



CHAPTER 10

Acceptance Test

10

Have we implemented a user story correctly

Suppose that the customer of this project is a conference and exhibition organizer. They request us to develop a software system to manage the participants of a particular conference. Suppose that they request us to implement four user stories in this iteration. One of the user stories is shown below:

Name: Import participants

Events:

1. A user asks the system to read the information about a batch of participants from a text file. The information about a participant include ID, password, name, address and email.
2. The system saves the information about the participants. In the future a user can input the ID of a participant, the system will retrieve the information about that participant.
3. The system sends an email to each participant, telling him his ID and password.

Strictly speaking, the underlined portion in the above user story should not appear here because it describes the internal behavior of the system. What is important for the user is that he can retrieve the information of the participants in the future. However, we allow it here because it should help the users understand.

We start by asking the customer about the details of the user story. For example:

- What is the format of the text file? Suppose that the customer says that every line contains a participant. Its various data items are separated by tabs.
- Must the ID, password, name, address and email be present? Suppose that the customer allows the address to be absent, while the other data items must be present. Otherwise, it will skip the line.
- What should the system do if a participant ID already exists? Suppose that the customer hopes to skip the line.

- So on.

We ask the customer, use CRC cards or some other methods to establish or discuss a quick design, write code and improve the design at the same time. Suppose that in two days we have finished all the code and there is nothing to improve in the design. However, we still need to do an important task: Test our code to see if it is implementing that user story correctly.

How to test

How to test? For example, we can run the following "test case":

Test case 1: Import participants

1. Create the following file:

```
p001 123456 Mary Lam abc mary@hotmail.com
p004 888999 John Chan def john@yahoo.com
p002 mypasswd Paul Lei ghi paul@excite.com
```

2. Delete all the participants in the system to prevent the case that p001, p002 or p004 already exists.
3. Run the system and let it import the file above into the database.
4. How to check if it has correctly imported the file? We should have a user story to let a user input a participant ID and then have the system display the information of that participant. We can implement that user story first, and then input p001 as the participant ID, check if the system will display the information of p001 (123456, Mary Lam and etc.), then input p002 and then p004 and etc.
5. How to check if it has sent the emails? We can contact Mary, John and Paul and ask them if they have received the emails and whether the contents of the emails are correct.

This kind of test is called "acceptance test" or "functional test". This kind of test only tests the external behaviors of a system, ignoring what modules (classes) are inside the system.

In order to test whether it is really OK to not to specify the address of a participant, we create a new test case:

Test case 2: Import a participant without address

1. Create the following file:

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
p001 123456 Mary Lam abc mary@hotmail.com
p004 888999 John Chan def john@yahoo.com
p002 mypasswd Paul Lei ghi paul@excite.com
```

2. Delete all the participants in the system.
3. Run the system and let it import the file above into the database.
4. Input p004 as the participant ID and check if the system displays the information of p004.
5. Contact John and ask him if he has received the email and whether the contents of the email are correct.

In order to test whether it will insist that all the data items other than address be specified, we create a new test case:

Test case 3: Import participants without other information

1. Create the following file:

```
p004 123456 Mary Lam abc mary@hotmail.com
p002 888999 John Chan def john@yahoo.com
p005 mypasswd Paul Lei ghi paul@excite.com
p005 secret Mike Chan jkl
```

2. Delete all the participants in the system.
3. Run the system and let it import the file above into the database.
4. Input an empty string, p002, p004 and p005 as the participant ID respectively and check if the system displays the information of any of them (it should not).
5. Contact Mary, Paul and Mike and ask them if they have received any emails (they should not have).

In order to test whether it will not add a participant that already exists, we create a new test case:

Test case 4: Import a participant with a duplicate ID

1. Create the following file:

```
p001 123456 Mary Lam abc mary@hotmail.com
p001 888999 John Chan def john@yahoo.com
```

2. Delete all the participants in the system.

3. Run the system and let it import the file above into the database.
4. Input p001 as the participant ID and check if the system displays the information of Mary instead of John.
5. Contact Mary and ask her if she has received the email.
6. Contact John and ask him if he has received any email (he should not have).

Problems with manual tests

We have created four test cases. They have one thing in common: Each step in the test cases (create text file, run the system, input a participant ID, check if the correct information is displayed, contact the people to check if they have received the emails and etc.) needs to be performed manually. Therefore, it takes a lot of effort and time.

This is a serious problem. When we implement the other user stories, we will have to modify the code or add new code. This may introduce bugs. Therefore, whenever the code is changed we hope to run the above test cases again to see if they still pass. If not, we can start finding the bug immediately (where is the bug? Very likely it is in the code that we just modified or added). However, because it takes too much effort and time to run these test cases, it is impossible for us to do it often.

In order to solve this problem, we hope that the test cases can be run automatically without human intervention. This kind of test is called "automated acceptance test".

Automated acceptance tests

We can write the code below to automatically run the four test cases above, where each method corresponds to one test case:

```
class AcceptTestForImportParticipants {
    static void testImport() {
        ...
    }
    static void testImportWithoutAddress() {
        ...
    }
    static void testImportWithoutOtherInfo() {
        ...
    }
    static void testImportDupId() {
        ...
    }
}
```

```
}

```

Where, testImport will implement test case 1. Before writing testImport, let's see this test case again:

Test case 1: Import participants

1. Create the following file:

```
p001 123456 Mary Lam abc mary@hotmail.com
p004 888999 John Chan def john@yahoo.com
p002 mypasswd Paul Lei ghi paul@excite.com
```

2. Delete all the participants in the system to prevent the case that p001, p002 and p004 already exist.
3. Run the system and let it import the file above into the database.
4. How to check if it has correctly imported the file? We should have a user story to let a user input a participant ID and then have the system display the information of that participant. We can implement that user story first, and then input p001 as the participant ID, check if the system will display the information of p001 (123456, Mary Lam and etc.), then input p002 and then p004 and etc.
5. How to check if it has sent the emails? We can contact Mary, John and Paul and ask them if they have received the emails and whether the contents of the emails are correct.

Now we need to implement each step above. We will use a "Command" to represent each step. To run a step (a command), just call the run method of the command. If it succeeds, run should return true:

```
interface Command {
    boolean run();
}
```

Now, implement the first step (create a text file):

```
class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new CreateImportFileCommand(
                "p001\t123456\tMary Lam\tabc\tmary@hotmail.com\n"+
                "p004\t888999\tJohn Chan\tdef\tjohn@yahoo.com\n"+
                "p002\tmyspasswd\tPaul Lei\tghi\tpaul@excite.com"),
            ...
        };
        runCommands(commands);
    }
    static void runCommands(Command commands[]) {
        for (int i = 0; i < commands.length; i++) {
```

```

        if (!commands[i].run()) {
            System.out.println("Test failed!");
        }
    }
}
}
class CreateImportFileCommand implements Command {
    String fileContent;
    CreateImportFileCommand(String fileContent) {
        this.fileContent = fileContent;
    }
    boolean run() {
        FileWriter fileWriter = new FileWriter("sourceFile.txt");
        try {
            fileWriter.write(fileContent);
            return true;
        } finally {
            fileWriter.close();
        }
    }
}
}
}

```

Implement the second step (Delete all the participants in the system):

```

class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new CreateImportFileCommand(...),
            new DeleteAllParticipantsCommand(),
            ...
        };
        runCommands(commands);
    }
}
}

```

In order to implement `DeleteAllParticipantsCommand`, we assume that the system has provided such a function in the `Participants` class. `DeleteAllParticipantsCommand` can simply call it:

```

class DeleteAllParticipantsCommand implements Command {
    boolean run() {
        ConferenceSystem.getInstance().parts.deleteAll();
        return true;
    }
}
class ConferenceSystem {
    Participants parts;
    static ConferenceSystem getInstance() {
        ...
    }
}
class Participants {
    void deleteAll() {
        ...
    }
}
}

```

Implement the third step (ask the system to import from the file):

```
class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new CreateImportFileCommand(...),
            new DeleteAllParticipantsCommand(),
            new ImportParticipantsFromFileCommand(),
            ...
        };
        runCommands (commands);
    }
}
class ImportParticipantsFromFileCommand implements Command {
    boolean run() {
        ConferenceSystem.getInstance().parts.importFromFile("sourceFile.txt");
        return true;
    }
}
class Participants {
    void deleteAll() {
        ...
    }
    void importFromFile(String path) {
        ...
    }
}
```

Implement the fourth step (Retrieve p001 and etc. from the system):

```
class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new CreateImportFileCommand(...),
            new DeleteAllParticipantsCommand(),
            new ImportParticipantsFromFileCommand(),
            new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            ...
        };
        runCommands (commands);
    }
}
class CheckParticipantStoredCommand implements Command {
    Participant part;
    CheckParticipantStoredCommand(String partId, String password, ...) {
        part = new Participant(partId, password, ...);
    }
    boolean run() {
        return ConferenceSystem.getInstance().parts.checkParticipantStored(part);
    }
}
class Participants {
    boolean checkParticipantStored(Participant part) {
        return part.equals(getParticipantById(part.getId()));
    }
}
```

```

    Participant getParticipantById(String partId) {
        ...
    }
}
class Participant {
    boolean equals(Object obj) {
        ...
    }
}

```

Now we need to implement the fifth step (check if Mary and etc. have received the emails), but it is rather difficult. For example, it is not easy to write code to check the mail box of Mary. At the same time, we don't have her password. Besides, she may have downloaded the mail before we check. Also, the email may be just on its way to her mail box. We can only lower our expectation. Instead of checking whether she has received the email, we check whether the system has sent the email. To check whether the system has sent the email, strictly speaking we need to check whether it has issued the SMTP commands, whether the recipient and email contents embedded in the commands are correct. This is also not easy to do. Therefore, we lower our expectation again. We assume that the system will record the recipient and contents of every email it sends. In the test we only check this record:

```

class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new CreateImportFileCommand(...),
            new DeleteAllParticipantsCommand(),
            new ImportParticipantsFromFileCommand(),
            new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id
& ..."),
            new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ...
"),
            new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ...
"),
            ...
        };
        runCommands(commands);
    }
}
class CheckRemoveOldestMailCommand implements Command {
    MailRecord mailRecord;
    CheckRemoveOldestMailCommand(String recipientEmail, String mailContent) {
        mailRecord = new MailRecord(recipientEmail, mailContent);
    }
    boolean run() {
        MailLog mailLog = ConferenceSystem.getInstance().mailLog;
        return mailRecord.equals(mailLog.takeOldestMailRecord());
    }
}
class ConferenceSystem {
    Participants parts;
    MailLog mailLog;
    static ConferenceSystem getInstance() {

```

```

    ...
}
}
class MailLog {
    MailRecord mailRecords[];
    MailRecord takeOldestMailRecord() {
        ...
    }
}
class MailRecord {
    MailRecord(String recipientEmail, String mailContent) {
        ...
    }
    boolean equals(Object obj) {
        ...
    }
}
}

```

In fact, there is a problem: When the test case is run, MailLog may contain some residual records. When we take the oldest MailRecord, we will get the wrong one. To solve this problem, testImport should empty the MailLog at the beginning:

```

class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new EmptyMailLogCommand(),
            new CreateImportFileCommand(...),
            new DeleteAllParticipantsCommand(),
            new ImportParticipantsFromFileCommand(),
            new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id
& ..."),
            new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ...
"),
            new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ...
"),
            ...
        };
        runCommands(commands);
    }
}
class EmptyMailLogCommand implements Command {
    boolean run() {
        MailLog mailLog = ConferenceSystem.getInstance().mailLog;
        mailLog.empty();
        return true;
    }
}
class MailLog {
    MailRecord mailRecords[];
    MailRecord takeOldestMailRecord() {
        ...
    }
    void empty() {
        ...
    }
}

```

```

    }
}

```

We find that at the beginning of a test case, we commonly need to empty many things such as MailLog, Participants and etc. Therefore, we seem to need a command to empty or initialize all the things in the system. Let's call it SystemInit. Therefore, we no longer needs the DeleteAllParticipants command and the EmptyMailLog command:

```

class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new SystemInitCommand(),
            new CreateImportFileCommand(...),
            new ImportParticipantsFromFileCommand(),
            new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id
& ..."),
            new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ...
"),
            new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ...
"),
            ...
        };
        runCommands(commands);
    }
}

class SystemInitCommand implements Command {
    boolean run() {
        ConferenceSystem.getInstance().init();
        return true;
    }
}

class ConferenceSystem {
    Participants parts;
    MailLog mailLog;
    void init() {
        parts.deleteAll();
        mailLog.empty();
    }
}

```

We have completely automated test case 1. Below is the complete code:

```

class ConferenceSystem {
    Participants parts;
    MailLog mailLog;
    void init() {
        parts.deleteAll();
        mailLog.empty();
    }
    static ConferenceSystem getInstance() {
        ...
    }
}

class Participants {

```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```

    void deleteAll() {
        ...
    }
    void importFromFile(String path) {
        ...
    }
    boolean checkParticipantStored(Participant part) {
        return part.equals(getParticipantById(part.getId()));
    }
    Participant getParticipantById(String partId) {
        ...
    }
}
class Participant {
    boolean equals(Object obj) {
        ...
    }
}
class MailRecord {
    MailRecord(String recipientEmail, String mailContent) {
        ...
    }
    boolean equals(Object obj) {
        ...
    }
}
class MailLog {
    MailRecord mailRecords[];
    MailRecord takeOldestMailRecord() {
        ...
    }
}
class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new SystemInitCommand(),
            new CreateImportFileCommand(...),
            new ImportParticipantsFromFileCommand(),
            new CheckParticipantStoredCommand("p001", "123456", "Mary
Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id
& ..."),
            new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ...
"),
            new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ...
"),
            ...
        };
        runCommands(commands);
    }
    static void testImportWithoutAddress() {
        ...
    }
    static void testImportWithoutOtherInfo() {
        ...
    }
    static void testImportDupId() {
        ...
    }
}

```

```
    }
    static void runCommands(Command commands[]) {
        for (int i = 0; i < commands.length; i++) {
            if (!commands[i].run()) {
                System.out.println("Test failed!");
            }
        }
    }
}
interface Command {
    boolean run();
}
class SystemInitCommand implements Command {
    boolean run() {
        ConferenceSystem.getInstance().init();
        return true;
    }
}
class CreateImportFileCommand implements Command {
    String fileContent;
    CreateImportFileCommand(String fileContent) {
        this.fileContent = fileContent;
    }
    boolean run() {
        FileWriter fileWriter = new FileWriter("sourceFile.txt");
        try {
            fileWriter.write(fileContent);
            return true;
        } finally {
            fileWriter.close();
        }
    }
}
class ImportParticipantsFromFileCommand implements Command {
    boolean run() {
        ConferenceSystem.getInstance().parts.importFromFile("sourceFile.txt");
        return true;
    }
}
class CheckParticipantStoredCommand implements Command {
    Participant part;
    CheckParticipantStoredCommand(String partId, String password, ...) {
        part = new Participant(partId, password, ...);
    }
    boolean run() {
        return ConferenceSystem.getInstance().parts.checkParticipantStored(part);
    }
}
class CheckRemoveOldestMailCommand implements Command {
    MailRecord mailRecord;
    CheckRemoveOldestMailCommand(String recipientEmail, String mailContent) {
        mailRecord = new MailRecord(recipientEmail, mailContent);
    }
    boolean run() {
        MailLog mailLog = ConferenceSystem.getInstance().mailLog;
        return mailRecord.equals(mailLog.takeOldestMailRecord());
    }
}
```

Commands should NOT include domain logic

All the XXXCommand classes above have one thing in common: They all contain just a little code. This is not by chance. This is because they are simulating the requests or actions of the users (e.g., ask the system to import a file). As the system already contains code to satisfy these requests, the commands only need to call the existing code in the system.

When we are writing these command classes, we may find that there is no existing code in the system for use. For example, when writing the CheckParticipantStoredCommand, we may find that there is no method like getParticipantById in the Participants class. If we are not careful, we may choose to implement this function directly in CheckParticipantStoredCommand:

```
class CheckParticipantStoredCommand implements Command {
    Participant part;
    ..
    boolean run() {
        return part.equals(getParticipantById(part.getId()));
    }
    Participant getParticipantById(String partId) {
        //lots of code to read the data from the database.
        ...
        ...
        ...
    }
}
```

However, there is a problem here. Because the users need to retrieve the information of the participants, getParticipantById has implemented a function really needed by the users (user story). As CheckParticipantStoredCommand is only used for testing, when the system is delivered, all these XXXCommand classes including CheckParticipantStoredCommand should be excluded. But here we would like to keep getParticipantById because it includes useful functionality (domain logic).

Therefore, all these XXXCommand classes should only implement test cases, not domain logic. Domain logic should be implemented by the "regular" classes in the system such as Participants, Participant and etc. The XXXCommand classes should only need to call these "regular" classes. If you find that a participant Command class contains "too much" code, or some other classes in the system need to call this Command class, you need to check if this Command class has implemented the business logic needed by the system. If yes, you should move the logic into the "regular" classes in the system.

Writing test cases first as the requirements

Suppose that after implementing this "import participants" user story, we work with the customer to write all the four automated test cases. We run them to see if they pass. If yes, it

means the user story has been implemented (for the best results, try to run them at least once in front of the customer). However, unfortunately, the system fails in a test case that tries to import the following file given by the customer:

```
p001 123456      Mary Lam   abc      mary@extremely.long.domain.com
p004 888999      John Chan  def      john@yahoo.com
p002 mypasswd Paul Lei   ghi      paul@excite.com
```

After two hours of debugging, we finally find that the maximum size of the email address in the database is set to 26 characters, but now the email of Mary contains 32 characters. Therefore, the database only has `mary@extremely.long.doma`, not `mary@extremely.long.domain.com`. Therefore, when retrieving `p001`, `checkParticipantStored` fails.

If at the beginning when we started to implement this user story, we had worked with the customer to write the test cases, we would have learned from the customer or from the test cases that we needed at least 32 characters to store an email address. We would have avoided the mistake when we created the database. We would also have saved that two hours of debugging.

Therefore, test cases should be written before we start to implement a user story. Test cases are an important component of our software requirements (the live customer is the other component). As part of the requirements, the test cases can guide our implementation work. The advantages of the live customer are that the information is most updated and that the communication is efficient (conversation is easier than reading); The advantages of (automated) test cases are that they are accurate and can be run often.

Make the test cases understandable to the customer

Only the customer knows the requirements. Therefore, only he has the right to specify the contents of the test cases. What we can do at most is just to help him write them down. We must never determine the contents of the test cases on his behalf. At the moment, we use Java code to specify the contents of the test cases, which can't be understood by a regular customer who is not a developer:

```
class AcceptTestForImportParticipants {
    static void testImport() {
        Command commands[] = {
            new SystemInitCommand(),
            new CreateImportFileCommand(...),
            new ImportParticipantsFromFileCommand(),
            new CheckParticipantStoredCommand("p001", "123456", "Mary Lam", ...),
            new CheckParticipantStoredCommand("p002", ...),
            new CheckParticipantStoredCommand("p004", ...),
            new CheckRemoveOldestMailCommand("mary@hotmail.com", "This is your Id
& ..."),
            new CheckRemoveOldestMailCommand("john@yahoo.com", "This is your Id & ...
"),
        },
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agilekills.org

```

        new CheckRemoveOldestMailCommand("paul@excite.com", "This is your Id & ...
    ),
    };
    runCommands (commands);
}
}

```

This is a serious problem. If the customer can't understand the test cases, how can he judge whether what we write is exactly what he has in mind? In order to make this test case easier to understand, we can rewrite it as a text file named testImport.test with the following contents:

```

SystemInit
CreateImportFile
    p001,123456,Mary Lam,abc,mary@hotmail.com
    p004,888999,John Chan,def,john@yahoo.com
    p002,mypasswd,Paul Lei,ghi,paul@excite.com
CreateImportFileEnd
ImportFromFile
CheckParticipantStored,p001,123456,Mary Lam,abc,mary@hotmail.com
CheckParticipantStored,p002,mypasswd,Paul Lei,ghi,paul@excite.com
CheckParticipantStored,p004,888999,John Chan,def,john@yahoo.com
CheckRemoveOldestMail,mary@hotmail.com,This is your Id &...
CheckRemoveOldestMail,john@yahoo.com,This is your Id &...
CheckRemoveOldestMail,paul@excite.com,This is your Id &...

```

Let's call this kind of file a "test file". In a test file basically every command takes one line, with the exception of CreateImportFile which takes five lines:

```

CreateImportFile
    p001,123456,Mary Lam,abc,mary@hotmail.com
    p004,888999,John Chan,def,john@yahoo.com
    p002,mypasswd,Paul Lei,ghi,paul@excite.com
CreateImportFileEnd

```

From the view of the customer, the system can execute this testImport.test file. That is, it can sequentially execute each command in this file.

In order to support this kind of test file, we need to know how to read in a test file and then generate a list of corresponding commands:

```

class TestFileParser {
    CommandParser commandParsers[] = {
        new SystemInitCommandParser(),
        new CheckParticipantStoredCommandParser(),
        new CreateImportFileCommandParser(),
        ...
    };
    Command[] parseCommandsFromFile(String pathToTestFile) {
        TokenIterator tokenIterator = parseFileToTokens(pathToTestFile);
        return parseCommands(tokenIterator);
    }
    TokenIterator parseFileToTokens(String path) {
        ...
    }
    Command[] parseCommands(TokenIterator tokenIterator) {
        Command commands[];

```

```
        while (tokenIterator.hasToken()) {
            Command nextCommand = parseCommand(tokenIterator);
            append nextCommand to commands;
        }
        return commands;
    }
}
Command parseCommand(TokenIterator tokenIterator) {
    for (int i = 0; i < commandParsers.length; i++) {
        Command command = commandParsers[i].parse(tokenIterator);
        if (command != null) {
            return command;
        }
    }
    throw new CommandSyntaxException();
}
}
interface TokenIterator {
    boolean hasToken();
    String peekToken();
    String takeToken();
    void next();
}
interface CommandParser {
    Command parse(TokenIterator tokenIterator);
}
class SystemInitCommandParser implements CommandParser {
    Command parse(TokenIterator tokenIterator) {
        if (tokenIterator.peekToken().equals("SystemInit")) {
            tokenIterator.next();
            return new SystemInitCommand();
        } else {
            return null;
        }
    }
}
class CheckParticipantStoredCommandParser implements CommandParser {
    Command parse(TokenIterator tokenIterator) {
        if (tokenIterator.peekToken().equals("CheckParticipantStored")) {
            tokenIterator.next();
            String partId = tokenIterator.takeToken();
            String password = tokenIterator.takeToken();
            String name = tokenIterator.takeToken();
            String address = tokenIterator.takeToken();
            String email = tokenIterator.takeToken();
            return new CheckParticipantStoredCommand(
                partId,
                password,
                name,
                address,
                email);
        } else {
            return null;
        }
    }
}
class CreateImportFileCommandParser implements CommandParser {
    ...
}
```

A test file doesn't have to be a text file

A test file doesn't have to be a text file. For example, it may be an Excel file:

SystemInit					
CreateImportFile					
	p001	123456	Mary Lam	abc	mary@hotmail.com
	p004	888999	John Chan	def	john@yahoo.com
	p002	mypasswd	Paul Lei	ghi	paul@excite.com
ImportFromFile					
CheckParticipantStored	p001	123456	Mary Lam	abc	mary@hotmail.com
CheckParticipantStored	p002	mypasswd	Paul Lei	ghi	paul@excite.com
CheckParticipantStored	p004	888999	John Chan	def	john@yahoo.com
...					

Of course, if we really used an Excel file, our TestFileParser would have to be changed accordingly. It is also possible to use an HTML file or some other formats.

Use test cases to prevent the system functionality from downgrading

Whenever we work with the customer to create a test file, we place it into a folder named "Queued". At the beginning all the test files are placed here. Whenever we make a test file pass, we move it from Queued into another folder named "Passed". Whenever we modify the code or add some code, we need to run all the test files in Passed again to make sure that they still pass. If a certain test file fails, we must modify the code to make it pass again. We must not move it from Passed back to Queued. That is, once a test file is moved into Passed, it will never get out of there (unless the customer finds an error in the test file itself), i.e., the movement from Queued to Passed is strictly one way only and can't be reversed.

Because this movement is one way only, there will be more and more test files in Passed and less and less test files in Queued, meaning that the functionality of our system can only go up and up. It will never go down. Finally when Queued becomes empty and if the customer has no more new test cases, we are done.

References

- <http://c2.com/cgi/wiki?AcceptanceTest>.
- Ward Cunningham has written an acceptance test framework called "Fit". It is available at <http://fit.c2.com>.
- Based on Fit, FitNesse is a web-based environment allowing developers to create and run acceptance tests. It is available at <http://fitnesse.org>.
- Ron Jeffries, Ann Anderson, Chet Hendrickson, Extreme Programming Installed, Addison-Wesley, 2000.



Chapter exercises

1. Write the test cases 2, 3 and 4 for the user story "import participants" as test files and implement the commands needed (you can assume the system already contains the required functions).

Sample solutions

1. Write the test cases 2, 3 and 4 for the user story "import participants" as test files and implement the commands needed (you can assume the system already contains the required functions).

The test file for test case 2:

```
SystemInit
CreateImportFile
    p001,123456,Mary Lam,abc,mary@hotmail.com
    p004,888999,John Chan,,john@yahoo.com
    p002,mypasswd,Paul Lei,ghi,paul@excite.com
CreateImportFileEnd
ImportFromFile
CheckParticipantStored,p001,123456,Mary Lam,abc,mary@hotmail.com
CheckParticipantStored,p002,mypasswd,Paul Lei,ghi,paul@excite.com
CheckParticipantStored,p004,888999,John Chan,,john@yahoo.com
CheckRemoveOldestMail,mary@hotmail.com,This is your Id &...
CheckRemoveOldestMail,john@yahoo.com,This is your Id &...
CheckRemoveOldestMail,paul@excite.com,This is your Id &...
```

The test file for test case 3:

```
SystemInit
CreateImportFile
    ,123456,Mary Lam,abc,mary@hotmail.com
    p004,888999,,def,john@yahoo.com
    p002,,Paul Lei,ghi,paul@excite.com
    p005,secret,Mike Chan,jkl,
CreateImportFileEnd
ImportFromFile
CheckNoParticipantsStored
CheckMailLogEmpty
```

We need to implement two new commands: `CheckNoParticipantsStored` and `CheckMailLogEmpty`.

```
class CheckNoParticipantsStoredCommand implements Command {
    boolean run() {
        Participants parts = ConferenceSystem.getInstance().parts;
        return parts.isEmpty();
    }
}
class CheckMailLogEmptyCommand implements Command {
    boolean run() {
        MailLog mailLog = ConferenceSystem.getInstance().mailLog;
        return mailLog.isEmpty();
    }
}
```

The test file for test case 4:

```
SystemInit
CreateImportFile
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agilekills.org

```
p001,123456,Mary Lam,abc,mary@hotmail.com
p001,888999,John Chan,def,john@yahoo.com
CreateImportFileEnd
ImportFromFile
CheckParticipantStored,p001,123456,Mary Lam,abc,mary@hotmail.com
CheckRemoveOldestMail,mary@hotmail.com,This is your Id &...
```