Acceptance Testing

• Acceptance tests represent the customer’s interests.
• The acceptance tests give the customer confidence that the application has the required features and that they behave correctly.
  • System Coverage Unit tests alone do not give the team all the confidence it needs.
  • The problem is, unit tests miss many bugs.
  • Functional tests fill in the gap.
  • Perfectly written unit tests may give you all the code coverage you need, but they don’t give you (necessarily) all the system coverage you need.
• The functional tests will expose problems that your unit tests are missing
• In theory when all the acceptance tests pass the project is done.
• Acceptance tests do three things for a software development team:
Acceptance Testing

- Acceptance tests represent the customer’s interests.
- The acceptance tests give the customer confidence that the application has the required features and that they behave correctly.
- In theory when all the acceptance tests pass the project is done.

Acceptance tests do three things for a software development team:

- They capture **user requirements** in a directly verifiable way, and they measure how well the system meets those requirements.
- They expose problems that unit tests miss.
- They provide a ready-made definition of how “done” the system is.
Acceptance vs. Unit Tests

- **Unit tests**
  - written in the same language as the code they test.
  - interact with the programming objects directly, often in the same name space.
  - written by the programmers.

- **Acceptance tests**
  - correspond to the programming objects in a much looser way.
  - A single acceptance test will almost always rely on many different programming objects.
  - a single programming object will often affect the outcome of multiple unrelated acceptance tests.
  - acceptance tests are frequently run through an external interface to the system (a bridge).
Acceptance vs. Unit Tests

<table>
<thead>
<tr>
<th>Acceptance Tests</th>
<th>Unit Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Written by Customer and Analyst.</td>
<td>Written by developers.</td>
</tr>
<tr>
<td>Written using an acceptance testing framework (also unit testing framework).</td>
<td>Written using a unit testing framework.</td>
</tr>
<tr>
<td>(extreme programming) When acceptance tests pass, stop coding. The job is done.</td>
<td>(extreme programming) When unit tests pass, write another test that fails.</td>
</tr>
<tr>
<td>The motivation of acceptance testing is demonstrating working functionalities.</td>
<td>The motivation of unit testing is finding faults.</td>
</tr>
<tr>
<td>Used to verify that the implementation is complete and correct. Used for Integration, System, and regression testing. Used to indicate the progress in the development phase. (Usually as %). Used as a contract. Used for documentation (high level)</td>
<td>Used to find faults in individual modules or units (individual programs, functions, procedures, web pages, menus, classes, …) of source code. Used for documentation (low level)</td>
</tr>
<tr>
<td><strong>Written before the development and executed after.</strong></td>
<td><strong>Written and executed during the development.</strong></td>
</tr>
<tr>
<td><strong>Starting point: User stories, User needs, Use Cases, Textual Requirements, …</strong></td>
<td><strong>Starting point: new capability (to add a new module/function or class/method).</strong></td>
</tr>
</tbody>
</table>
The Tester Contribution

- A tester combines quality assurance and development skills.
- A tester understands the:
  - Customer’s needs
  - Development needs.
  - User needs.
Acceptance Test Activities can help in clarifying requirements

- It is estimated that 85% of the defects in developed software originate in the requirements (communication between customer and analyst, communication between analyst and developer).
- There are several “sins” to avoid when specifying requirements
  - noise
  - silence
  - ambiguity
  - over-specification
  - wishful thinking
Tester activities

- Help **negotiate quality** with the customer.
- Help **clarify requirements** (stories).
- Help with **estimates** during planning.
- Advocate the **customer's rights**.
- Guard the **programmers' rights**.
- Work with the customer to **write effective and thorough acceptance tests**.
- Make sure acceptance tests (functional, load, stress, performance, compatibility, installation, security, anything not covered by unit and integration tests) **verify the quality specified by the customer**.
- Help **automate maintainable acceptance tests**.
- Make sure test **results are reported** in a timely manner, forming a continuous feedback loop.
- Make sure **acceptance testing keeps pace with development**.
- Help the programmers **design more testable code**.
Setting quality criteria

- Customers have realistic constraints (funding, timing). Therefore: *Customers do not always want the highest quality system.*
- The customer may define quality standards as a set of features.
- The customer asks for a certain level of quality
  - The programmers account for that in estimating the risk and effort of the requirements (stories).
- Many quality criteria can become requirements on their own.
  - For example, the customer could write a story saying the system should handle all exceptions and print out a user-friendly error message.
10 prerequisites that must be met

1. Business Requirements must be available

2. Application Code should be fully developed

3. Unit Testing, Integration Testing & System Testing should be completed

4. No Show stoppers, or High or Medium defects in the System Integration Test Phase

5. Only Cosmetic errors are acceptable before acceptance testing
10 prerequisites that must be met

6. Regression Testing should be completed with no major defects

7. All the reported defects should be fixed and tested

8. Traceability matrix for all testing should be completed

9. UAT Environment must be ready

10. Sign off mail or communication from System Testing Team that the system is ready for acceptance testing execution
Survey Results

How are Agile Teams Validating their own Work?

- Iteration demos: 79%
- Developer regression testing: 71%
- Developer TDD: 53%
- Acceptance TDD: 44%
- All-hands demos: 42%
- End-of-lifecycle testing: 41%
- Non-solo development: 39%
- Static-code analysis: 32%
- Parallel independent testing: 26%
- External reviews: 23%
- Dynamic code analysis: 21%

Source: 2010 How Agile Are You? Survey
www.ambysoft.com/surveys/
Copyright 2010 Scott W. Ambler
Survey Results

Primary Approach to Acceptance Testing

- Not doing it: 3%
- Test First: 9%
- End of lifecycle: 10%
- Independent parallel testing: 16%
- Performed by team during iteration: 38%
- Performed by stakeholders: 24%

© 2012 Scott W. Ambler www.ambysoft.com/surveys/
Goals of tester activities

- **Identifying and making explicit** hidden assumptions in the stories.
- **Defining acceptance tests for each** story.
- **Including time for acceptance tests in story estimates.**
- Enabling **accurate estimates of time and development** velocity.
Defining High-Level Acceptance Tests (1)

• Basic Acceptance Test Definitions
• Given a user story (like a short use-case, or an operation+contract):
  • Identify all the actions in the story.
  • For each action, write two tests in a table
    • For some data, supply inputs that should make the action succeed, and fill in whatever a successful result is for result.
    • For other data, supply inputs to make the action fail, and fill in what the response should be.
Defining High-Level Acceptance Tests (1)

- **Acceptance test template**

<table>
<thead>
<tr>
<th>Action</th>
<th>Data</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. action</td>
<td>somedata</td>
<td>Success: result</td>
</tr>
<tr>
<td>2</td>
<td>otherdata</td>
<td>Failure: response</td>
</tr>
</tbody>
</table>

- The input/output data selection should exploit knowledge of **boundary conditions** and **equivalence class partitioning**.
- Concentrate on **what** should be tested, not **how**.
Defining High-Level Acceptance Tests (2)

- Example 1:
  - **Customer**: a regional telephone company.
  - **Application**: A business-directory application.
  - **User**: A resident or business looking for **business information**.
  - **Story**: Create a screen that will allow the user to search on the Web for businesses by **category of business** and **location**.
  - Acceptance test for “Business Search” Story

<table>
<thead>
<tr>
<th>Action</th>
<th>Data</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Search</td>
<td>A category/location combination for which some businesses exist</td>
<td>Success: a list of each business within the category and location</td>
</tr>
<tr>
<td>2</td>
<td>A category/location combination for which no businesses exist</td>
<td>Failure: empty list</td>
</tr>
</tbody>
</table>
Example 2:

- **Customer**: A tracking system provider.
- **Application**: A tracking application named ProjectTrack.

*ProjectTrack* is a Web application that allows to maintain information about projects, such as their iterations, stories, tasks, and tests, in a central online repository available to all project stakeholders.
Defining High-Level Acceptance Tests (4)

- This project's tracker application serves several objectives:
  - Maintain information about a project
    - estimated and actual time to complete stories,
    - who owns which tasks
    - which stories are assigned to each iteration.
  - These data and metrics are available to all interested stakeholders online.
    - If an upper-level manager of the business wants to know how the current iteration is progressing, he can log into ProjectTrack and see.
  - Teams split across various locations can track tasks, stories, and iterations
  - Maintain historical information about each project.
    - For example, the developers can check whether they're getting better at accurately estimating stories.
Defining High-Level Acceptance Tests (5)

- **Example 2 (cont):**

- **User:** A project developer.

- **Story:** The user can provide an estimate for how long it will take to implement a story, prioritize the story, and assign the story to an iteration.

- **Acceptance tests for “story estimation update” story:**

<table>
<thead>
<tr>
<th>Action</th>
<th>Data</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update story with estimate, priority and Iteration</td>
<td>A story already updated to the repository; <strong>Estimate; Priority; Iteration Number</strong></td>
<td>Success: story is updated with appropriate information</td>
</tr>
</tbody>
</table>
Identifying Hidden Assumptions
Identifying Hidden Assumptions (1)

- Stories have loads of hidden assumptions.
- The ones to worry about are those that will cause problems.
- To identify the potential problems:
  A. Think about the system from various viewpoints:
     1. The *customer's viewpoint*.
     2. The *user viewpoint*.
     3. The *developer viewpoint*.
     4. The *tester viewpoint*: Identify mismatches between the customer and the developer, and between the user and the developer.
  B. Mismatches lead to *hidden assumptions*.
  C. Write *acceptance tests for the hidden assumptions*. 
Identifying Hidden Assumptions (2)

A. Think about the system from various viewpoints:

1. The customer's viewpoint:
   - How this story relates to your business?
   - What business problem is it solving?
   - How does it solve the problem?
   - How could the system implement the story and not solve the problem?
   - Are there alternate solutions?
   - Are there related problems?
Identifying Hidden Assumptions (3)

..Think about the system from various viewpoints:

2. The **user viewpoint**:
   - What would be the **worst thing** for you that could happen?
   - How about the **best thing**?
   - What would really **irritate** you?
   - In what ways can you, as a user, "**screw up**"?
   - How would you want the system to respond?
   - How often will you be involved in this story?
   - What are you likely to have just previously done?
   - What are you likely to do next?
Identifying Hidden Assumptions (4)

..Think about the system from various viewpoints:

3. The *developer viewpoint*.
   - Based on what's written in the story, what's the simplest and easiest way to implement it that could possibly work?

4. The *tester viewpoint: How likely is it that the simplest implementation will*
   - Solve the customer's business problem?
   - Solve the related problems?
   - Be the best solution?
   - Avoid the user's worst-case scenario and achieve the best case?
   - Integrate with the user's workflow?
   - …
B. **Mismatches lead to hidden assumptions**

Whenever the answer to a question in step 1.4 is "Not very likely:

- Imagine what additional information is needed to change the answer to "Likely."
- This gives rise to a hidden assumption.
- Trigger further questions to ask.

C. **Write acceptance tests for the hidden assumptions.**
Identifying Hidden Assumptions (6)

- *Example 1 – the business directory application:*

- *Story:* Create a screen that will allow the user to search on the Web for businesses by category of business and location.

A. Think about the system from various viewpoints:

1. The *customer's viewpoint:*
   - *How does this relate to my business?*
   - *The phone directory is crucial to the phone system* for both residential customers and business customers.
   - *Business listings and advertisements* in the directory generate revenue.
Identifying Hidden Assumptions (7)

- **Example 1 – the business directory application:**
  
  A. Think about the system from various viewpoints:

1. The **customer's viewpoint:**
   
   - What business problem is it solving? How does it solve it?
   - It's expensive to print and distribute a hard-copy directory, so I don't do it very often. This means new businesses can't get in for a long time, and I can't charge them.
   
   - Plus users of the directory can't find them, so they call a directory operator, which is an **expensive service** to maintain.
   
   - The Web directory can be updated cheaply, so I can afford to do it daily.
   
   - This allows me to add new listings and realize the revenue right away as well as reducing directory-operator staffing and costs.
Example 1 – the business directory application:

A. Think about the system from various viewpoints:

1. The customer's viewpoint:
   - How could the system implement this and not solve the problem?
   - If people don't use the system, the problem isn't solved. Businesses won't pay if they still have to wait for the hard copy, and I won't be able to reduce the number of operators.
   - Are there alternate solutions? Are there related problems?
   - Can't think of any alternate solutions.
   - Finding residential listings is a similar problem, but I can't get any revenue from that.
   - If it works out as expected with the business listings, may try residential next.
Identifying Hidden Assumptions (9)

• **Example 1 – the business directory application:**
  A. Think about the system from various viewpoints:
  2. The **user's viewpoint:**
     • What are the worst and best things that can happen? What would really irritate me?
     • **Worst thing:** I find a business and drive to its location, but it's been out of business for a month
     • **Best thing:** I find the business, click a few times, and accomplish my goal (purchase/order/complaint) online.
     • **Irritating:** I'm looking for a particular business and know the name (approximately), but I can't just search on the name. I have to figure out how some idiot categorized it and search on that, then pick it out of a hundred others. (I get irritated just thinking about it!)
Identifying Hidden Assumptions (10)

• Example 1 – the business directory application:
  A. Think about the system from various viewpoints:
  2. The user's viewpoint:
  • How can I screw up? How should the system respond?
  • **Misspell something, like the category.** The system should correct it for me or at least offer some correction alternatives.
  • How often will I do this? What will I do before and after?
  • **Not very often. A couple of times a week at most—that's how often I use the hard-copy phone directory.**
  • What I've done before will usually be completely unrelated.
  • **Next thing** is probably to call the business (if I found one) or get out the hard copy (if I didn't).
Identifying Hidden Assumptions (11)

- **Example 1 – the business directory application:**
  
  A. Think about the system from various viewpoints:

  3. The *developer's viewpoint*:

     - What's the simplest implementation that could possibly work?
     - A form with text input fields for category and location;
     - A Search button in the browser.
     - A server program gets all the businesses that match the category and location from the database and sends a list in response.
     - Each item in the list has the business name, address, and phone number.
Identifying Hidden Assumptions (12)

- **Example 1 – the business directory application:**
  
  A. Think about the system from various viewpoints:
  
  4. The *tester's viewpoint – look for mismatches:*

  - How likely is the implementation to solve the business problem?
  - **Not very likely**, because solving the problem requires a significant percentage of customers to use the system.
  - **Peak load**…130 uses per second means about 200 searches and 1,000 hits per second. … This is a fairly high performance requirement that the simplest implementation is not likely to deliver.

  - How likely is the implementation to solve the related problems (residential listings)?
  - **Not very likely, for the same reason**, and even more so (more users, higher performance requirement).
Identifying Hidden Assumptions (13)

• **Example 1 – the business directory application:**
  A. Think about the system from various viewpoints:
  4. The *tester's viewpoint – look for mismatches:*
  
  • Respond appropriately to user mistakes. How likely is the implementation to avoid irritating the user?
  • **Not very likely,** because the simplest implementation has no search by name and no way to narrow the search or otherwise deal with a large number of returned results
  • The software has **no provision for a pull-down list;** users have to type in the category.
  • It **doesn't offer potential spelling** corrections if a category is Misspelled.
Identifying Hidden Assumptions (14)

- Example 1 – the business directory application:

B. Hidden assumptions:

- What information is the customer assuming we know?
  - That the **system will be able to handle a significant load**, about 200 searches and 1,000 hits per second at peak.
  - That **users should be able to search for a business by a number of things besides category**, like name and address, and can search within the results when a large number is returned.
  - That users should be able to browse and **select the category from a list**. If they **do type in a category** that isn't on the list, the system should say so and offer the closest match.
Identifying Hidden Assumptions (15)

- **Example 1 – the business directory application:**

B. **Hidden assumptions:** Further questions to ask:

- **Assumption 1:**
  - How many concurrent users do you expect?
  - What kind of response time is required?
  - What values for category searches and locations would produce tests representative of production use?

- **Assumption 2:**
  - Is this story just for the category searches, or can the user search for a business by name?
  - What do the search results look like?
  - What sort of error handling is desired?
Identifying Hidden Assumptions (16)

- *Example 1 – the business directory application:*

B. **Hidden assumptions:**

- Further questions to ask:
  - Assumption 3:
    - What fields should be on the screen? Which ones are required?
    - What happens if a given search produces no results?
    - What edits may be required for the fields?
Identifying Hidden Assumptions (17)

- Example 1 – the business directory application:

  C. Additional acceptance tests for “business search” story:

<table>
<thead>
<tr>
<th>Action</th>
<th>Data</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>.3</td>
<td>Enough concurrent users and category/location combinations to generate 200 searches and 1,000 hits/second</td>
<td>Success: each user gets appropriate results list within a reasonable response time</td>
</tr>
<tr>
<td>.4</td>
<td>Business name/location combo for which some businesses exist</td>
<td>Success: list of businesses that match the name at the specified location.</td>
</tr>
<tr>
<td>.5</td>
<td>Business address/location combo for which some businesses exist</td>
<td>Success: list of businesses that match the address at the specified location</td>
</tr>
<tr>
<td>.6</td>
<td>Search that retrieves more than 20 matching businesses</td>
<td>Success: list has a &quot;Search within these results&quot; option</td>
</tr>
<tr>
<td>.7</td>
<td>Misspelled category/location combo for which businesses exist</td>
<td>Failure: contains a &quot;Did you mean?&quot; type message and a list of categories that includes the correct category</td>
</tr>
</tbody>
</table>
Example 2 – The tracking system:

Story: The user can provide an estimate for how long it will take to implement a story, prioritize the story, and assign the story to an iteration.

A. Think about the system from various viewpoints:

1. The customer's viewpoint:

   • How does this relate to my project?
   • We need a way to track estimates for stories, to help with choosing stories for an iteration.
   • What business problem is it solving? How does it solve it?
   • Customers need to know how much a story costs before they can choose stories for an iteration.
Example 2 – The tracking system:

A. Think about the system from various viewpoints:

1. The customer's viewpoint:
   - How could the system implement this and not solve the problem?
   - Someone still has to input and maintain the online data. Some stories may not have estimates yet. That's why it has to be easy to update the priority, iteration, and estimate.
   - None of these fields can be required, because we'll have stories for which we don't know this information yet.
   - Are there alternate solutions? Are there related problems?
   - We could put the information on a whiteboard, story cards, or a spreadsheet, but that makes it harder to share the information with others who aren't in our location and has the same problem of needing to be kept current.
Example 2 – The tracking system:
A. Think about the system from various viewpoints:
   2. The user's viewpoint:
      • What are the worst and best things that can happen? What would really irritate me?
      • Worst thing: I can't locate the story or iteration I want. It's in the system, but I can't find it. I can't figure out how to assign a new story to an iteration. Two stories are similar, and I can't tell which one I want.
      • Best thing: I can see all the stories for an iteration and all the pertinent information about them online and select the one I want from a list.
      • Irritating: How, exactly, am I locating the story? Can I search by iteration? By name? By number?
Identifying Hidden Assumptions (21)

Example 2 – The tracking system:

A. Think about the system from various viewpoints:

2. The user's viewpoint:

• How can I screw up? How should the system respond?
  o Can I accidentally assign the same story to more than one iteration? Assign the same story twice to one iteration?

• Is it worth having the system prevent me from doing that.

• How often will I do this? What will I do before and after?
  o During release planning, we'll write the stories and estimate them, we'll choose the stories for that iteration and prioritize them.
Example 2 – The tracking system:

A. Think about the system from various viewpoints:

3. The developer's viewpoint:
- What's the simplest implementation that could possibly work?
- A list of all the stories, with columns for estimate, iteration, and priority.
- Click on the story to select it to update these fields.
- We need some way to identify the iteration, probably by number..
Identifying Hidden Assumptions (23)

Example 2 – The tracking system:

A. Think about the system from various viewpoints:

3. The tester's viewpoint (identifying mismatches):
   
   • How likely is the implementation to solve the business problem?
   • Fairly likely.
   
   • How likely is the implementation to solve the related problems?
   • We need to be able to unassign stories from an iteration, in case the customer changes his mind or we can’t complete all the stories.
   • How likely is the implementation to avoid irritating the user?
   • We aren't likely to have a lot of user mistakes.
Identifying Hidden Assumptions (24)

• **Example 2 – The tracking system:**

  B. **Hidden assumptions:**
  
  • The system provides **a list of stories not already assigned** to an iteration.
  • The system provides **a list of the stories assigned** to a particular iteration.
  • The system provides **a running total of estimates** for all stories in an iteration.
  • Estimate, priority, and iteration number **aren't required fields.**
  • The user can list stories and see the current values for estimate, priority, and iteration.
  • **The user can select a story for update** and update these fields at any time.
  • The system **will not validate for duplicate story names.** It's up to the user to make sure names are unique.
Identifying Hidden Assumptions (25)

- Example 2 – The tracking system:

C. Additional acceptance tests for “story estimation update” story:

<table>
<thead>
<tr>
<th>Action</th>
<th>Data</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2 update story</td>
<td>Existing story, no values entered in iteration, priority, or estimate</td>
<td>Success: these aren't required fields</td>
</tr>
<tr>
<td>3.3</td>
<td>Existing story, exceed maximum values for estimate, priority, and iteration</td>
<td>Failure: should not be able to enter a number that large</td>
</tr>
<tr>
<td>4. Add a new story with estimate, iteration, and priority</td>
<td>New story with valid values for estimate, iteration, and priority</td>
<td>Success: story is added with appropriate information</td>
</tr>
<tr>
<td>5. Search for stories not assigned to an iteration and select one to update</td>
<td>Search criteria and existing stories with nothing in the iteration field</td>
<td>Success: stories are listed with current values for estimate, priority, and iteration. When user clicks on story name, story fields appear on update screen and are modifiable.</td>
</tr>
<tr>
<td>6. Search for stories assigned to an iteration and select one to update</td>
<td>Search criteria and existing stories for a particular iteration</td>
<td>Success: stories are listed with current values for estimate, priority and iteration. When user clicks on story name, story fields appear on update screen with current values and are modifiable.</td>
</tr>
<tr>
<td>Requirement</td>
<td>Test</td>
<td>Test Case</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>Functional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U – Users reviewed
Q – QA reviewed
T – Ready for testing
**Traceability matrix**

<table>
<thead>
<tr>
<th>Requirement identifiers</th>
<th>Req Tested</th>
<th>UC 1.1</th>
<th>UC 1.2</th>
<th>UC 1.3</th>
<th>UC 2.1</th>
<th>UC 2.2</th>
<th>UC 2.3</th>
<th>UC 2.3.1</th>
<th>UC 2.3.2</th>
<th>UC 2.3.3</th>
<th>UC 3.1</th>
<th>UC 3.2</th>
<th>UC 3.3</th>
<th>TECH 1.1</th>
<th>TECH 1.2</th>
<th>TECH 1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Cases</td>
<td>321</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Tested Implicitly</td>
<td>77</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.1</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.2</td>
<td>2</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.3</td>
<td>2</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.4</td>
<td>1</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.5</td>
<td>2</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.1</td>
<td>2</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.2</td>
<td>2</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3.1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3.2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3.3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3.4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3.5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>etc..</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.6.2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
Ways to Estimate Acceptance-Test Effort
Ways to Estimate Acceptance-Test Effort

• For each acceptance test, estimate the time needed for the following:
  • **Preparation.**
    Defining, creating, and validating test data; designing coding; debugging automated tests
  • **Running tests.**
    Setting up, running, evaluating the outcome, and reporting the results
  • **Special considerations.**
    A limited test window, for example
      For instance, a 6,000-concurrent-user test may need access to a production database that's available only late at night or on weekends

• Add these three estimates to get the complete estimate for this test.
• Add the estimates for all the acceptance tests to get the total for the story.
## Estimated efforts

### Example 1 “business search” story

<table>
<thead>
<tr>
<th>Test</th>
<th>Preparation</th>
<th>Execution</th>
<th>Special</th>
<th>Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define details 0.5</td>
<td>Setup 0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create test records (DB) 0.5</td>
<td>Run 0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write tests 0.5</td>
<td>Evaluate 0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Make tests runnable 2.5</td>
<td>Report 0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>4.0</td>
<td>Total 0.4</td>
<td>Total 0.0</td>
<td>Total 4.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test</th>
<th>Preparation</th>
<th>Execution</th>
<th>Special</th>
<th>Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.0</td>
<td>0.4</td>
<td>0.0</td>
<td>4.4</td>
</tr>
<tr>
<td>2</td>
<td>4.0</td>
<td>0.4</td>
<td>0.0</td>
<td>4.4</td>
</tr>
<tr>
<td>3</td>
<td>12.0</td>
<td>4.5</td>
<td>8.5</td>
<td>25.0</td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
<td>0.4</td>
<td>0.0</td>
<td>4.4</td>
</tr>
<tr>
<td>5</td>
<td>4.0</td>
<td>0.4</td>
<td>0.0</td>
<td>4.4</td>
</tr>
<tr>
<td>6</td>
<td>4.0</td>
<td>0.4</td>
<td>0.0</td>
<td>4.4</td>
</tr>
<tr>
<td>7</td>
<td>4.0</td>
<td>0.4</td>
<td>0.0</td>
<td>4.4</td>
</tr>
<tr>
<td>Total:</td>
<td>51.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(in hours)
Estimated efforts

Example 1 “business search” story

- The acceptance-testing estimate for this story is **51.4 ideal hours** (6.43 ideal days).

- The estimate for test 3 is a lot more than the others – why?

<table>
<thead>
<tr>
<th>Action</th>
<th>Data</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. Search</td>
<td>Enough concurrent users and category/location combinations to generate 200 searches and 100 hits/second</td>
<td>Success: each user gets appropriate result list within a reasonable response time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test</th>
<th>Preparation</th>
<th>Execution</th>
<th>Special</th>
<th>Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Define details</td>
<td>2.0</td>
<td>0.5</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>Create test records</td>
<td>2.0</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Write tests</td>
<td>4.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Make tests runnable</td>
<td>4.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>12.0</td>
<td>4.5</td>
<td>8.5</td>
</tr>
</tbody>
</table>
Estimated Acceptance-Test (Example 1, “business search” story test3)

• **Preparation** will be fairly extensive.
  • **Define details** - We have to figure how many concurrent users we need to get the expected throughput and define "reasonable" response time. - 2 hours.
  • **create the test records**, it will take longer because we need a lot more— another 2 hours.
  • **Write test** - a longer spike for the automation because of all the concurrent users -4 hours
  • **Make tests runnable** - write the tests and get them running through the HTTP interface (you can't do load simulation through direct calls) - 4 hours

• **Preparation** - total of 12 hours.
Estimated Acceptance-Test (Example 2, “business search” story test3)

- **Execution** of the test will take a relatively long time - 2 hours.
  - Setup tasks: make sure nothing else is going on in our test system, clear out the log files, set up monitoring tools. **0.5 hour.**
  - evaluating what happened: noting response times and throughput data, counting failures, preserving log records. - **1 hour**
  - producing a graphical results report for the team. – **1 hour**

- **Execution-** total of 4.5 hours.
Estimated Acceptance-Test (Example 2, “business search” story test3)

- **Special considerations** come into play.
  - System limitations often dictate a limited window for running load tests.
    - We'll assume we can run this test only between midnight and 8:00 A.M. That means the worst-case time we'd have to wait to run the test would be 16 hours, (if we were ready to test at 8:00 A.M. and had to wait until midnight.)
    - We estimate the average case at half - **8 hours**.
  - Based on experience, we expect to have some kind of problem running a large test like this.
    - We'll probably have to restart it or rerun part of it, not because the system fails the test but because something goes wrong with the test.
    - Restart time – **0.5 hour**
- **Special considerations** total **8.5 hours**.

- All acceptance tests must be automated.
- A manual test may be worse than no test at all.
- Manual tests are unreliable.
- Manual tests are divisive.

“There’s no place for human beings to be doing regression testing manually.”

-Jez Humble
Writing Executable Acceptance Tests (1)

- Things to determine:
  - Test structure.
  - Obtaining concrete data for tests.
  - Assumptions that tests make about the system state;
  - Analysis of tests that change the system’s state.
  - Tests inter-relationships.
  - Modular test organization.

- Executable test automated support:
  - `assertTrue(Boolean)`
  - `assertFalse(Boolean)`
  - `assertEqual(_, _)`
  - Tests must be *self-evaluating: evaluate to Success or Failure.*
Writing Executable Acceptance Tests (2)

• Given a story *S for which:*
  
  • *High level acceptance tests are already analyzed.*
  
  • For each test:
    
    • **Data has already been obtained.**
      
      • *Test assumptions about system state, and inter-relationships has been analyzed.*
      
      • The story acceptance tests are organized within a single module (e.g., class) for testing the story.
    
    • For each action –
      
      write a testing method:

```java
public class SStoryTest {
    public void testAction1() {
        // Test data and assumptions for action1 go here
    }
    public void testAction2() {
        // Test data and assumptions for action2 go here
    }
}
```
Writing Executable Acceptance Tests (3)

**Example 1:**

- *Login story:* A user can login with a valid ID and password.
- **High level acceptance tests:**
  
<table>
<thead>
<tr>
<th>Action</th>
<th>Data</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Login valid id</td>
<td>Valid Id or valid password</td>
<td>Success: User logged in.</td>
</tr>
<tr>
<td></td>
<td>Invalid ID, or invalid password</td>
<td>Failure: invalid id or password message.</td>
</tr>
</tbody>
</table>

- **Additional assumptions and information:**
  - User ID is not case sensitive.
  - User password is case sensitive.
  - `login(<name>, <passWord>)` should be bridged to the real application.

```java
public class LoginStoryTest {
    public void testLogin() {
        assertTrue( login("bob","bobspassword") );
        assertTrue( login("BOB","bobspassword") );
        assertFalse( login("bob","") );
        assertFalse( login("bob","BOBSPASSWORD") );
    }
}
```
Writing Executable Acceptance Tests (5)

- **Example 2:**
  - Add a new Task story: *User can create, update, display, and delete a task.* A task has a name, a description, assignee, estimate, actual time spent, state, and created/updated timestamps.

- **High level acceptance tests:**

<table>
<thead>
<tr>
<th>Action</th>
<th>Data</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Add a new task</td>
<td>Valid values for the task field</td>
<td>Success: Task added.</td>
</tr>
<tr>
<td></td>
<td>Invalid values for task fields</td>
<td>Failure: invalid fields message.</td>
</tr>
</tbody>
</table>

- **Additional assumptions and information:**
  - *Users must be logged in to add, update, or delete tasks.*
  - *The name and description fields are required.*
  - *State has a fixed list of values: Not Started, Started, Completed.*
  - *Estimate and actual time spent must be numeric.*
public class TaskUpdateStoryTest {
    public void testAddNewTask() {
        login("bob","bobspassword"); // Must be logged in.
        // Add a new task with valid values – it succeeds:
        assertTrue( addTask( "User Gui", // name
                         "Create GUI", // description
                         "Bob", // assignee
                         "2", // estimate
                         "3", // actual
                         "Not Started" ) ); // state
        // Add a new task with invalid values – it fails:
        assertFalse( addTask( "", // name -- missing
                             ", // name -- missing
                             "Bob", // assignee
                             "long time", // estimate – non numeric
                             "longer", // actual – non numeric
                             "Moshe" ) ); // state -- invalid
    }
}

• addTask(_,_,_,_,_,_) should be bridged to the real application
Obtaining Data for Tests (1)

- To get the details down, wear the four hats: Customer, User, Developer, Tester.
Obtaining Data for Tests (1)

- To get the details down, wear the four hats: Customer, User, Developer, Tester.
Obtaining Data for Tests (1)

- To get the details down, wear the four hats: Customer, User, Developer, Tester.
- Explore the happy, sad, and bad paths to enumerate the details:
  - Happy, where nothing unexpected happens and all is well
  - Sad, where users make mistakes and data is invalid
  - Bad, where everything breaks apart.
- Look at the system from multiple viewpoints on each path:
  - What the user could do
  - How the data could be partitioned (equivalence classes).
  - External events
- No need to test everything:
  - Acceptance tests do not require 100% success.
- Design your tests to minimize risk:
  - Concentrate on risky areas.
  - Do not search in the light.
Obtaining Data for Tests (2)

The happy, the sad and the bad

- **Story:** Create a screen that will allow the user to search on the Web for businesses by category of business and location.

  - **The happy path:**

    1. **Some user scenarios:**
       - Enter a search, get results, see the desired business info, done
       - Enter a search, get no results, search again
       - Enter search, get page or less of results, don't see desired info, search again
       - Enter search, get more than one page of results, don't see desired info, go to next page of results, see info, done
       - Enter search, get more than one page of results, don't see desired info, go to next page and some or all remaining, never find desired info, search again

    2. **Data conditions:**
       - Category/location combinations that retrieve a page or less of results
       - Category/location combinations that retrieve more than one page of results
       - Category/location combinations that retrieve no results
Obtaining Data for Tests (3)
The happy, the sad and the bad

- The **sad path:**
  1. **Some user scenarios:**
     - Search with a misspelled category, get notice suggesting correct spelling, search again with correct spelling
     - Search with misspelled location, get notice suggesting correct spelling, search again with correct spelling
     - Search with misspelled category and location, get notice suggesting correct spellings, search again with correct spellings
  2. **Data conditions:**
     - Category names misspelled by one and two letters
     - Location names misspelled by one and two letters
     - Pairs of misspelled category and location names for which the correct spellings retrieve one or more businesses
     - Pairs of misspelled category and location names for which the correct spellings retrieve no businesses
  3. **External events:**
     - Changes to database records included in the search while search is in progress
Obtaining Data for Tests (4)
The happy, the sad and the bad

- The bad path:

  1. Some user scenarios:
     - Enter a search consisting only of special characters
     - Enter the maximum number of characters in each form field
     - Click for a search without selecting a category or city
     - Start a search, click the Stop button in the browser, start another search
     - Perform a search, click the Back button from the results screen, perform another search

  2. Data conditions (that would be disastrous ..)
     - Maximum number of database sessions reached
     - Database goes down during session

  3. External events:
     - Volume of concurrent searches exceeds server capacity
     - Server runs out of memory
     - Power to servers goes out
Writing Data for Tests

- Tests data can be organized in tables – based on the high level acceptance tests and the usage scenarios (happy, sad, bad).
- Writing the data in tables might be more convenient to the customer.
- The tables might extend the high level acceptance tests with concrete data details and expected results.

<table>
<thead>
<tr>
<th>Outcome</th>
<th>Id</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>bob</td>
<td>bobspassword</td>
</tr>
<tr>
<td>success</td>
<td>BOB</td>
<td>bobspassword</td>
</tr>
<tr>
<td>fail</td>
<td>bob</td>
<td></td>
</tr>
<tr>
<td>fail</td>
<td>bob</td>
<td>bobspassword</td>
</tr>
<tr>
<td>fail</td>
<td>bob</td>
<td>wrong</td>
</tr>
<tr>
<td>fail</td>
<td>bob</td>
<td>bobspassword</td>
</tr>
</tbody>
</table>
Data Independent Tests

- Tests can be made independent from concrete data, using tables (or other data structures) for storing testing data.

- For example -

```java
public void testLogin() {
    assertTrue( login("bob","bobspassword") );
    assertTrue( login("BOB","bobspassword") );
    assertFalse(login("bob",""));
    assertFalse(login("bob","BOBSPASSWORD") );
}
```

- Can be replaced by:

```java
public void testLogin(loginDataTable) {
    <loop for Success tests>;
    <loop for Failure tests>;
}
```

- Decision between the 2 options: A matter of style or taste. In any case, the concrete data must be coded.
Tests that Change the System State (1)

- Tests must be run as regression tests following every change, or at any integration stage.
- Rule:
  - *No integration is complete unless 100% of the previously passing acceptance tests continue to pass.*
- Some tests assume some initial system state.
- If a test changes the system state – what about regression testing?
  
  Options:
  - Reset the system before testing.
  - Write tests that do not change the system’s state.
Tests that Change the System State (2)

- **State changing tests:**

```java
public class UserIdStoryTest {
    public void testCreate() {
        login("super","superpassword");
        assertTrue( createUserId( "new", "newpassword", "new@tester.org") );
        assertFalse( createUserId( "fred", "", "" ) );
    }

    public void testDelete() {
        login("super","superpassword");
        assertTrue( deleteUserId( "bob" ) );
        assertFalse( deleteUserId( "doug" ) );
    }
}
```
Tests that Change the System State (3)

- Revising the tests so to preserve the system state:

```java
public class UserIdStoryTest {
    public void testCreate() {
        login("super","superpassword");
        assertTrue( createUserId( "new", "newpassword", "new@tester.org") );
        assertFalse( createUserId( "fred", "", "" ) );
        deleteUserId("new");
    }

    public void testDelete() {
        login("super","superpassword");
        createUserId( "john", "johnspassword", "john@tester.org" );
        assertTrue( deleteUserId( " john " ) );
        assertFalse( deleteUserId( "doug" ) );
    }
}
```
The `testCreate` and the `testDelete` can be combined:

```java
public class UserIdStoryTest {
    public void testCreateDelete() {
        login("super","superpassword");
        assertTrue( createUserId("new", "newpassword", "new@tester.org") );
        assertFalse( createUserId("fred", "", "") );
        assertTrue( deleteUserId("new") );
        assertFalse( deleteUserId("doug") );
    }
}
```

Problem: The combined test avoids separate testing of the Create and the Delete actions.
Making Executable Tests Run (1)

- The Executable Test must be linked to the application code.
- The *story actions must be bridged*.

```java
public class LoginStoryTest {
    public void testLogin() {
        assertTrue( login("bob","bobspassword") );
        ...
    }
}
```

- The *login test action should be bridged* – related to the actual code.
Making Executable Tests Run (2)

- Where should this bridge appear?

- Option 1: within the same story test class (e.g., `LoginStoryTest`)
  - Bad, because the action might appear in other stories.
  - See the `UserIDStoryTest` that applies the login action following an assumption on the system’s state.

- Option 2: A parent test class for the application.

- Option 3: Known interface between the acceptance test classes and the application code.
Making Executable Tests Run (3)

- Selected bridge option: A parent test class for the application.

- Example1:

```java
public class LoginStoryTest extends ProjectTrackTest {
    public LoginStoryTest(String name) { super(name); }

    public void testLogin() {
        assertTrue( login("bob","bobspassword") );
        assertTrue( login("BOB","bobspassword") );
        assertFalse(login("bob",""));
        assertFalse(login("bob","BOBSPASSWORD") );
    }
}
```
Making Executable Tests Run (4)

- The bridge: class `ProjectTrackTest`.
  - Question: How is `login(id, psw)` action implemented?
  - Answer: By the `login(id, psw)` method of the application class `ProjectTrackSession`. In order to login you need to have a `ProjectTrackSession`.

```java
import junit.framework.*;
import ProjectTrack.*;
public class ProjectTrackTest extends TestCase {
    public projectTrackTest(String name) { super(name); }
    public void assertFalse(boolean condition) { assertTrue(!condition); }
    public boolean login(String id, String psw) {
        ProjectTrackSession session = new ProjectTrackSession();
        return session.login(id, psw);
    }
}
```
Making Executable Tests Run (5)

- **Example2:**

```java
public class UserIdStoryTest extends ProjectTrackTest {
    public void testCreateDelete() {
        login("super","superpassword");
        assertTrue( createUserId( "new", "newpassword", "new@tester.org") );
        assertFalse( createUserId( "fred", "", "" ) );
        assertTrue( deleteUserId( "new" ) );
        assertFalse( deleteUserId( "doug" ) );
    }
}
```
Making Executable Tests Run (6)

• The bridge: class ProjectTrackTest.
  • Question: How are createUserID(id, psw, email) and deleteUserID(id) implemented?
  • Answer: By the create(id, psw, email) and the delete (id) methods of the application class ProjectTrackUser, respectively.
  • In order to create or delete you need to have a ProjectTrackUser.
import junit.framework.*;
import ProjectTrack.*;

public class ProjectTrackTest extends TestCase {
    public ProjectTrackTest(String name) { super(name); }
    public void assertFalse(boolean condition) { … }

    public boolean login(String id, String psw) { … }
    public boolean createUser(String id, String psw, String email) {
        ProjectTrackUser user = new ProjectTrackUser();
        return user.createUser(id, psw, email);
    }
    public boolean deleteUserId(String id) {
        ProjectTrackUser user = new ProjectTrackUser(id);
        return user.delete();
    }
}
Making Executable Tests Run (8)

- **Selected bridge option**: Known interface between the acceptance test classes and the application code.

- **Example 2**:

```java
public class XTrackTestInterface {
    public boolean login( String id, String psw) {
        return false;
    }

    public boolean createUser( String id, String psw, String email) {
        return false;
    }

    public boolean deleteUser( String id) {
        return false;
    }
}
```
Making Executable Tests Run (9)

```java
import junit.framework.*;
import xtrack.*;

public class XTrackRealInterface extends XTrackTestInterface {

    public boolean login( String id, String psw) {
        XTrackSession session = new XTrackSession();
        return session.login(id, psw);
    }

    public boolean createUser( String id, String psw, String email) {
        XTrackUser user = new XTrackUser();
        return user.create(id, psw, email);
    }

    public boolean deleteUser( String id) {
        ...
    }
}
```
Making Executable Tests Run (10)

```java
public class XTrackTest extends TestCase {

    public XTrackTest(String name) {
        super(name);
        setTestInterface("Test");
    }

    private XTrackTestInterface testInf;

    public void setTestInterface(String interfaceType) {
        if (interfaceType.equals("Real"))
            testInf = new XTrackRealInterface();
        else if (interfaceType.equals("other"))
            testInf = new XTrackOtherInterface();
        else
            fail("Undefined interface " + interfaceType);
    }
}
```
public class XTrackTest extends TestCase {
    ...
    public boolean login( String id, String psw) {
        return testInf.login(id, psw);
    }

    public boolean createUser( String id, String psw, String email) {
        return testInf.createUserId(id, psw, email);
    }

    public boolean deleteUserId( String id) {
        return testInf.deleteUserId(id);
    }
}
Combining Multiple Tests into Test Suites

• Recall the rule:
  • *No integration is complete unless 100% of the previously passing acceptance tests continue to pass.*
• Therefore: Tests must be repeatedly applied.
• It is easier if they are combined into a single *TestSuit object*.

```java
import junit.framework.*;

public class AllTests {
    public static Test suite () {
        TestSuite suite = new TestSuite("ProjectTrack");
        suite.addTest( new TestSuite(LoginStoryTest.class));
        suite.addTest( new TestSuite(UserIdStoryTest.class));
        return suite;
    }
}
```
Acceptance Tests and Quality (1)

• Internal and external quality:
  • External quality is quality as measured by the customer.
  • Internal quality is quality as measured by the programmers.

• The two quality measures can be characterized using different application aspects:
  • External quality: through system requirements/functionality.
  • Internal quality: through execution features
Acceptance Tests and Quality (2)

- External quality can be defined as a set of system features. For example:
  - Whenever the user makes a mistake, a user-friendly error screen appears.
  - It's impossible to crash the server via the user interface.
  - The system can handle a hundred concurrent logins.
  - The system will stay up 99.995% of the time.
  - External quality intends to clarify the customer preferences.
Acceptance Tests and Quality (3)

• Internal quality can be defined as a set of process attributes:
  • 85% of unit level defects are found by automated unit tests.
  • 80% of estimates are achieved.
  • 90% of the stories picked for an iteration are completed, on average.
  • 100% of projects deliver a product to the customer.

• Internal quality should always be maximized
Acceptance Tests and Quality (4)

- Internal and external quality features:
  - Number of defects in the code – internal.
  - Number of overtime hours spent – internal.
  - Customer satisfaction – external.
  - Development team morale – internal.
  - System reliability – external.
  - Code readability – internal.
Acceptance Tests and Quality (5)

- **Question:** would customers always want the highest quality system they can get?
- **Answer:** No. Customers with time and money constraints consider a lot of other things.

**Customers do not always want the highest quality system.**

- Customers cannot make intelligent choices about quality without understanding the cost of producing software that contains the desired features at the desired level of quality.
- This information is provided in the form of story estimates, and in details of acceptance tests.
- Defining details of the acceptance tests is really defining the quality the customer expects of the system.
Non agile development practice

- Even if your development team isn't using agile practices, you can benefit by applying agile practices to quality assurance and testing.
- Work closely with business (customers) in the early stages of the project.
- Help business experts visualize the final product; write acceptance tests and requirements accordingly.
- Have the business experts define what quality is for the software being produced.
- Attend system design meetings and help the programmers stay focused on the immediate needs of the customers.
Non agile development practice

- Use the test-automation practices - They aren't just for XP projects.
- Work with programmers to test components as soon as the components are ready. Don't wait and test at the tail end of the project.
- Be a change agent; work to get your team to implement practices such as unit testing.
- Involve the business experts in acceptance testing.
- Hold a retrospective to review what worked and what didn't and select areas for improvement in the next release.