



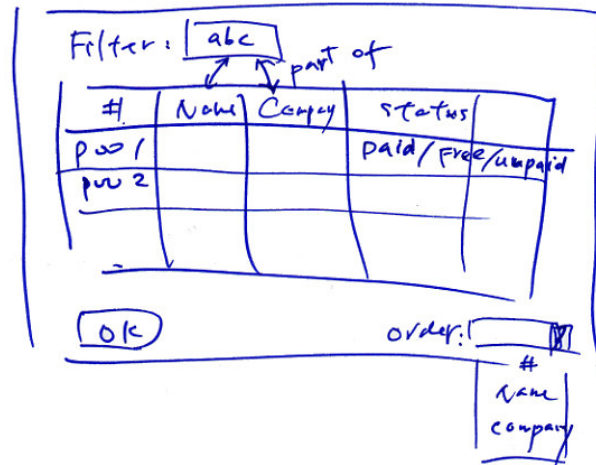
CHAPTER 15

Essential Skills for Communications

Different ways to communicate requirements

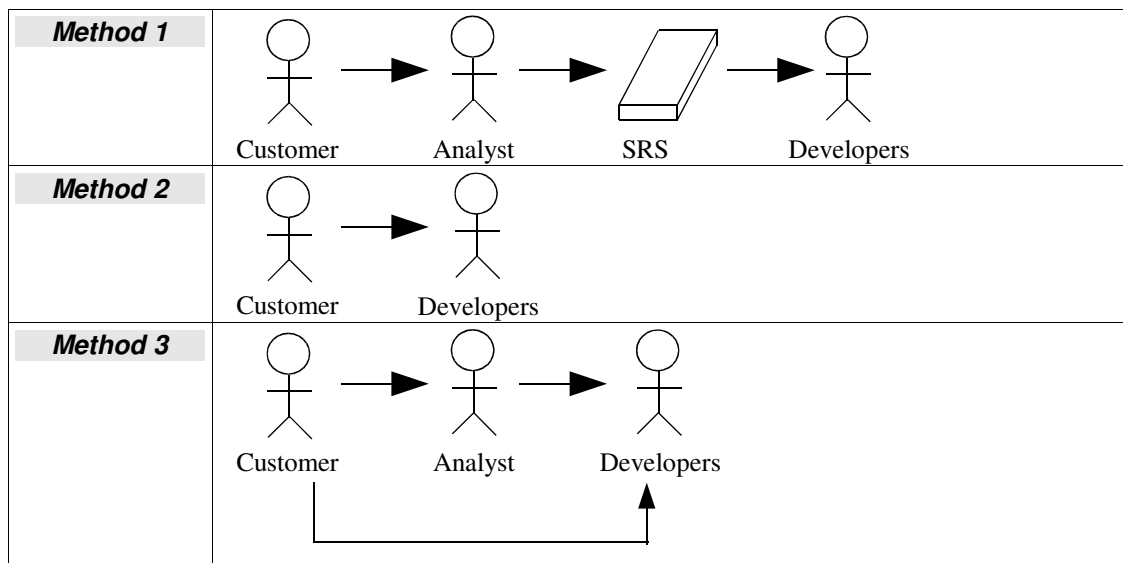
Suppose that we are developing an application for a customer. To do that, we must understand what the requirements are, e.g., what functions it should have or how it can be used to say input some particular data? There are different possible ways to communicate these requirements. For example:

1. We may have a system analyst talk to the customer to understand their requirements. Then the analyst writes down the requirements in a software requirements specification (SRS). The other developers will read the SRS to understand the requirements. If they don't understand something, they will fill out a requirement clarification request and send it to the system analyst. The system analyst may in turn email the customer for answers.
2. All the developers may talk to the customer directly. The requirements are then coded as automated acceptance tests. The customer must be always in the same room as the developers. If the developers have questions, they ask the customer face to face immediately.
3. A system analyst may talk to the customer to have the major requirements. Then he writes a brief description of each feature on a notepad and brief the developers. For the details, all the developers will talk to the customer directly in regular weekly meetings. As the customer doesn't stay with the developers, between meetings they communicate by phone as required. The important information will be added to a flip chart (an example is shown below). The requirements are also coded as automated acceptance tests but with textual explanations.



Which method above is the best way to communicate? It really depends the current situation. Method 1 is commonly called a "heavy" method because much of the communication must be done in writing and probably in a fixed format/template (e.g., SRS and requirement clarification request). It has some drawbacks:

First, the communication path between the customer and the developers is very long (i.e., very indirect) when compared to the other two methods (see below). The more indirect the path is, the more time is needed to communicate and the more errors will creep in.



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

Second, it takes more time for the analyst to write the SRS and for the other developers to read the SRS than just letting them talk to one another.

Third, no analyst in the world can express the requirements that he knows about 100% completely and correctly through writing. Talking can't either but talking allows easy bi-directional communication, i.e., if the other developers don't understand something or find something ambiguous, they can ask the analyst immediately.

	<i>Support for bi-directional communication</i>
<i>Method 1</i>	Hard (must be done in writing).
<i>Method 2</i>	Easy (talk in the same room).
<i>Method 3</i>	Not as easy as method 2 but easier than method 1 (talk in weekly meetings or on the phone).

So it seems that method 2 (a "light" method) must be better than method 1? If the customer is in US but the developers are in India, then it is not going to work. If there are ten or less developers on the team, if the customer has some important thing to tell, he can simply tell it to all of them ("Hey, guys, listen up! I have something important to tell..."). But if there are more than ten developers (e.g., 20 or 100), they simply can't fit in a room. Even if they can (e.g., use a stadium), the customer simply can't tell something to all of them easily.

Method 2 is probably not the best for someone trying to understand the requirements in the future. If the developers are still there readily accessible, he can just ask them. However, if they are all gone, that person will need to go through the acceptance tests to understand the requirements. This should work, but some well written textual explanations would make it a lot easier for him. Let's compare:

<i>Acceptance test without textual explanations</i>	<i>Acceptance test with textual explanations</i>
<pre>SystemInit AddParticipant,p001,Paul AddParticipant,p002,John AddParticipant,p003,Mary AddSeminar,s001,How to go agile?,1000,2 EnrolSeminar,s001,p001 EnrolSeminar,s001,p002 ExpectError,EnrolSeminar,s001,p003</pre>	<pre>//Make sure no enrollment is //accepted if a seminar is //already full. SystemInit AddParticipant,p001,Paul AddParticipant,p002,John AddParticipant,p003,Mary //Seminar has only 2 seats. AddSeminar,s001,How to go agile?,1000,2 EnrolSeminar,s001,p001 EnrolSeminar,s001,p002 //The third enrollment. ExpectError,EnrolSeminar,s001,p003</pre>

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

This is because the acceptance test contains much irrelevant data. For example, what is relevant above is that the seminar has only 2 seats and that there are already 2 enrollments. The other data is simply irrelevant. Without some text explanations a reader won't know what data is relevant and what is not. So it takes more time and effort to understand the test. In contrast, the text explanations can point out the relevant data and thus can speed up the understanding process.

Method 3 is somewhere between the method 1 and 2. When compared to method 1, it requires less formality (e.g., use notepad and flip chart instead of an SRS) and encourages face to face (weekly meetings) or oral (phone) communication. When compared to method 2, weekly meetings and oral communication are not as good as constant face to face communication. However, it may make sense if the customer doesn't work in the same building as the developers. It also adds explanations to the acceptance tests. Must this be better? Not necessarily. Adding explanations takes time and effort. If you change the acceptance test, you must also update the explanations. So, you must weigh in the costs and benefits for your team and project.

There are method 4, 5, 6 and lots more subject to your imagination and validation. For example, if you can afford something better than weekly meeting, try changing to daily meeting. Or if it is the opposite, you can change it to bi-weekly meeting. Instead of phone, you may use NetMeeting or ICQ depending on your actual situation.

In summary:

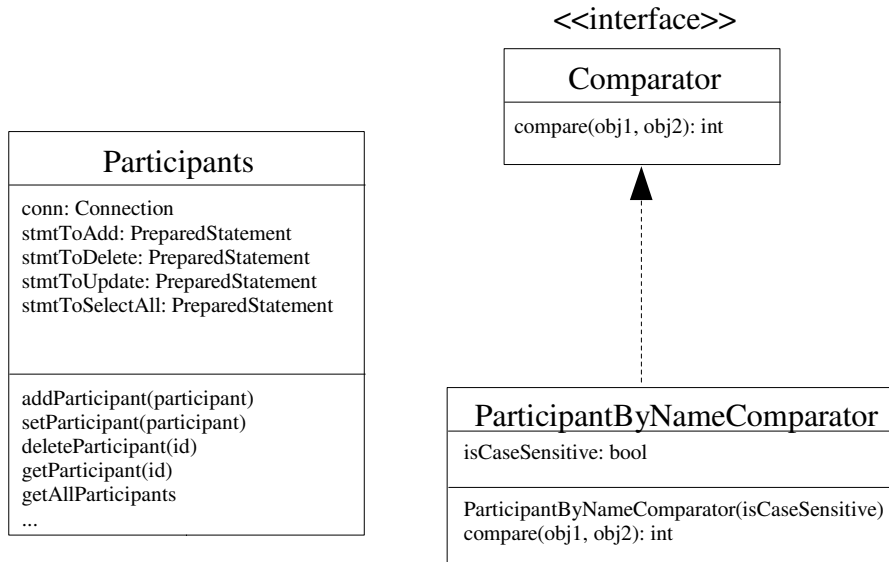
- Do not create documentation just because it feels right. Do it only for the purpose of communication. Even then, there are other ways to communicate and documentation may not be the best way.
- Face to face communication (with a white board or flip chart) is much more effective than documentation, unless the team is quite large or the participants can't come together (not in the same place or not at the same time).
- Keep the communication as frequent as possible.
- Keep the communication path as short as possible.
- Communicate relevant information only.

How to communicate designs

During the development, the developers need to communicate the designs of the system. Just like communicating requirements, there are different ways to communicate designs. For example, regarding the design to support the user story of listing the conference participants in

a grid, there may be different ways to communicate it:

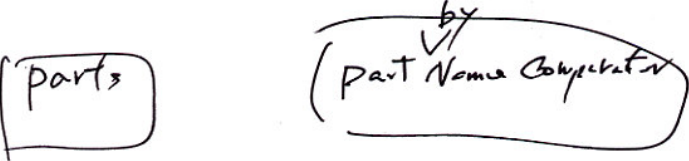
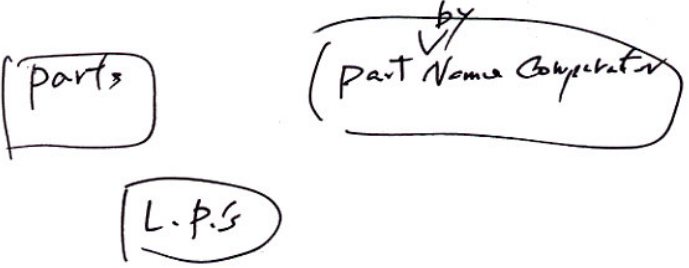
1. The developers can draw a detailed "UML class diagram" like:



In a UML class diagram, the symbols are well defined. For example, the dotted arrow shown above means "implement an interface".

2. The developers can make their code very easy to understand and make the design clear from the code. If they need to communicate the design to someone, they let him look at the code and explain what it does. They also show him the client code or the unit tests (just like another client) and explain how a client may use it.
3. The developers can talk about the design while drawing on the white board:

Narration	White board
<p>To list the participants, we need to read the participants from the database. This is done by the Participants class.</p>	<p>A hand-drawn sketch on a whiteboard showing a rounded rectangular box with the word "parts" written inside in a cursive, handwritten style.</p>

Narration	White board
We need to sort them according to their names. All we need is a Comparator. Let's call it ParticipantByNameComparator.	
Then we need a TableModel. Let's call it ListedParticipants and assume each row is a ListedParticipant (without "s" at the end).	

Note that what is said may not be drawn exactly on the white board. For example, the Participants class is simply written as Parts; the ListedParticipants class is simply written as L.P.'s. The ListedParticipant class is simply not drawn at all.

- The developers can use CRC cards. It is like method 3 but the cards can be moved around to simulate the collaboration.

Which method above is the best? Again, it depends on the situation.

Method 1 is the heaviest method above. It includes a lot of details. Some of the details such as the addParticipant method are irrelevant to the question at hand (how to list the participants). This makes it much harder to understand. It is not much different from just looking at the source code.

Method 2 is the lightest. Code plus oral explanations work very well. However, if it is the maintenance developer who is trying to understand the design, then he will have to get hold of one of those developers. If no one is available, then the oral explanation part will be missing and it will become much harder to understand the code alone. Therefore, if you can foresee who the maintenance developer is, then get him working on the team for the last several months so that the knowledge on the design is transferred.

Method 3 and 4 are also quite light. They are useful when the code is not yet written and thus method 2 cannot be applied. For example, when we need to discuss the design just before we implement the user story. In addition, if the target audience is a maintenance developer who

will never be known until after the release of the system, we can keep a copy of the drawings using a flip chart or an electronic white board, or even record the discussions with audio tapes and tape the CRC sessions. Then if the maintenance developer is trying understand how a user story is implemented, he can quickly lookup the relevant design information. However, if the team is large (more than ten people), this method no longer works well and we should find a middle ground with the formal UML class diagrams and the informal class drawings or CRC cards. For example, keep the formal meaning of the symbols, forbid the use of short form for the names and draw the relevant methods and attributes only.

References

- <http://www.xprogramming.com/xpmag/docIndex.htm>.
- <http://www.agilemodeling.com/essays/communication.htm>.
- <http://www.agilemodeling.com/essays/agileDocumentation.htm>.
- Alistair Cockburn, Agile Software Development, Addison-Wesley, 2001.
- Dean Leffingwell, Don Widrig, Managing Software Requirements: A Unified Approach, Addison-Wesley, 1999.
- Jim Highsmith, Agile Software Development Ecosystems, Addison-Wesley, 2002.
- Karl E. Wiegers, Software Requirements, Second Edition, Microsoft Press, 2003.
- Kent Beck, Extreme Programming Explained, Addison-Wesley, 2000.
- Martin Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition, Addison-Wesley, 2003.
- Scott W. Ambler, The Object Primer 3rd Edition: Agile Model Driven Development with UML 2, Cambridge University Press, 2004.