



CHAPTER 8

Managing Software Projects with User Stories

What is a user story

Suppose that the customer of this project is a manufacturer of vending machines for soft drinks. They request us to develop software to control their vending machines. We can contact their marketing manager for the requirements on this system software. Therefore, their marketing manager is our customer. He once said: "Whenever someone inserts some money, the vending machine will display how much he has inserted so far. If the money is enough to buy a certain type of soft drink, the light inside the button representing that type of soft drink will be turned on. If he presses one such button, the vending machine will dispense a can of soft drink and return the changes to him."

The description above describes a process by which a user may use the system to complete a certain task that is valuable to him (buy a can of soft drink). Such a process is called a "use case" or "user story". That is, what the customer said above describes a user story.

If we would like to write down a user story, we may use the format below:

Name: Sell soft drink

Events:

1. A user inserts some money.
2. The vending machine displays how much money he has inserted so far.
3. If the money is enough to buy a certain type of soft drink, the vending machine lights up the button representing that type of soft drink.
4. The user presses a certain lit-up button.
5. The vending machine sells a can of soft drink to him.
6. The vending machine returns the changes to him.

Note that the events in a user story typically look like this:

1. User does XX.
2. System does YY.
3. User does ZZ.
4. System does TT.
5. ...

User story only describes the external behaviors of a system

A user story should only describe the external behaviors of a system that can be understood by the customer. It completely ignores the internal behaviors of the system. For example, the parts underlined in the user story below are internal behaviors and should not appear in the user story at all:

1. A user inserts some money.
2. The vending machine deposits the money into its money box, sends a command to the screen to display how much money he has inserted so far.
3. The vending machine looks up the price list in its database to check if the money is enough to buy a certain type of soft drink. If yes, the vending machine lights up the button representing that type of soft drink.
4. The user presses a certain lit-up button.
5. The vending machine sells a can of soft drink to him and reduces the inventory in the database.
6. The vending machine returns the changes to him.

This is a common mistake when describing user stories, no matter it is in written or spoken form. In particular, never mention things like databases, records or fields that make no sense to the customer. Stay alerted.

Estimate release duration

What are user stories for? Suppose that the customer hopes to have the system delivered in

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

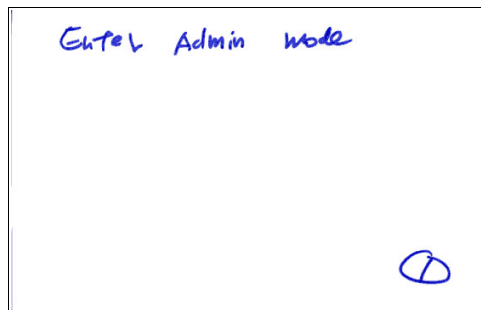
50 days. Are we going to make it? To answer that question, at the beginning of the project, we will try to find out all the user stories and estimate how much time we will need for each. How? For example, we may have gathered the following user stories from the customer:

User story	Brief description
Sell soft drinks	As mentioned above.
Cancel purchase	After inserting some coins, the user can cancel the purchase.
Enter administration mode	An authorized person can enter administration mode and then add the inventory, set the prices, take the money and etc.
Add soft drinks	An authorized person can add the inventory after entering administration mode.
Take money	An authorized person can take the money after entering administration mode.
Security alarm	If something usually happens, it can turn on its security alarm.
Print monthly sales report	An authorized person can download the data and then print a monthly sales report.

Then find a user story that is among those that are easiest (by "easy", we mean "take less time to implement"). We don't have to be very accurate. Just pick one that feels easy. For example, let's say "enter administration mode" is such a user story. Then we will declare that this user story is worth 1 "story point":

User story	Story points
Sell soft drinks	
Cancel purchase	
Enter administration mode	1
Add soft drinks	
Take money	
Security alarm	
Print monthly sales report	

Instead of having a list like above, you will probably use a 4"x6" index card to represent a user story and write the story point on it:



Such a card is called a "story card".

Then, consider the other user stories. For example, for "take money", we think its should be about as easy as "enter administration mode", so it is also worth 1 story point. We will show it on our list but you will write it on the story card for "take money":

User story	Story points
Sell soft drinks	
Cancel purchase	
Enter administration mode	1
Add soft drinks	
Take money	1
Security alarm	
Print monthly sales report	

For "cancel purchase", we think it should be about twice as hard as "take money", so it is worth 2 story points:

User story	Story points
Sell soft drinks	
Cancel purchase	2
Enter administration mode	1
Add soft drinks	
Take money	1
Download sales data to notebook	
Print monthly sales report	

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

For "sell soft drinks", we think it should be twice as hard as "cancel purchase", so it is worth 4 story points:

<i>User story</i>	<i>Story points</i>
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	
Take money	1
Security alarm	
Print monthly sales report	

For "add soft drinks", we think it should be harder than "take money" but easier than "sell soft drinks", so it should be somewhere between 1 and 4. Is it harder than "cancel purchase (2)"? We think so, so it is worth 3 story points:

<i>User story</i>	<i>Story points</i>
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm	
Print monthly sales report	

Similarly, we think "Security alarm" should be somewhat easier than "add soft drinks", so it is worth 2 story points:

<i>User story</i>	<i>Story points</i>
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm	2

<i>User story</i>	<i>Story points</i>
Print monthly sales report	

"Print monthly sales report" should be as hard as "sell soft drinks", so it is worth 4 story points and we have totally 17 story points:

<i>User story</i>	<i>Story points</i>
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm	2
Print monthly sales report	4
Total	17

Now pick any user story and estimate how much time it would take you (an individual) to implement. Suppose as we have done something similar to "Take money" before, so we pick it and estimate that it would take 5 days to implement. As this story is worth one story point, it means that we estimate that it takes 5 days to do one story point. As we have totally 17 story points to do, we should need $17 \times 5 = 85$ days to finish the project (if only one developer works on it). Suppose that there are two developers on the team, so we need $85/2 = 43$ days.

So, it seems that we are going to meet the deadline (50 days). But it is too early to tell! This estimate is more a guess than an estimate. Usually developers are extremely poor at workload estimation. In fact it is quite common for people to under-estimate the workload by a factor of four! So, how to get a better estimate?

Actually how fast we can go

To get a better estimate, we should ask the customer to give us two weeks to work on the user stories and measure how many story points we can do. We call a period of two weeks an "iteration".

Which stories to implement in the first iteration? It is totally up to the customer to decide. However, the total number of story points must not exceed our capacity. More specifically, because an iteration contains 10 days (assuming we don't work on weekends) and it is currently estimated that it takes 5 days for one developer to implement one story point, as

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

there are two developers on the team, we should be able to do $(10/5)*2=4$ story points in one iteration.

Then, customer will chooses any user stories that he would like to get implemented in this iteration. The customer should choose those that he considers the most important, regardless of their apparent ordering, as long as the total doesn't exceed 4 story points. For example, if the sales report and taking the money are the most important stories for him, he can choose them without choosing "sell soft drinks":

<i>User story</i>	<i>Story points</i>
Take money	1
Generate monthly sales report in text	2
Total	3

Suppose that when the iteration is ended, we have completed "Take money" but the "Monthly sales report" is not yet completed. Suppose that we estimate that it would take 0.5 story point to complete the missing part. It means we have done only $3-0.5=2.5$ story points in this iteration (worse than the original estimate of 4).

This number of 2.5 story points is called our current "velocity". It means 2.5 story points can be done by the team in one iteration. It is very useful for us. It is can be used in two ways.

First, in the next iteration we will assume that we can do 2.5 story points only, and the user stories selected by the customer cannot exceed 2.5 story points.

Second, after getting the velocity of the first iteration, we should re-estimate the release duration. Originally we estimated that it takes 5 days for one developer to do a story point. Now we don't need this estimate anymore, because we have the actual data. We know that the team (2 developers) can finish 2.5 story point in one iteration (10 days). Because we have 17 story points to do for the release, we will need $17/2.5=7$ iterations, i.e., 14 weeks, which is 70 days. So, it means we are not going to meet the deadline (50 days)! What should we do?

What if we are not going to make it

Obviously we can't finish all the user stories. More specifically, in 50 days we can do only $50/10*2.5=12.5$ story points. As there are totally 17 story points, we should ask the customer to take out some story cards that are worth 4.5 story points in total and delay them to the next release. The customer should delay those that are not as important as the rest. For example, the customer may delay the report printing:

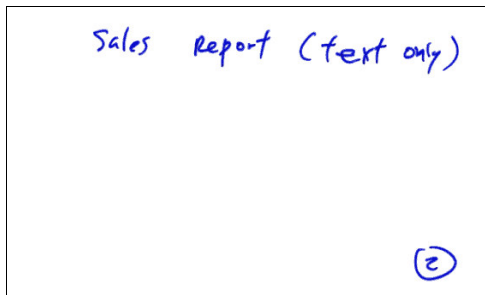
<i>User story</i>	<i>Story points</i>
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm	2
Print monthly sales report	4
Total	13

Then he will need to delay another story that is worth at least 0.5 point.

What if the sales report is extremely important for the customer and there is no other stories that can be delayed? Then we may try to simplify some of the stories. For example, originally the sales report is supposed to be done using a third party reporting library and a pie chart is required, if we can generate the report in plain text format (to be imported into Excel for further processing), we think we can reduce the story points from 4 to 2. This will save 2 story points. If the customer agrees, we can split the "print monthly sales report" into two stories "generate monthly sales report in text" and "graphical report" and delay the latter to the next release. You will do that by tearing up the "print monthly sales report" card:



and create two new cards:



Then take the "graphical report" card out of this release.

<i>User story</i>	<i>Story points</i>
Sell soft drinks	4
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm	2
Generate monthly sales report in text	2
Total	15

For the other 2.5 story points, we can offer to simplify say "sell soft drinks". Suppose that it is originally supposed to support different types of soft drinks with different prices. If we only support a single type of soft drinks and thus a single price, then we think we can reduce the story point from 4 to 2. If the customer agrees, we can split the "sell soft drinks" into two stories "sell soft drinks (single type)" and "sell multiple soft drink types" and delay the latter to the next release:

<i>User story</i>	<i>Story points</i>
Sell soft drinks (single type)	2
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1

<i>User story</i>	<i>Story points</i>
Security alarm	2
Generate monthly sales report in text	2
Total	13

For the other 0.5 story points, we can offer to simplify say "security alarm". Suppose that it is originally supposed to trigger both the local alarm and inform a specified police station. If we only trigger the local alarm, then we think we can reduce the story point from 2 to 1. If the customer agrees, we can split the "security alarm" story into two stories "security alarm (local)" and "notify police" and delay the latter to the next release:

<i>User story</i>	<i>Story points</i>
Sell soft drinks (single type)	2
Cancel purchase	2
Enter administration mode	1
Add soft drinks	3
Take money	1
Security alarm (local)	1
Generate monthly sales report in text	2
Total	12

Now there are only 12 story points to do in the release (≤ 12.5). The process above of choosing what stories to include in the current release is called "release planning".

Meeting release deadline by adding developers

In the example above, we try to meet the release deadline by delaying user stories to the next release. This is called "controlling the scope" and is usually the best way. However, if you have tried it but it doesn't work, you may keep the user stories but add more developers. In this example, assume that we would need "n" developers so that in 50 days we can do 17 story points, i.e., $(50/10) \times 2.5 \times (n/2) = 17$. It means $n = 2.7$, i.e., we need 3 developers. However:

- A larger team will require more communication between the team members. Therefore doubling the team size will not shorten the release duration by half. It may just shorten it by say 1/3. If the team gets larger than 10 developers, adding more people will probably only slow down the project.

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

- Adding team members near the end will only slow down the project because the new members don't understand the system enough to do anything useful to it. So it has to be done early.

Steering the project according to the velocity

The velocity of 2.5 story points per iteration is just the velocity of the first iteration, for the second iteration it may turn out to be say 2 or 3 (usually it shouldn't change by too much). If it is 2, then for the third iteration we will assume a velocity of 2 and the user can choose stories totaling up to 2 story points.

For most projects, the velocity will become stable very soon (e.g., after a few iterations). When it does, we should re-estimate the release duration and perform release planning again. If the velocity suggests that we can do say 3 story points instead of 2.5 story points, we will let the customer choose more stories to include in the release. On the contrary, if the velocity suggests that we can do say only 2 story points, we will ask the customer to take out some more stories (split some stories if required), or add more developers if the team is still very small, it is still early in the project and it is absolutely necessary.

How much details do we need to estimate a user story

At the beginning of the project, we need to find out all the major user stories in the release and estimate the story point of each. How to estimate the story points of a user story say like "sell soft drinks"? The name "sell soft drinks" ignores many details such as: What type of money a user can insert? Are notes OK? Is RMB (Chinese currency) OK? What is the intensity of a lit-up button? Can one type of soft drink be presented by two buttons? After pressing a button, will it be turned off? When returning the changes can we simply return lots of 10 cents?

Do we need to find out all these details? We shouldn't need to know the intensity of the button, because it shouldn't affect the workload. However, how the changes should be returned seems to affect the workload and therefore should be determined (returning a heap of 10 cents should be easy; having to use the minimum number of coins may be quite difficult). Whether it needs to handle multiple currencies should also be determined.

In general, we don't need to worry too much about missing the details. For each user story, it is generally enough to ask a few "important" questions.

What if we can't estimate comfortably

If we can't comfortably estimate a user story, there are some possibilities:

1. The user story is too large. In this case, we can breakdown the user story into several, e.g.:
 - New user story 1: Display the total amount.
 - New user story 2: Light up a button when the total amount is enough.
 - New user story 3: Press a lit-up button to buy soft drink.
2. We have never programmed a vending machine. Therefore, we don't know how hard or easy it is to control it. In this case, we should run some simple experiments such as program to ask a vending machine to return some money. This kind of experiment is called a "spike".

Iteration planning

The whole project will be implemented in iterations. At the start of each iteration, we will let the customer choose those user stories that he would like to get implemented in this iteration. The customer should choose those that he considers most important, regardless of their apparent ordering, as long as the total doesn't exceed the current velocity (2.5 story points). For example, if the sales report is the most important story for him, he can choose it without choosing "sell soft drinks":

<i>User story</i>	<i>Story points</i>
Generate monthly sales report in text	2
Total	2

How can we generate the sales report if the system can't sell yet ("sell soft drinks" is not done yet)? Yes, we can. We can hard code some sales data into the system so that the sales report can be generated.

What if he would like to do:

<i>User story</i>	<i>Story points</i>
Sell soft drinks (single type)	2
Generate monthly sales report in text	2

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

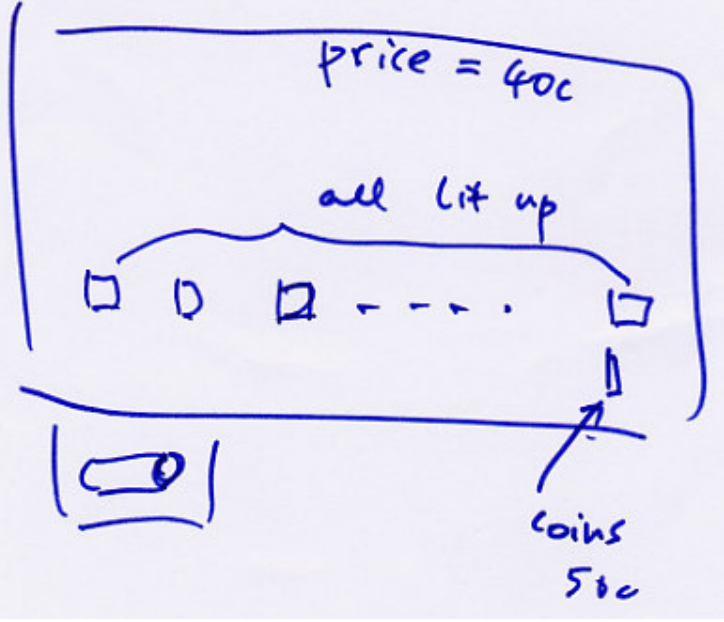
<i>User story</i>	<i>Story points</i>
Total	4

The problem here is that the total is 4 story points and exceeds 2.5. In that case, we can try to simplify by splitting up the stories. For example, we can skip the function of returning changes to the user when selling soft drinks. This may reduce the story points by 1.5. If the customer agrees, we can split "sell soft drinks" into two and delay one of them to the future:

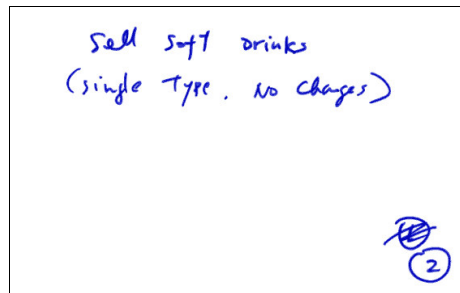
<i>User story</i>	<i>Story points</i>
Sell soft drinks (single type; no changes)	0.5
Return changes	1.5
Generate monthly sales report in text	2
Total	2.5

When will we implement "return changes"? It is just a regular story card. If the customer chooses it in the next iteration, we will do it in the next iteration. If he chooses it in the iteration after the next one, we will do it then. It is totally up to him.

For each user story selected for this iteration, unlike release planning where we only wanted just a few important details of a story, now we will ask the customer for all the details required for implementation. For example, for "sell soft drinks", we may draw some sketches on a white board showing the user interaction while providing him with suggestions:

This is the vending machine...	
The user inserts coins there...	
Suppose that he has inserted 50c and the price is 40c, all the buttons will be lit up. Remember that we only deal with a single type of soft drinks, so there is only one price...	
Then the user presses any button, a soft drink will be put into the dispenser...	
...	

After getting enough details, we may find that "sell soft drinks (single type; no changes)" actually should be worth 2 story points instead of 0.5. This is allowed. Then, we will update the story card:



and let the customer take out some cards for this iteration so that the total is ≤ 2.5 . If it is the opposite, i.e., we find that it is only worth less than 0.5 story points, then we will update the story card and let the customer add more cards for this iteration.

The process above of choosing what stories to do in the upcoming iteration is called "iteration planning".

User story is the beginning of communication, not the end of it

Suppose that after getting enough details from the customer, we are ready to start the implementation. Note that we don't have to write down all the details provided by the customer. Why not? If in the future you have a question about this user story, and if the customer is standing in front of you, would you directly ask him or go find the user story description in the requirement specification? Of course you would ask the customer, because in most of the time this live customer can provide requirements that are more accurate, complete and updated than a piece of dead requirement specification. In particular, whenever you have implemented a user story, try to let him see it or actually test drive it, because the more he knows about the system being developed, the more accurate and complete the requirements he can provide in the future. What if you forget about the details in the future? You can ask him again.

Remember, user story is the beginning of communication, not the end of it.

Therefore, usually we don't write down the events in a user story and only write the name of the user story, e.g., writing "Sell soft drinks" is enough.

Going through the iteration

After the customer choosing the stories, in the coming two weeks, we will implement them one by one. For each one we will perform design, coding, testing and etc. After implementing each user story, we will demonstrate the system according to the user story to the customer to see if this is what he wants.

Within the two weeks, if we finish these user stories early, we will ask the customer for more user stories. On the other hand, if we can't finish all these user stories, we will also let the customer know our actual progress.

References

- <http://c2.com/cgi/wiki?UserStory>.
- <http://c2.com/cgi/wiki?UseCase>.
- <http://www.xprogramming.com/xpmag/whatisxp.htm>.

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

- Kent Beck, Martin Fowler, Planning Extreme Programming, Addison-Wesley, 2000.
- Kent Beck, Extreme Programming Explained, Addison-Wesley, 2000.
- Ron Jeffries, Ann Anderson, Chet Hendrickson, Extreme Programming Installed, Addison-Wesley, 2000.
- Martin Fowler, Kendall Scott, UML Distilled, Addison-Wesley, 1999.
- Alistair Cockburn, Writing Effective Use Cases, Addison-Wesley, 2000.
- Jim Highsmith, Agile Project Management: Creating Innovative Products, Pearson Education, 2004.
- Craig Larman, Agile and Iterative Development: A Manager's Guide, Addison-Wesley, 2003.
- Brooks, F.P., The Mythical Man Month: Essays on Software Engineering Anniversary Edition. Addison-Wesley, 1995.



Chapter exercises

Introduction

Some of the problems will test you on the topics in the previous chapters.

Problems

1. This application is about courses. A course has a course code, a title, a fee and a number of sessions. A session specifies a date, a starting hour and ending hour. You have also created the following classes:

```
class Course {
    String courseId;
    String title;
    int fee;
    Session sessions[];
}
class Session {
    Date date;
    int startHour;
    int endHour;
}
```

The DBA (database administrator) has created the tables below:

```
create table Courses (
    courseId varchar(20) primary key,
    title varchar(20) not null,
    fee int not null
);
create table Sessions (
    courseId varchar(20),
    sessionId int,
    sessionDate date not null,
    startHour int not null,
    endHour int not null,
    primary key(courseId, sessionId)
);
```

Your tasks are:

1. Create an interface for accessing the courses while hiding the database.
2. Create a class to implement that interface and show how to implement its method for adding a course to the database.
3. Implement its method for enumerating all courses whose total duration is greater

than a specified number of hours.

4. Identify which layers they belong to.
2. In the application above, some courses are now included in the so called "Training Scheme for the employed". For each such course, the students will get a certain reimbursement after completing the course successfully. The discount is a constant for each course but may vary from one course to another. You know it is good to keep the Course class slim, so you decide to create a new class. So you have written the code below:

```
interface TrainingSchemeForEmployedRegistry {
    boolean isInScheme(String courseId);
    int getDiscount(String courseId);
    void addToScheme(String courseId, int discount);
}
```

To store this information in the database, you ask the DBA. He says that it is most efficient to just add a discount field to the Courses table. If that field is NULL, it means the course is not in the scheme:

```
create table Courses (
    courseId varchar(20) primary key,
    title varchar(20) not null,
    fee int not null,
    discount int
);
```

Your tasks are:

1. Create a class to implement TrainingSchemeForEmployedRegistry and show how to implement its method for the isInScheme and addToScheme methods.
2. Determine if you need to modify your code done in the previous question. If so, where?
3. Determine what happens to its scheme status if a course is deleted?
3. This program below implements a game called "MasterMind". The game is played like this: the computer "comes up with" a secret code such as "RGBY" in which "R" means red, "G" means green, "B" means blue, "Y" means yellow, "P" means pink, "C" means cyan. The task of the player is to try to find out this secret code. Every turn the player can input a code also consisting of four colors such as "RBPY". In this case, he has got "R" correct because the secret code also contains "R" in the first position. The same is true for "Y". In contrast, the secret code contains "B", but the position is not correct. In response to the guess, the computer will tell the player that he has got two pegs in correct color and position (but will not tell him that they are "R" and "Y") and that he has got one peg in the correct color but incorrect position (but will not tell him that it's "B"). The player has at most 12 turns. If he can find out the secret code within 12 turns, he wins. Otherwise he loses.

Your tasks are:

1. Point out and remove the problems in the code below.
2. Divide your revised code into appropriate layers.

```
class MasterMind {
    String secret = "RGBY";
    static public void main(String args[]) {
        new MasterMind();
    }
    MasterMind() {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        for (i = 0; i < 12;) { //can guess at most 12 times
            String currentGuess = br.readLine();
            String currentSecret = secret;
            int m = 0; //how many pegs are right in color and position.
            int n = 0; //how many pegs are right in color but wrong position.
            //valid the colors and find those in right color and position.
            for (j = 0; j < currentGuess.length(); ) {
                //must sure each peg is one of: Red, Yellow, Pink,
                //Green, Blue or Cyan.
                if ("RYPGBC".indexOf(currentGuess.charAt(j))==-1) {
                    System.out.println("Invalid color!");
                    break;
                }
                if (currentGuess.charAt(j)== currentSecret.charAt(j)) {
                    //right color and position.
                    m++;
                    //delete the peg.
                    currentGuess=
                        currentGuess.substring(0, j)+
                        currentGuess.substring(j+1);
                    currentSecret=
                        currentSecret.substring(0, j)+
                        currentSecret.substring(j+1);
                } else {
                    j++;
                }
            }
            //see how many pegs are in right color but wrong position.
            for (j = 0; j < currentGuess.length(); ) {
                //is it in right color regardless of the position?
                k = currentSecret.indexOf(guess.charAt(j));
                if (k!=-1)
                    n++;
                //delete the peg.
                currentGuess=
                    currentGuess.substring(0, j)+
                    currentGuess.substring(j+1);
                currentSecret=
                    currentSecret.substring(0, k)+
                    currentSecret.substring(k+1);
            } else {
                j++;
            }
        }
    }
}
```

```
        System.out.println(m+" are right in color and position");
        System.out.println(n+" are right in color but wrong position");
        if (m==4) { //all found?
            System.out.println("You won!");
            return;
        }
        i++;
    }
    System.out.println("You lost!");
}
}
```

4. This is a web-based application concerned with food orders. A user can enter an order id in a form and click submit. Then the application will display the customer name of the order and the order items (food id and quantity) in the order. The servlet doing that is shown below.

```
public class ShowOrderServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><TITLE>Order details</TITLE><BODY>");
        String orderId = request.getParameter("orderId");
        String customerId;
        Connection dbConn = ...;
        PreparedStatement st =
            dbConn.prepareStatement("select * from Orders where orderId=?");
        try {
            st.setString(1, orderId);
            ResultSet rs = st.executeQuery();
            try {
                if (!rs.next()) {
                    out.println("Error: Order not found!");
                    return;
                }
                customerId = rs.getString("customerId");
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
        //find and show the customer's name from his id.
        PreparedStatement st = dbConn.prepareStatement(
            "select * from Customers where customerId=?");
        try {
            st.setString(1, customerId);
            ResultSet rs = st.executeQuery();
            try {
                if (!rs.next()) {
                    out.println("Error: Customer not found!");
                    return;
                }
            }
            out.println("Customer: "+rs.getString("name"));
        } finally {
            rs.close();
        }
    }
}
```

```
    } finally {
        st.close();
    }
    //find and show the order items.
    st = dbConn.prepareStatement(
        "select * from OrderItems where orderId=?");
    try {
        st.setString(1, orderId);
        ResultSet rs = st.executeQuery();
        try {
            out.println("<TABLE>");
            while (rs.next()) {
                out.println("<TR>");
                out.println("<TD>");
                out.println(rs.getString(3)); //food id
                out.println("</TD>");
                out.println("<TD>");
                out.println(rs.getInt(4)+""); //quantity
                out.println("</TD>");
                out.println("</TD>");
                out.println("</TR>");
            }
            out.println("</TABLE>");
        } finally {
            rs.close();
        }
    } finally {
        st.close();
    }
    out.println("</BODY></HTML>");
}
```

Your tasks are:

1. Point out and remove the problems in the code below.
2. Divide your revised code into appropriate layers.
5. Come up with 10 user stories and estimate the story points of each. Assume that you have 2 developers and that you estimate one story point takes one developer 4 days to do, estimate how many story points can be done by the team in the first iteration. If you were the customer, which user stories would you like to be implemented in the first iteration? Assume that in the first iteration the team finished just 90% of the story points planned. What is the current velocity? According to this velocity, estimate the project duration. If the customer insists that you deliver the system by 80% of the duration you propose, what should you do? How many story points can be done in the second iteration?

Sample solutions

1. This application is about courses. A course has a course code, a title, a fee and a number of sessions. A session specifies a date, a starting hour and ending hour. You have also created the following classes:

```
class Course {
    String courseId;
    String title;
    int fee;
    Session sessions[];
}
class Session {
    Date date;
    int startHour;
    int endHour;
}
```

The DBA (database administrator) has created the tables below:

```
create table Courses (
    courseId varchar(20) primary key,
    title varchar(20) not null,
    fee int not null
);
create table Sessions (
    courseId varchar(20),
    sessionId int,
    sessionDate date not null,
    startHour int not null,
    endHour int not null,
    primary key(courseId, sessionId)
);
```

Your tasks are:

1. Create an interface for accessing the courses while hiding the database.

```
interface Courses {
    void addCourse(Course course);
    void deleteCourse(String courseId);
    void updateCourse(Course course);
    CourseIterator getAllCoursesById();
}
```

2. Create a class to implement that interface and show how to implement its method for adding a course to the database.

```
class CoursesInDB implements Courses {
    void addCourse(Course course) {
        PreparedStatement st =
            dbConn.prepareStatement("insert into from Courses values(?, ?, ?)");
        try {
            st.setString(1, course.getId());
            st.setString(2, course.getTitle());
```

```

        st.setInt(3, course.getFee());
        st.executeUpdate();
    } finally {
        st.close();
    }
}
st = dbConn.prepareStatement(
    "insert into from Sessions values(?,?,?,?)");
try {
    for (int i = 0; i < course.getNoSessions(); i++) {
        Session session = course.getSessionAt(i);
        st.setString(1, course.getId());
        st.setInt(2, i);
        st.setDate(3, session.getDate());
        st.setInt(4, session.getStartHour());
        st.setInt(5, session.getEndHour());
        st.executeUpdate();
    }
} finally {
    st.close();
}
}
}
}

```

3. Implement its method for enumerating all courses whose total duration is greater than a specified number of hours.

Update the interface first:

```

interface Courses {
    ...
    CourseIterator getCoursesLongerThan(int duration);
}

```

You may implement the selection using SQL. It is more efficient because only those courses meeting the condition are transferred. But it is useless for another storage mechanism other than a DB.

```

class CoursesInDB {
    ...
    CourseIterator getCoursesLongerThan(int duration) {
        PreparedStatement st =
            dbConn.prepareStatement(
                "select * from Courses where "+
                "(select sum(endHour-startHour) from Sessions "+
                "where Courses.courseId=Sessions.courseId)>? "+
                "order by courseId");
        st.setInt(1, duration);
        return new CourseResultSetIterator(st);
    }
}

class CourseResultSetIterator implements CourseIterator {
    PreparedStatement stToLoadCourses;
    ResultSet rsOfCourses;
    PreparedStatement stToLoadSessions;
    CourseResultSetIterator(
        Connection dbConn, PreparedStatement stToLoadCourses) {
        this.stToLoadCourses = stToLoadCourses;
    }
}

```

```

    this.rsOfCourses = stToLoadCourses.executeQuery();
    this.stToLoadSessions = dbConn.prepareStatement(
        "select * from Sessions where courseId=? order by sessionId");
}
boolean next() {
    return rsOfCourses.next();
}
Course getCourse() {
    Course course = new Course(
        rsOfCourses.getString(1),
        rsOfCourses.getString(2),
        rsOfCourses.getInt(3));
    loadSessionsFromDB(course);
    return course;
}
void loadSessionsFromDB(Course course) {
    stToLoadSessions.setString(1, course.getId());
    ResultSet rsOfSessions=stToLoadSessions.executeQuery();
    try {
        while (rsOfSessions.next()) {
            Session session = new Session(
                rsOfSessions.getDate(3),
                rsOfSessions.getInt(4),
                rsOfSessions.getInt(5));
            course.addSession(session);
        }
    } finally {
        rsOfSessions.close();
    }
}
void finalize() {
    stToLoadSessions.close();
    rsOfCourses.close();
    stToLoadCourses.close();
}
}

```

You may also implement the selection using domain logic in Courses. Courses will become an abstract class instead of an interface. This method is less efficient because all the courses are transferred. But it can be used with any other storage mechanisms.

```

class Course {
    ...
    int getDuration() {
        int duration = 0;
        for (int i = 0; i < getNoSessions(); i++) {
            duration += getSessionAt(i).getDuration();
        }
        return duration;
    }
}
class Session {
    ...
    int getDuration() {
        return endHour-startHour;
    }
}
abstract class Courses {

```

```

...
CourseIterator getCoursesLongerThan(final int duration) {
    final CourseIterator allCourses = getAllCoursesById();
    return new CourseIterator() {
        Course currentCourse;
        boolean next() {
            while (allCourses.next()) {
                currentCourse = allCourses.getCourse();
                if (currentCourse.getDuration() > duration) {
                    return true;
                }
            }
            return false;
        }
        Course getCourse() {
            return currentCourse;
        }
    };
}
}

```

4. Identify which layers they belong to.

Domain: Courses, Course, Session, CourseIterator.

DB access: CoursesInDB, CourseResultSetIterator.

2. In the application above, some courses are now included in the so called "Training Scheme for the employed". For each such course, the students will get a certain reimbursement after completing the course successfully. The discount is a constant for each course but may vary from one course to another. You know it is good to keep the Course class slim, so you decide to create a new class. So you have written the code below:

```

interface TrainingSchemeForEmployedRegistry {
    boolean isInScheme(String courseId);
    int getDiscount(String courseId);
    void addToScheme(String courseId, int discount);
}

```

To store this information in the database, you ask the DBA. He says that it is most efficient to just add a discount field to the Courses table. If that field is NULL, it means the course is not in the scheme:

```

create table Courses (
    courseId varchar(20) primary key,
    title varchar(20) not null,
    fee int not null,
    discount int
);

```

Your tasks are:

1. Create a class to implement TrainingSchemeForEmployedRegistry and show how to

implement its method for the `isInScheme` and `addToScheme` methods.

```
class TrainingSchemeForEmployedRegistryInDB
implements TrainingSchemeForEmployedRegistry {
    boolean isInScheme(String courseId) {
        PreparedStatement st =
            dbConn.prepareStatement(
                "select discount from Courses where courseId=?");
        try {
            st.setString(1, courseId);
            ResultSet rs=st.executeQuery();
            rs.next();
            return rs.getObject(1)!=null;
        } finally {
            st.close();
        }
    }
    void addToScheme(String courseId, int discount) {
        PreparedStatement st =
            dbConn.prepareStatement(
                "update Courses set discount=? where courseId=?");
        try {
            st.setInt(1, discount);
            st.setString(2, courseId);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    ...
}
```

2. Determine if you need to modify your code done in the previous question. If so, where?

When adding an course record to the `Courses` table, there is one more field (`discount`). It must be `NULL` to indicate that the course is not in the scheme. Some DBMS may allow you to use the previous code unchanged, while still having this effect. But to be safe than sorry, do it explicitly:

```
class CoursesInDB implements Courses {
    void addCourse(Course course) {
        PreparedStatement st =
            dbConn.prepareStatement(
                "insert into from Courses values(?,?,?,?)");
        try {
            st.setString(1, course.getId());
            st.setString(2, course.getTitle());
            st.setInt(3, course.getFee());
            st.setNull(4, java.sql.Types.INTEGER);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    ...
}
```

- Determine what happens to its scheme status if a course is deleted?

The scheme status is also deleted. This is what we want.

- This program below implements a game called "MasterMind". The game is played like this: the computer "comes up with" a secret code such as "RGBY" in which "R" means red, "G" means green, "B" means blue, "Y" means yellow, "P" means pink, "C" means cyan. The task of the player is to try to find out this secret code. Every turn the player can input a code also consisting of four colors such as "RBPY". In this case, he has got "R" correct because the secret code also contains "R" in the first position. The same is true for "Y". In contrast, the secret code contains "B", but the position is not correct. In response to the guess, the computer will tell the player that he has got two pegs in correct color and position (but will not tell him that they are "R" and "Y") and that he has got one peg in the correct color but incorrect position (but will not tell him that it's "B"). The player has at most 12 turns. If he can find out the secret code within 12 turns, he wins. Otherwise he loses.

Your tasks are:

- Point out and remove the problems in the code below.
- Divide your revised code into appropriate layers.

```
class MasterMind {
    String secret = "RGBY";
    static public void main(String args[]) {
        new MasterMind();
    }
    MasterMind() {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        for (int i = 0; i < 12;) { //can guess at most 12 times
            String currentGuess = br.readLine();
            String currentSecret = secret;
            int m = 0; //how many pegs are right in color and position.
            int n = 0; //how many pegs are right in color but wrong position.
            //valid the colors and find those in right color and position.
            for (int j = 0; j < currentGuess.length(); ) {
                //must sure each peg is one of: Red, Yellow, Pink,
                //Green, Blue or Cyan.
                if ("RYPGBC".indexOf(currentGuess.charAt(j))==-1) {
                    System.out.println("Invalid color!");
                    break;
                }
            }
            if (currentGuess.charAt(j)== currentSecret.charAt(j)) {
                //right color and position.
                m++;
                //delete the peg.
                currentGuess=
                    currentGuess.substring(0, j)+
                    currentGuess.substring(j+1);
                currentSecret=
                    currentSecret.substring(0, j)+
                    currentSecret.substring(j+1);
            }
        }
    }
}
```

```

        } else {
            j++;
        }
    }
    //see how many pegs are in right color but wrong position.
    for (int j = 0; j < currentGuess.length(); ) {
        //is it in right color regardless of the position?
        int k = currentSecret.indexOf(currentGuess.charAt(j));
        if (k!=-1) {
            n++;
            //delete the peg.
            currentGuess=
                currentGuess.substring(0, j)+
                currentGuess.substring(j+1);
            currentSecret=
                currentSecret.substring(0, k)+
                currentSecret.substring(k+1);
        } else {
            j++;
        }
    }
    System.out.println(m+" are right in color and position");
    System.out.println(n+" are right in color but wrong position");
    if (m==4) { //all found?
        System.out.println("You won!");
        return;
    }
    i++;
}
System.out.println("You lost!");
}
}

```

The code mixes model logic with UI. Some code is duplicated. The comments in the code can be removed by making the code as clear as the comments.

We can extract the domain logic into:

```

class MasterMind {
    static final int MAX_NO_GUESSES=12;
    static final int NO_PEGS=4;
    String secret = "RGBY";
    GuessResult makeGuess(String guess) throws InvalidColorException {
        GuessPartialResult partialResult=
            new GuessPartialResult(secret, guess);
        partialResult.validateColors();
        partialResult.findPegsInRightColorAndPos();
        partialResult.findPegsInRightColorIgnorePos();
        return partialResult.makeFinalResult();
    }
}
class InvalidColorException extends Exception {
}

```

GuessPartialResult represents the intermediate state when the pegs are being compared:

```
class GuessPartialResult {
    String secretLeft;
    String guessLeft;
    int noPegsInRightColorAndPos;
    int noPegsInRightColorButWrongPos;
    GuessPartialResult(String initSecret, String initGuess) {
        secretLeft=initSecret;
        guessLeft=initGuess;
    }
    void validateColors() {
        for (int j = 0; j < guessLeft.length(); j) {
            assertValidColor(guessLeft.charAt(j));
        }
    }
    void assertValidColor(char color) throws InvalidColorException {
        final String RED="R";
        final String YELLOW="Y";
        final String PINK="P";
        final String GREEN="G";
        final String BLUE="B";
        final String CYAN="C";
        final String VALID_COLORS=RED+YELLOW+PINK+GREEN+BLUE+CYAN;
        if (VALID_COLORS.indexOf(color)==-1) {
            throw new InvalidColorException();
        }
    }
    void findPegsInRightColorAndPos() {
        for (int j = 0; j < guessLeft.length(); j) {
            if (isPegInRightColorAndPos(j)) {
                noPegsInRightColorAndPos++;
                guessLeft=deletePegAt(guessLeft, j);
                secretLeft=deletePegAt(secretLeft, j);
            } else {
                j++;
            }
        }
    }
    void findPegsInRightColorIgnorePos() {
        for (int j = 0; j < guessLeft.length(); j) {
            int k = findFirstPegInSecretOfColor(guessLeft.charAt(j));
            if (k!=-1) {
                noPegsInRightColorButWrongPos++;
                guessLeft=deletePegAt(guessLeft, j);
                secretLeft=deletePegAt(secretLeft, k);
            } else {
                j++;
            }
        }
    }
    boolean isPegInRightColorAndPos(int idx) {
        return guessLeft.charAt(idx)== secretLeft.charAt(idx);
    }
    int findFirstPegInSecretOfColor(char color) {
        return secretLeft.indexOf(color);
    }
    String deletePegAt(String pegs, int idx) {
        return pegs.substring(0, idx)+pegs.substring(idx+1);
    }
    GuessResult makeFinalResult() {
        return new GuessResult(
```

```

        noPegsInRightColorAndPos,
        noPegsInRightColorButWrongPos);
    }
}

```

GuessResult represents the final result of a guess:

```

class GuessResult {
    int noPegsInRightColorAndPos;
    int noPegsInRightColorButWrongPos;
    boolean allFound() {
        return noPegsInRightColorAndPos==MasterMind.NO_PEGS;
    }
}

```

The UI will use the domain logic:

```

class MasterMindApp {
    static public void main(String args[]) {
        new MasterMindApp();
    }
    MasterMindApp() {
        MasterMind masterMind = new MasterMind();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String guess = br.readLine();
        for (int i = 0; i < MasterMind.MAX_NO_GUESSES;) {
            try {
                GuessResult guessResult = masterMind.makeGuess(guess);
                System.out.println(
                    guessResult.getNoPegsInRightColorAndPos() +
                    " are right in color and position");
                System.out.println(
                    guessResult.getNoPegsInRightColorButWrongPos() +
                    " are right in color but wrong position");
                if (guessResult.allFound()) {
                    System.out.println("You won!");
                    return;
                }
            } catch (InvalidColorException e) {
                System.out.println("Invalid color!");
                break;
            }
            i++;
        }
        System.out.println("You lost!");
    }
}

```

The layers are:

- Domain: MasterMind, GuessPartialResult, GuessResult, InvalidColorException.
- UI: MasterMindApp.

4. This is a web-based application concerned with food orders. A user can enter an order id in

a form and click submit. Then the application will display the customer name of the order and the order items (food id and quantity) in the order. The servlet doing that is shown below.

```
public class ShowOrderServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><TITLE>Order details</TITLE><BODY>");
        String orderId = request.getParameter("orderId");
        String customerId;
        Connection dbConn = ...;
        PreparedStatement st =
            dbConn.prepareStatement("select * from Orders where orderId=?");
        try {
            st.setString(1, orderId);
            ResultSet rs = st.executeQuery();
            try {
                if (!rs.next()) {
                    out.println("Error: Order not found!");
                    return;
                }
                customerId = rs.getString("customerId");
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
        //find and show the customer's name from his id.
        st = dbConn.prepareStatement(
            "select * from Customers where customerId=?");
        try {
            st.setString(1, customerId);
            ResultSet rs = st.executeQuery();
            try {
                if (!rs.next()) {
                    out.println("Error: Customer not found!");
                    return;
                }
                out.println("Customer: "+rs.getString("name"));
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
        //find and show the order items.
        st = dbConn.prepareStatement(
            "select * from OrderItems where orderId=?");
        try {
            st.setString(1, orderId);
            ResultSet rs = st.executeQuery();
            try {
                out.println("<TABLE>");
                while (rs.next()) {
                    out.println("<TR>");
                    out.println("<TD>");
```

```
        out.println(rs.getString(3)); //food id
        out.println("</TD>");
        out.println("<TD>");
        out.println(rs.getInt(4)+""); //quantity
        out.println("</TD>");
        out.println("</TD>");
        out.println("</TR>");
    }
    out.println("</TABLE>");
} finally {
    rs.close();
}
} finally {
    st.close();
}
out.println("</BODY></HTML>");
}
```

Your tasks are:

1. Point out and remove the problems in the code below.
2. Divide your revised code into appropriate layers.

The code mixes model logic with database and UI. Remember that servlet is UI. We can extract the domain code into:

```
class Order {
    ...
    Customer customer;
    OrderItem orderItems[];
}
class OrderItem {
    ...
    String foodId;
    int quantity;
}
class Customer {
    ...
}
interface Orders {
    ...
    Order getOrder(String orderId);
}
interface Customers {
    ...
    Customer getCustomer(String customerId);
}
```

We can extract the database access code into:

```
class OrdersInDB implements Orders {
    Order getOrder(String orderId) {
```

```
PreparedStatement st =
    dbConn.prepareStatement("select * from Orders where orderId=?");
try {
    st.setString(1, orderId);
    ResultSet rs = st.executeQuery();
    try {
        if (!rs.next()) {
            throw new OrdersException();
        }
        String customerId = rs.getString("customerId");
        CustomersInDB customers = new CustomersInDB();
        Customer customer = customers.getCustomer(customerId);
        Order order = new Order(..., customer, ...);
        getOrderItemsFromDB(order);
        return order;
    } finally {
        rs.close();
    }
} finally {
    st.close();
}
}

void getOrderItemsFromDB(Order order) {
    PreparedStatement st = dbConn.prepareStatement(
        "select * from OrderItems where orderId=?");
    try {
        st.setString(1, order.getId());
        ResultSet rs = st.executeQuery();
        try {
            while (rs.next()) {
                order.addOrderItem(new OrderItem(
                    ...,
                    rs.getString("foodId"),
                    rs.getInt("quantity"),
                    ...));
            }
        } finally {
            rs.close();
        }
    } finally {
        st.close();
    }
}
}

class CustomersInDB implements Customers {
    Customer getCustomer(String customerId) {
        PreparedStatement st = dbConn.prepareStatement(
            "select * from Customers where customerId=?");
        try {
            st.setString(1, customerId);
            ResultSet rs = st.executeQuery();
            try {
                if (!rs.next()) {
                    throw new CustomersException();
                }
            }
            return new Customer(..., rs.getString("name"), ...);
        } finally {
            rs.close();
        }
    }
}
```

```

    }
    } finally {
        st.close();
    }
}
}

```

The UI will use the domain logic (but not the DB) as:

```

public class ShowOrderServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Orders orders =
            (Orders)getServletContext().getAttribute("Orders");
        out.println("<HTML><TITLE>Order details</TITLE><BODY>");
        String orderId = request.getParameter("orderId");
        try {
            Order order = orders.getOrder(orderId);
        } catch (OrdersException e) {
            out.println("Error: Order not found!");
            return;
        } catch (CustomersException e) {
            out.println("Error: Customer not found!");
            return;
        }
        out.println("Customer: "+order.getCustomer().getName());
        showOrderItems(out, order);
        out.println("</BODY></HTML>");
    }
    void showOrderItems(PrintWriter out, Order order) {
        out.println("<TABLE>");
        for (int i = 0; i < order.getNoOrderItems(); i++) {
            out.println("<TR>");
            out.println("<TD>");
            out.println(order.getOrderItemAt(i).getFoodId());
            out.println("</TD>");
            out.println("<TD>");
            out.println(order.getOrderItemAt(i).getQuantity()+"");
            out.println("</TD>");
            out.println("</TD>");
            out.println("</TR>");
        }
        out.println("</TABLE>");
    }
}

```

The layers are:

- Domain: Order, OrderItem, Orders, Customer, Customers, OrdersException, CustomersException.
- DB access: OrdersInDB, CustomersInDB.

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

- UI: ShowOrderServlet.
5. Come up with 10 user stories and estimate the story points of each. Assume that you have 2 developers and that you estimate one story point takes one developer 4 days to do, estimate how many story points can be done by the team in the first iteration. If you were the customer, which user stories would you like to be implemented in the first iteration? Assume that in the first iteration the team finished just 90% of the story points planned. What is the current velocity? According to this velocity, estimate the project duration. If the customer insists that you deliver the system by 80% of the duration you propose, what should you do? How many story points can be done in the second iteration?

Assume the 10 user stories are:

Name	Story points
C1	2
C2	3
C3	2
C4	1
C5	4
C6	5
C7	3
C8	2
C9	4
C10	1

The total is 25 story points. As one story point needs 4 days and we have two developers, the team should be able to do $10/4 \times 2 = 5$ story points in the first iteration.

If we were the customer, we would pick C1, C3 and C4. The total is 5 story points (≤ 5).

If in the first iteration we only did 90% of the story points, i.e., 4.5 points. So the current velocity is 4.5 points. According to this velocity, as there are 25 story points, we will need $25/4.5 = 5.6$ iterations. We cannot have a partial iteration, so we will count it as 6 iterations. So the project duration is $6 \times 10 = 60$ days.

If the customer needs to get the system in just 80% of the proposed duration, i.e., 48 days, we may let him choose which user stories to include as long as the total does not exceed $(48/10) \times 4.5 = 21.6$ story points. It means he needs to take out at least 3.4 story points. For example, he may take out C5 (4 points); he may take out C9 (4 points); he

may take out C7 and C10 (3+1 points). It is totally up to him.

Should he insist to include all the user stories without reducing any functionality (this is extremely rare!), then we need "n" developers such that $(48/10) \times 4.5 \times (n/2) = 25$, i.e., "n" should be 2.3. As we cannot have 0.3 of a developer, we need to hire one additional developer and the customer will have to pay for the salary.

For the second iteration, the customer can only choose user stories not exceeding 4.5 story points.