDislikedThis.size());
                        if(probabilityOfLikingThis > LIKING_PROBABILITY_THRESHOLD) {
                                newSuggestions.add(unratedItem);
                        }
                }

                suggestions.put(user, newSuggestions);
        }
}
```

Consider, in particular, the tests written for *Similarity* and *Recommendations*. Notice how the way we have written the tests has no bearing on what our implementation looks like. For example, the *Similarity* tests have no notion of a similarity index, and we can change the implementation at any time to use some other method for finding the similarity between two users. Similarly, the *Recommendations* tests are unaware that a probability calculation is done and evaluated against a threshold to find similar users. When writing the tests, we think about what behaviour a user should expect. When implementing the code, our concern is only to deliver that behaviour promised by the tests. Working in this way prevents the tests from becoming a maintenance burden. Our tests break only when we stop delivering on our promise, not when we decide to change the guts of the implementation.

And finally, with all the pieces of the puzzle in place, we are ready to complete the *Engine* tests.

```
public class Engine {

        private Similarity similarity = new Similarity();
        private Recommendations recommendations = new Recommendations();

        public Set<Item> getRecommendations(User user)
```

```
        {
                similarity.update(users, ratings);
                recommendations.update(user, items, similarity, ratings);

                return recommendations.get(user);
        }
        ...

}

public class EngineTest {

        @Test
        public void givenANamedUserWhenUserIsAddedThenUserGetsAddedSuccessfully()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");

                engine.addUsers(Alice);

                assertThat(engine.getUsers(), contains(Alice));
        }

        @Test
        public void givenTwoUsersWithSameNameWhenUsersAreAddedThenOnlyOneUserIsRecorded()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User anotherAlice = new User("Alice");

                engine.addUsers(Alice, anotherAlice);

                assertThat(engine.getUsers().size(), is(1));
        }

        @Test
        public void givenANamedItemWhenItemIsAddedThenItemGetsAddedSuccessfully()
        {
                Engine engine = new Engine();
                Item aerobics = new Item("Aerobics");

                engine.addItems(aerobics);

                assertThat(engine.getItems(), contains(aerobics));
        }

        @Test
        public void givenTwoItemsWithSameNameWhenItemsAreAddedThenOnlyOneItemIsRecorded()
        {
                Engine engine = new Engine();
                Item aerobics = new Item("Aerobics");
                Item moreAerobics = new Item("Aerobics");

                engine.addItems(aerobics, moreAerobics);

                assertThat(engine.getItems().size(), is(1));
        }

        @Test
        public void givenNoUsersExistWhenUsersAreQueriedThenNoUsersAreReturned()
        {
                Engine engine = new Engine();

                assertThat(engine.getUsers(), is(empty()));
        }

        @Test
        public void givenUsersExistWhenUsersAreQueriedThenAllUsersAreReturned()
```

```
        {
                Engine engine = new Engine();

                User Alice = new User("Alice");
                User Bob = new User("Bob");
                engine.addUsers(Alice, Bob);

                assertThat(engine.getUsers(), containsInAnyOrder(Alice, Bob));
        }

        @Test
        public void givenNoItemsExistWhenItemsAreQueriedThenNoItemsAreReturned()
        {
                Engine engine = new Engine();

                assertThat(engine.getItems(), is(empty()));
        }

        @Test
        public void givenItemsExistWhenItemsAreQueriedThenAllItemsAreReturned()
        {
                Engine engine = new Engine();

                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                engine.addItems(aerobics, basketball);

                assertThat(engine.getItems(), containsInAnyOrder(aerobics, basketball));
        }

        @Test
        public void givenAnItemNotRatedByUserWhenUserLikesItemThenLikeIsRecordedForThatUserItemPair()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");

                engine.like(Alice, aerobics);

                assertThat(engine.getRating(Alice, aerobics), is(RATING.LIKE));
        }

        @Test
        public void
givenAnItemNotRatedByUserWhenUserDislikesItemThenDislikeIsRecordedForThatUserItemPair()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                Item basketball = new Item("basketball");

                engine.dislike(Alice, basketball);

                assertThat(engine.getRating(Alice, basketball), is(RATING.DISLIKE));
        }

        @Test
        public void givenItemLikedByUserWhenUserDislikesItemThenOnlyDislikeIsRecordedForThatUserItemPair()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                Item aerobics = new Item("aerobics");
                engine.like(Alice, aerobics);

                engine.dislike(Alice, aerobics);

                assertThat(engine.getRating(Alice, aerobics), is(RATING.DISLIKE));
        }

        @Test
```

```
	public void givenItemDislikedByUserWhenUserLikesItemThenOnlyLikeIsRecordedForThatUserItemPair()
	{
		Engine engine = new Engine();
		User Alice = new User("Alice");
		Item aerobics = new Item("aerobics");
		engine.dislike(Alice, aerobics);

		engine.like(Alice, aerobics);

		assertThat(engine.getRating(Alice, aerobics), is(RATING.LIKE));
	}

	@Test
	public void
givenAUserWhoLikesOrDislikesAnItemWhenTheUserUnratesThatItemThenNoRatingExistsForThatUserItemPair()
	{
		Engine engine = new Engine();
		User Alice = new User("Alice");
		Item aerobics = new Item("aerobics");
		Item basketball = new Item("basketball");
		engine.like(Alice, aerobics);
		engine.dislike(Alice, basketball);

		engine.removeRating(Alice, aerobics);
		engine.removeRating(Alice, basketball);

		assertThat(engine.getRating(Alice, aerobics), is(RATING.UNSPECIFIED));
		assertThat(engine.getRating(Alice, basketball), is(RATING.UNSPECIFIED));
	}

	@Test
	public void givenNoItemsExistWhenAnyUserAsksForRecommendationsThenNoRecommendationsAreGiven()
	{
		Engine engine = new Engine();
		User Alice = new User("Alice");
		User Bob = new User("Bob");
		engine.addUsers(Alice, Bob);

		assertThat(engine.getRecommendations(Alice), is(empty()));
		assertThat(engine.getRecommendations(Bob),is(empty()));
	}

	@Test
	public void
givenOnlyOneUserExistsWhenThatUserAsksForRecommendationsThenNoRecommendationsAreGiven()
	{
		Engine engine = new Engine();
		User Alice = new User("Alice");
		Item aerobics = new Item("aerobics");
		Item basketball = new Item("basketball");
		engine.addUsers(Alice);
		engine.addItems(aerobics, basketball);

		assertThat(engine.getRecommendations(Alice), is(empty()));
	}

	@Test
	public void
givenAUserWhoHasNotRatedAnyItemsWhenThatUserAsksForRecommendationsThenNoRecommendationsAreGiven()
	{
		Engine engine = new Engine();
		User Alice = new User("Alice");
		User Bob = new User("Bob");
		Item aerobics = new Item("aerobics");
		Item basketball = new Item("basketball");
		engine.addUsers(Alice, Bob);
		engine.addItems(aerobics, basketball);

		engine.like(Bob, basketball);
```

```
                assertThat(engine.getRecommendations(Alice), is(empty()));
        }

        @Test
        public void
givenAUserWhoHasRatedSomeItemsWhenThatUserAsksForRecommendationsThenItemsNotYetRatedByUserAndLikedByAllSim
ilarUsersAreReturned()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cory = new User("Cory");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                engine.addUsers(Alice, Bob, Cory);
                engine.addItems(aerobics, basketball, curling);

                engine.like(Alice, aerobics);

                engine.like(Bob, aerobics);
                engine.like(Bob, basketball);

                engine.like(Cory, aerobics);
                engine.like(Cory, curling);

                assertThat(engine.getRecommendations(Alice), contains(basketball, curling));

        }

        @Test
        public void
givenAUserWithTwoSimilarUsersWhenThatUserAsksForRecommendationThenRecommendationsExcludeItemsDislikedByEit
herUser()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cory = new User("Cory");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                engine.addUsers(Alice, Bob, Cory);
                engine.addItems(aerobics, basketball, curling);

                engine.like(Alice, aerobics);

                engine.like(Bob, aerobics);
                engine.dislike(Bob, basketball);

                engine.like(Cory, aerobics);
                engine.dislike(Cory, curling);

                assertThat(engine.getRecommendations(Alice), not(hasItems(basketball, curling)));
        }

        @Test
        public void
givenAUserWithTwoDissimilarUsersWhenThatUserAsksForRecommendationsThenRecommendationsExcludeItemsLikedByEi
therUser()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cory = new User("Cory");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
```

```
                engine.addUsers(Alice, Bob, Cory);
                engine.addItems(aerobics, basketball, curling);

                engine.like(Alice, aerobics);

                engine.dislike(Bob, aerobics);
                engine.like(Bob, basketball);

                engine.dislike(Cory, aerobics);
                engine.like(Cory, curling);

                assertThat(engine.getRecommendations(Alice), not(hasItems(basketball, curling)));
        }

        @Test
        public void
givenAUserWithTwoDissimilarUsersWhenThatUserAsksForRecommendationsThenRecommendationsIncludeItemsDislikedB
yEitherUser()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cory = new User("Cory");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                engine.addUsers(Alice, Bob, Cory);
                engine.addItems(aerobics, basketball, curling);

                engine.like(Alice, aerobics);

                engine.dislike(Bob, aerobics);
                engine.dislike(Bob, basketball);

                engine.dislike(Cory, aerobics);
                engine.dislike(Cory, curling);

                assertThat(engine.getRecommendations(Alice), hasItems(basketball, curling));

        }

        @Test
        public void
givenAUserWithOneSimilarAndOneDissimilarUserWhenThatUserAsksForRecommendationsThenRecommendationsExcludeIt
emsLikedByBothUsers()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cory = new User("Cory");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                engine.addUsers(Alice, Bob, Cory);
                engine.addItems(aerobics, basketball);

                engine.like(Alice, aerobics);

                engine.like(Bob, aerobics);
                engine.like(Bob, basketball);

                engine.dislike(Cory, aerobics);
                engine.like(Cory, basketball);

                assertThat(engine.getRecommendations(Alice), not(hasItem(basketball)));
        }

        @Test
```

```
        public void
givenAUserWithOneSimilarAndOneDissimilarUserWhenThatUserAsksForRecommendationsThenRecommendationsExcludeIt
emsDislikedByBothUsers()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cory = new User("Cory");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                engine.addUsers(Alice, Bob, Cory);
                engine.addItems(aerobics, basketball);

                engine.like(Alice, aerobics);

                engine.like(Bob, aerobics);
                engine.dislike(Bob, basketball);

                engine.dislike(Cory, aerobics);
                engine.dislike(Cory, basketball);

                assertThat(engine.getRecommendations(Alice), not(hasItem(basketball)));
        }

        @Test
        public void
givenAUserWithOneSimilarUserWhoLikesAnItemAndALessSimilarUserWhoDislikesTheItemWhenThatUserAsksForRecommen
dationsThenTheItemIsRecommended()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cory = new User("Cory");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                engine.addUsers(Alice, Bob, Cory);
                engine.addItems(aerobics, basketball, curling);

                engine.like(Alice, aerobics);
                engine.like(Alice, basketball);

                engine.like(Bob, aerobics);
                engine.like(Bob, basketball);
                engine.like(Bob, curling);

                engine.like(Cory, aerobics);
                engine.dislike(Cory, basketball);
                engine.dislike(Cory, curling);

                assertThat(engine.getRecommendations(Alice), hasItem(curling));
        }

        @Test
        public void
givenAUserWithOneSimilarUserWhoDislikesAnItemAndALessSimilarUserWhoLikesTheItemWhenThatUserAsksForRecommen
dationsThenTheItemIsNotRecommended()
        {
                Engine engine = new Engine();
                User Alice = new User("Alice");
                User Bob = new User("Bob");
                User Cory = new User("Cory");
                Item aerobics = new Item("aerobics");
                Item basketball = new Item("basketball");
                Item curling = new Item("curling");
                engine.addUsers(Alice, Bob, Cory);
                engine.addItems(aerobics, basketball, curling);

                engine.like(Alice, aerobics);
```

```
                engine.like(Alice, basketball);

                engine.like(Bob, aerobics);
                engine.like(Bob, basketball);
                engine.dislike(Bob, curling);

                engine.like(Cory, aerobics);
                engine.dislike(Cory, basketball);
                engine.like(Cory, curling);

                assertThat(engine.getRecommendations(Alice), not(hasItem(curling)));
        }
}
```

Conclusion

In this tutorial, we built a simple recommendation system that uses similarity with other users to calculate new recommendations for a user. We worked in a behavioural test-driven way, starting with requirements that were encoded into tests, and using these tests to drive our implementation. By working in this way, I hope you were able to see that:

1. By directly linking requirements, tests that verify those requirements, and code that implements those requirements, we bring immense clarity to the software development process, and minimize communication gaps resulting from changing requirements or stale tests.
2. When we write tests first, our code is always protected by a test harness. This builds reliability into the application from the very start of the project, allowing us to instantly find defects from known requirements. It also gives us confidence when refactoring.
3. By writing tests that speak directly to requirements, and by letting such tests guide our implementation, we let the design of the system emerge organically. We write just enough code to fulfill the requirements, so the resulting design is usually the simplest possible, which further helps to maintain the codebase.
4. By driving implementation directly from behavioural tests, we let the expected user experience stay in control, rather than having the user experience suffer from a poor implementation.

References

1. Robert Martin (2008). The Truth About BDD. Retrieved from https://sites.google.com/site/unclebobconsultingllc/the-truth-about-bdd
2. Kent Beck (2002). *Test-Driven Development: By Example*. Amazon link https://www.amazon.ca/Test-Driven-Development-Kent-Beck/dp/0321146530
3. Mahmud Ridwan. Building a Simple Collaborative Filtering Recommendation Engine. Retrieved from https://www.toptal.com/algorithms/predicting-likes-inside-a-simple-recommendation-engine