Requirements and Use cases

Introduction to Requirements

You should distinguish between [System Requirements] from [Software Requirements] in this course. You also need to learn about Functional and Quality requirements (below). The main skill is writing use cases. You need to know some popular or classic methods.

System Requirements vs Software Requirements

System Requirements constrain the whole system to do what the client needs. They are analysed and the results allocated to different components in the systems architecture -- hardware + software + procedures + data + people. You must be able to show that if each part fits its requirements then they will work together to achieve what is required before we spend any money. We test this by using walkthroughs, calculations, and inspections. Notice that a requirement must state a necessary constraint on a part of a system. System requirements are often high level requirements. They are broken down to smaller constraints on the individual parts of the system.

Software Requirements are more or less precise descriptions of what software (and sometimes hardware) has to do plus how well it should do it. They must not say too much about the technology used or how the software is coded. They give logical picture of a piece of software. Software Requirements have to be thought about and specified whenever software is being developed or maintained. It is especially important with outsourced software development. As a rough rule, the worse the communications between stakeholders and developers is (for example outsourced overseas), the more formal and rigorous your requirements must be. Well written requirements tend to avoid over-constraining the solution to problems. They avoid describing the internal structure of the software.

In Software requirements you must carefully distinguish [Functional requirements] from [non-functional requirements] as described below. Both types are important but they have to be dealt with in very different ways.

Systems Design is not program design The heart of system design is DATA. In particular, PERSISTENT data. This is data outside the computer. System design is about what goes on between and outside computers. It involves specifying the data, the hardware, the people, and the procedures. It specifies how software behaves but not how it is programmed. Program design is about what goes on inside a computer.
**Functional requirements vs Quality Requirements**

This is a key distinction. **Functional requirements** define what a process or system must do. They tend to define what happens to the data that flows in and out of the process. They define the what not the how. For example:

The student is entered in the roster for the chosen section.

In addition to these functional requirements we have **quality requirements** that define how well the process or system carries out its function. The traditional name is **non-functional requirements** for these. They demand special care when developing software.

Quality requirements are about **how good** the software has to be. For example:

Registration must be completed in less than 1 second 90% of the time.

Quality requirements are often about things that must not happen. For example:

During registration the student’s Id must be protected from sniffers.

Typically you will find both functional and quality requirements in the initial (high level) requirements. You should learn to parse them. As an example:

Grades will be posted securely.

- Functional: Grades will be recorded...
- Nonfunctional: ...securely.

**Typical high level quality requirements**: include security, safety, reliability, speed, efficiency, storage space, ease of use, sustainability, economy, etc. You can add various costs and the time to develop and/or maintain a system to these.

Notice that a functional requirement describe something that must be done but a quality requirement can say that something must be difficult or even impossible:

A student can not change any grades stored on the system

The architecture of software often depends more on the qualities than the functions. Quality requirements start off as a nice desirable quality like "User friendly". But they need to be made precise. They need to be expressed as a set of measurements that can be made of the final product: "The user can find out how to cut and paste data on the form without being told", "In a sample of 10 users, after using the product, 9 of them rate it as 'easy to use'" -- The best quality requirements are based on national standards. Working out the detailed qualities of a component so that it is capable of supporting the high level requirements is a challenging and interesting task.

Similarly, a requirement that "Grades are stored securely" might imply that nobody, even if they access the data, can determine a student’s grade unless they are the teacher or the student. This is quite difficult to achieve, and might be watered down in practice so that reading grades without being the teacher or the student should take several hours of hacking. Another refinement is: Anyone can read the grades but they can not figure out who each grade is assigned to.

**Common Error -- Thinking that a non-functional requirement means it does not work**

Use the more modern terminology and talk about "Quality Requirements"!

**Technique -- List Features**

A common technique in mass marketed software is to document the requirements in terms of a list of "features" that will be provided. This is largely driven by marketing. The technique is also used in other situations. Executives and steering committees like making up a list of the features they want if a new or updated system. They are a mix of functions and qualities. They are disorganized and difficult to analyze. It is possible that this technique contributes to the low quality of much software.

**Technique -- Top-down Functional decomposition**

Functional requirements can be developed top-down. You start with large scale needs and then decompose them into a set of requirements, which together achieve the top-level. DFDs help to do this right. However, this technique was taken to extremes in the two decades proceeding the 1990's. As a rule: stop decomposing functions when you are down to a single program. Then switch to other techniques.

**Tool -- Checklists and Forms**

A checklist is a handy tool to avoid forgetting things. So are forms. So most organizations have a whole set of these to
help manage projects. For example, in the CSUSB CSE School, we require all senior and graduate projects to follow the IEEE standard "Software Requirement Specification". [../SRS/]

**Common Error -- ignoring bad input**

It is easy to only specify the processing of perfect data. However we must also specify what can go wrong, and what to do about it.

Use Cases (next) formalize the need to study the good and the bad interactions between users and your system.

**Requirements Tool -- Use cases**

For the last 10 years Jacobson's idea of a "use case" has come to dominate software engineering and development. It organizes the visible behavior of a system around the needs of the users. A **use case** is a description (in text or table) of how a user uses the system to get something tangible from it. Low level functions/processes in a DFD tend to be use cases.

**Definition of Use Case**

A use case describes how a user gets something of benefit from a system. It can also spell out what can go wrong and what is done about it. A use case describes the interaction between a user and the software -- it must not describe what happens inside the software! It can show the steps taken by the user to get what they want -- but not format/layout/media.

```
Use_case ::= defines the following,
Net
  1. Users and other stakeholders
  2. What they get out of the system -- Goals
  3. How they try to get what they want -- Scenarios
(End of Net)
```

For example:

```
A student registers in a class by logging in (if not logged in), selecting an open class, and saving the new schedule.
```

Key point: a use cases says who wants what, and how they can get it.

```
1. USER: A student
2. GOAL: registers in a class
3. SCENARIO: by logging in (if not logged in), selecting an open class, and saving the new schedule.
```

Use cases are text documents. Use Cases are not diagrams. Use case **diagrams** are just a visual summary of a collection of use cases. A useful use case diagram shows many users and use cases.

In essence, a use case defines a set of scenarios associated with one user and one of their needs, but you can add:

1. The secondary actors -- involved but not initiating the use case.
2. The other stakeholders.
3. The value to the user and enterprise.
4. The development costs: time and money.
5. Other costs: equipment and personnel.

**Tools for use cases**

Any word processor or editor can be used to write use cases. Remember: a use case is written not drawn. Only when you have several use cases do you need a diagram tool: Dia, Visio, and IBM Rational Rose all help you draw these diagrams.

**Hints on use cases**

- **First name your uses cases** (Strong verb + who) then elaborate the interesting ones.
- A good use case name has a strong verb in it. The name of a use case should start with a strong verb.
- Start a use case description by naming the primary user who gets something out of the system.
- Refine use cases a little bit at a time: First the name, then a **brief** description, then a **casual description**, and then elaborate to a **fully dressed** one (if it is interesting enough).
- At any time you will have uses cases in all 5 states of description: Name, Name+User, Brief, Casual, Fully dressed.
If there are more than half-a-dozen use cases it may pay to give them a short name or number:

- (UC1): student registers in class.
- (UC2): Faculty gets a roster for their class.

You can then use hyperlinks to refer to them.

Writing use cases is a team sport. Get developers and stakeholders involved in drafting and refining them.

Do the interesting ones first. **Interesting use cases** are ones with high value to the stakeholders that also test the architecture of the system. Interesting use cases are not single steps like "Login", "logout", etc.

Most systems include some **standard administrative use cases** such as "start up", "shutdown", "back up data", "Maintain IDs", etc. These are best done after the interesting use cases since their details depend on discoveries made when working out how users get things of value out of the system. Notice that administrative use cases need different designs to ordinary ones. [Limoncelli11]

Use cases have set of scenarios **Scenarios** are a sequence of steps taken by the user and the system. The first scenario must be the ideal "sunshine" scenario. Later scenarios spell out less pleasing scenarios for the user. But don't get bogged down in all the special ways things go wrong (power cut, mistyping, interrupted user, network down,...) until you've finished the main success scenario. Use the steps in existing scenarios to create the extensions: ask what can go wrong with each step in turn and write an alternative scenario that handles the problem.

Each step should state what the user does and/or what the system responds. It is OK for a step to be the name of a fairly complex procedure as long as the name makes it clear what is achieved. Define the details later. If (and only if) a step is interesting and turns up in several use cases you can document it as a use case and - - <<include>> - - > It.

The steps must not mention how the system does something. However, a step can refer to actions involving secondary actors or to data that is held in the system database. Avoid describing internal coding details of algorithms and objects. Avoid user interface details. Keep the actions logical rather than physical. The steps must not describe the hardware, format, or look-and-feel of the user-interface.

Make sure you **store use cases** so that they are easily found, edited, and used. For example: Put use cases on a project web site with a search engine. Recall that use cases are hyper-text -- find a repository that searches text files. Have back up copies and plans. Keep track of different versions. With a team, use tools to handle versions and revisions. Your CSE home directory is OK for your own work but it is not designed to make sharing easy. Both Google Docs and Drop Box are convenient places in the cloud for student projects.

You can state that a sequence of steps **can be repeated** -- "The user inputs selects several items to put in the basket." You can have **simple conditional step in a scenario** -- "If the user has a Club Card they input its number and the system notes the extra discount" -- but don't write nested if-then-elses. Handle these by introducing an extension or alternate scenario. If the sequence/logic starts getting complicated then you may need to use an activity diagrams etc. [r1.html] to sort it out.

Delay drawing a use case diagram until you need a visual summary of all the things a piece of software can do for its user. See below...

---

**Three Formats for Use Cases -- Brief -- Casual -- Fully-dressed**

- **Memorize the three formats.**
  - **Remember -- the Name comes first**
    - With a strong verb!
  - **Brief Format Use Case**
    - (Name): Terse one paragraph description of who does what to get what.
    - Examples:
      - (nightmare use case): The president of the USA can login, identify himself, and start World War III.
      - (Process my Email): A student can login, see the headers of Email messages and then read, delete, reply to, and forward a message.
  - **Casual Format Use Case**
    - (Name): short name
**Main Success Scenario**: one paragraph.

**Alternative scenario 1**: if ..., one paragraph

**Alternative scenario 2**: if ..., one paragraph

Example

1. Name: Guest registers on system.
2. Main: A guest can register with the system by giving certain details, supplying a system "handle", and a password(twice). The system responds by reflecting the information back to the guest (but not the password). The guest confirms the data and is allowed access to it via the usual login.
3. Alternative 1: If the "handle" is already in use the guest is asked to change their "handle".
4. Alternative 2: If the two passwords are not equal the guest must re-input them.

---

**Fully Dressed format**

This is a long document with several sections. Here is a list of sections. Some are optional. Some can be repeated.

1. System
2. Name -- May include a unique Id.
3. Actors -- Main actors and other participants
4. Stakeholders
5. Description -- Brief Format Description
6. Main Success Scenario -- sequence of steps achieving the primary actor's needs.
7. Alternatives & Extensions -- scenarios for the less successful or more complex cases. Note: number alternatives to show which in which main flow step they can happen. "1a", "1b", "1c" are alternatives to "1", "2a" to "2", "2-5a" can occur in any step from 2 to 5 inclusively, and "*a" can happen in any step.
8. Preconditions -- what can be assumed to be true before. Example: user must be logged in.
9. Post Conditions -- what must be guaranteed to be true after. Example: student is enrolled in class.
10. Assumptions
11. Special Requirements -- reliability, speed, cost, ...

**Example of a fully dressed use case in table format**

<table>
<thead>
<tr>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System</strong></td>
</tr>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td><strong>Actors</strong></td>
</tr>
<tr>
<td><strong>Other Stakeholders</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
</tbody>
</table>
| **Main Success Scenario** | 1. Customer enters credit card number and expiration date  
2. System verifies card with credit Card company  
3. System sends authorization message to the customer |
| **Alternative 2a** | In step 2, if the credit card company rejects card, system sends rejection message to user. |
| **Alternative 2b** | In step 2, if the credit card company does not respond within 10 seconds,  
1. Tell the user of the delay.  
2. Repeat verifying the card. |
| **Alternative *a** | In any step, the client does not respond after 30 minutes, cancel the transaction. |
| **Alternative *b** | In any step, if there is a power failure.... TBD |
| **Alternative ...** | - |
| **Precondition** | Customer has selected at least one item and has proceeded to checkout area. |
| **Post condition** | Credit card information has been validated. Customer can continue with order. |
| **Assumptions** | User has a credit card |
| **Special Requirements** | Security -- No way for customer data to be sniffed. |
Common Use Case Errors

1. The use case is not named with a strong active verb. "Authentication" is an error. "Authenticate" is better.
2. The use case does not define a tangible and desirable goal for its user. "Authenticate" is probably too small a goal. "Enroll in class" is better.
3. There is no main or sunny day scenario.
4. The use case is too high level. It should define a user and something specific that they need.
5. The use case is too small/low level to be useful to any body. Try the "Hey mom, I just ___ " test.
6. Too much predictable detail.
7. Including any physical or technical details of how a use case is coded: example mentioning the DBMS or interface technology. These are going to change. Keep use cases logical or essential.
8. Attempting to complete one use case before starting on others. Start with a list of use cases and select one to work on, then move on to another.
9. Missing descriptions but a nice diagram.
10. Drawing a diagram with only one use case.
11. Trying to put the whole system in one use case.
12. Copying use cases from other projects that are not relevant.

User Stories are not Use Cases

A user story can be anything that the user desires of a system, such as it being able to run 24/7/365. All brief use cases are user stories. But some user stories are not use cases. For example:

1. The system will be user friendly.

is a story but not a use case. This is a high-level, fuzzy, nonfunctional requirement. It needs refinement: who is the user? what do they find to be friendly? Further: how do we test "user friendliness"? Once we can express the nice, warm, fuzzy term as an objective test -- then we are getting close to producing solutions that achieve what the stake holders want. Alternatively we may identify and area that need research -- what do the mean by "user friendly"?

A Scenario is less than a use case

A scenario only becomes a use case when you add a description of what the user gets out of the scenario (or what he user was trying to get out of it). Note: KISS in CSE557. We do the complex diagrams and formats in CSE375. Develop use cases using text processors... Use the Casual Format use case in this class:

Use Case Diagrams

2. Keep use case diagrams very simple. Here are the four symbols you need: actor, communication, system, use case.

Keep Diagrams Simple!

3. DON'T put steps on a diagram! For example: "login" is a common first step in many scenarios. Do not show "login" as a separate use case. Show it as a step, or document the need for "login" as a pre-condition for the use case.
4. Don't go overboard on drawing use case diagrams. Keep them simple. Use cases are about writing not drawing.
5. Use case diagrams show communication between actors and use cases. Actors do not communicate with out a use case. Use cases do not communicate under any circumstance.
6. The use case diagram is a good place to "boiler plate" the administrative use cases. They are always there but don't help you analyse the situation. It is best to put them off until you've analysed the interesting use cases.
There are notations for showing common scenarios shared between several use cases (<<include>>) and for showing extended (alternative) use cases. Please avoid them, if you can, in this class. The following part of a use case diagram shows connections between use cases: Adding classes includes logging in and logging out. So does getting a class roster. It also shows the special alternative case of trying to add a full class. This is called an "extension" and is expressed as extending particular steps in the basic use case. These are the only connections between use cases that you should draw.

The diagram does not show the "Special case of" arrow. This is rarely used.

**When should we use a UML use case diagram**

When you want to present your ideas and have more than one use case or user.

You need one in complex projects when you have a dozen or more use cases and you notice some common patterns developing.

They make a nice poster for the developers work space. Even better put it on a board next to an ERD and change it each iteration.

**Questions on use cases**
What is a Sunshine scenario
The simplest perfect case when a user does the right things, the system has the right resources and the user gets what they wanted.

How many use case scenarios should be included in a project
There is no ideal number. Big projects have many use cases and small projects have a few (simple) use cases. In fact the number of the total number of scenarios indicates how big and expensive a project will be.

Methods of Requirements Development

Involve Stakeholders
All successful requirements specification processes involve the stakeholders in the process.

Method -- Traditional System Development Life cycle -- SDLC
A straightforward project can follow the traditional sequence:
1. Analyze situation
2. Design System
3. Specify Software Requirements -- in detail, top down, ...
4. Implement Software
5. Test and Improve Software
   - Unit tests -- inside software
   - Acceptance tests of pieces of software
   - Integration tests at the system level
   - Performance tests/Benchmarks
6. Training, initial data, and cut over.
7. Operate and Maintain System

In the above the requirements must be complete and correct before implementation starts.

Method -- JAD -- Joint Application Design
This is similar to Participatory Design. Joint Application Design involves the users and other stakeholders in the process of designing systems and specifying requirements. Typically the developers and the stakeholders have meetings. And the meeting generate requirements for the software being developed. The advantage is that the stakeholders have a strong chance of getting what they want.

However, they can also run a large project into the ground by producing requirements faster than they can be coded. When the clients continuously produce new and corrected requirements faster than the developers can implement them JAD breaks down. Some form of incremental delivery and a system to control the rate of change of requirements is almost a necessity when using JAD with large systems.

Method -- RAD -- Rapid Application Development
The two above techniques assume that the stakeholders and the developers have all the knowledge that they need to come up with a better system. This is not always true. We often need to do some research before we develop software. One response to the need to learn about the solutions and needs is the RAD method of prototyping.

The aim is to give stakeholders what they want quicker. RAD relies on tools that let's developers throw software together quickly. Prototypes define requirements and are then thrown away. They are developed quickly with no attention to other qualities. They are not documented, tested, or designed to be maintained. RAD was fashionable in the 1990's but is getting less press these days.

These prototypes are best used when you think that people don't really know what they need. They are also very useful for sorting out user interfaces; you can show two versions of the GUI and ask which is best? User interfaces need to evolve. So it is best place the user interface a separate but well-crafted part of the software.

Agile Methods
Agile methods stress people and providing value to the stakeholders. Instead of prototypes they provide a sequence of quick high-quality, high value, useful iterations. Instead of meetings they tend to have an informal work place where everybody can hear what is going on... complete with on-site users. They also stress shared code and documents. They often stress doing detail work in pairs: pairing. Some have "stand up meetings".

Instead of paper work documentation they stress face-to-face communication, test plans, chalkboards or white boards, posters, 3-by-5 cards, sticky notelets, and code. They want documentation to be "just good enough".
However -- "Don't let the sun set on bad code". They stress excellent source code and 100% compliance with tests for programs -- from the first iteration onward. They stress automating routine tasks -- like testing. Indeed most of them are "Test Driven Development" -- they write the tests before they write the code. Thus the software requirements specification for a project is a set of tests!

They often include an explicit "code refactoring" process that improves the quality of working code without breaking any tests. This counters the well known tendency of code to "rust" as small changes are made to it. In refactoring a series of small changes are made to the code that are sure to not break it. Each is followed by a quick retest. Step by step the code becomes clearer and better structured.

The key difference between RAD and Agile development is that one uses prototypes and the other iterations.

**Method -- XP - eXtreme Programming**

XP is a very rigorous agile method -- it works in terms of 1 week iterations and 100% test driven coding. The whole programming team must work in a shared area and must involve a user representative. Requirements are expressed as user stories written on cards. The week starts with a negotiation to choose the cards that will be tackled in that week. The user writes acceptance tests, the programmers work on the code. All the code is open to all the programmers to work on. But programmers always work in pairs: one types and the other thinks and comments. The week ends in refactoring the code so that it remains easy to understand and easy to maintain.

**Iterations vs Prototypes**

An iteration is a high quality and usable subset of a project. One iteration is also a part of the development time that carries out a more or less complete Software Development Life Cycle for a few functions at a time: *Analyze a little, Design a little, Code a little, Test a little.*

Each iteration is well made and documented but doesn't meet all requirements, yet.

A prototype is often incomplete and nearly always of low quality. As a rule it is harder to put quality into bad software than it is to add features to good software. Some call them "throw-away" prototypes.

### Can a prototype be incomplete

Yes.

### Can a prototype be of low quality

They often are. A high quality prototype program is best called an *iteration* and in some sources it is called an *evolutionary prototype*.

### Can RAD prototypes evolve into better programs

In your dreams!

RAD means low quality prototypes ... and low quality code makes evolution difficult.

### Does a special RAD language like DELPHI give prototypes that can be converted into a final product

Only in small projects -- In My Humble Opinion.

In my experience the prototype tends to become a nightmare. For example see [../tools/mth2html.txt](../tools/mth2html.txt) (Mad laughter echoes from the dark dusty dungeon of deadly code)

**A Family of Methods -- RUP -- Rational Unified Process**

The RUP is a family of similar processes. RUP can be tuned to operate like a traditional SDLC or an agile process. We will use something like this in the next class CSE375.

### How do you decide on the method JAD, RAD, ...

The different methods do have different personalities -- even if they share thought: "Involve the stakeholders in the process". In fact the first decision you need to involve others in -- is in selecting the method: Have they the time and resources for JAD meetings? Would they prefer to see demos of the product every now and then? Would they be willing to send a representative to live and work as part of the development team (as in XP)? ...

### What is the best way to start the design of a system

I still like the classic approach: map out the current system and look for a set of processes and data stores that could work *better*. Think in terms of doing surgery to the living enterprise: "What organs need replacing?". As you talk to people take note of where the current system hurts. Be ready to brainstorm. Then "drill down" into the
details at the boundaries of the old and new.

Don't forget to involve the stakeholders and get your ideas reviewed.

**What are the advantages and disadvantages of JAD**

JAD gives people what they want -- but occasionally JAD tends to lead to wasting time and money on a system that does not work.

Ideally as each requirement comes out of the JAD team it should have a price tag attached to it, and a Change Control Board (or equivalent) decides if the enterprise can afford the price for the new feature/quality/... 

**Which is used most in the workforce -- JAD or RAD**

I don't have any data! My feeling (from reading 5 or 6 magazines on software development each month) is that RAD getting unpopular.

Indeed I think we are seeing mixed methods where a JAD meeting is used with a PC prototyping system and projector to rapidly develop a mock up system -- RAD.

------- (end of section **Methods of Requirements Development**) <<Contents | End>>

**Online Resources**

**Story -- Think outside the box**

This [s6083.htm](http://csci.csusb.edu/dick/cs557/r2.html) paper is in the IEEE Digital Library. You can read on campus, via the library proxy, or if you are a member of the IEEE-CS.

**Enrichment -- Real Quality Requirements**

The following should be read before you get into a real project. There has been a lot of research and thought about this type of requirement and one result has been some complex models, for example [column4](http://www.ebgconsulting.com/) to see a list of things that a quality requirement should include plus some UML pictures. Focus in on the checklist (part 3) of the article.

**Creeping Featuritis and Death by JAD**

Uncontrolled features and JAD can ruin a project.

See [000980.html](http://csci.csusb.edu/dick/cs557/r2.html) in the Coding Horror series for example feature lists, and how "creeping featuritis" leads to buggy software.

A classic example was the FBI VCF project but the page on it has vanished.

**Details on JAD**

See [Joint_application_design](http://csci.csusb.edu/dick/cs557/r2.html) for a good definition and description.

There have been several books published on JAD

2. Bodker, Kensing and Simonson, Participatory IT design, MIT Press 2004 QA76.9 W88B65 2004
3. Courage and Baxter, Understanding your users, Morgan Kaufman 2005 QA76.9 H85 C69 2004

**Who is Ellen Gottesdiener**

She is an author and consultant [ellen in about](http://csci.csusb.edu/dick/cs557/r2.html)

**Who invented JAD**

Interesting question. The earliest reference in my collection of publications[ JAD](http://csci.csusb.edu/dick/cs557/r2.html) on software development is to articles on the FAA's "Advanced Automation Program" from 1987. But -- the articles describe "joint application design" with their users without once mentioning "JAD" by name. I guess that I already knew the term from other sources and attached it to this example.

JAD is explicitly mentioned in a special issue (Vol 36 no 4) of the "Communications of the Association for Computing Machinery" on "Participatory Design". An article (page 41) credits IBM in the 1970's as the source. This article also tracks the evolution of the JAD method... The whole issue (looking back at it from 13 years later) is full of ideas that
RAD on the Wikipedia

See [Rapid_application_development] for history etc.

Real world examples of using a prototype

Here are half-a-dozen references to publications of experiences with various kinds of prototypes:
[Carr] [Carra] [Cer] [Cloyd] [Con] [Gordon] [Kamata] [Keil] [Morales]

The Agile Alliance

[http://www.agilealliance.com/]

RUP

For more see [RUP] on the Wikipedia.

CSE375 is all about UP which is a non-proprietary version of RUP. See [../cs375] for details.

Alternate Technique -- Usage Scenarios

Usage Scenarios are simpler and more precise than use cases. They rapidly get to the heart of how the software and its users interact. However they are not widely used. See [usage_scenarios.html] for more.

Use cases on the Wikipedia

See [Use_cases] for a good description.

Use case Resources on the Web

Sensible introduction: [systemUseCase.htm]

Proposal made to Hewlet-Packard [use_case.pdf] (Full disclosure -- I worked with the author 40 years ago).

Example from Virginia Tech [usecases-ex1.html]

Wikipedia: [Use_cases]

History and Advice [Use+cases,+ten+years+later]

Books on Use Cases

Alexander Cockburn: writing effective use cases

Martin Fowler; UML distilled

UML diagrams on the web

You can find the UML notations in Fowler and the UML manuals. See [Use_case_diagram]

If you have time or the tools you can make the use case diagram a MAP in an HTML page and link the bubbles to their definitions See [UseCases.html] for an slightly over the top example.

Here are some easily found examples of diagrams [images?q=use+case&hl=en] but most are more complex than we need for CSE557!

Review Questions on Use Cases and Requirements

1. What is a use case?
2. What are the levels of description of a use case?
3. Give examples of a requirement and parse it into functional and quality parts.
4. How does a non-functional or quality requirement differ from a functional requirement?
5. Compare and contrast: functional and quality requirements.
6. Compare and contrast: use case and scenario.
7. Compare and contrast: use case and user story.
8. Compare and contrast: use case diagram vs each of these in turn: DFD, ERD, and Activity Diagram
9. List the components of a fully dressed Use case.
10. Name half-a-dozen high level (fuzzy) quality requirements.
11. Take a given high level non-functional requirement and propose a partial test for the requirement.
12. How use case diagram, use cases, DFDs, ERDs, and other artifacts fit into a strict systems development life cycle?
13. What do you say to a new recruit who refers to a use case diagram and calls it a use case?
14. Correct a given use cases with errors...
15. Write a use case that fits in as part of any of the requirements described below for: [20040223ChouFlyer.pdf] [20040308ChimpanchFlyer.txt] [20040318RohiniRreddy.txt] [20050315HumairaRahim.txt] [20050322AmitDhir.txt] [20050321JuyongJeong.txt] [20050509RayYang.txt]
16. In the UML, as taught in this class, which is the node, the entity, the use case, and the activity in the diagram below.

A  B  C  D

17. What connections are allowed (in CSE557) between use cases in a use case diagram? Between actors? Joining actors to/from use cases?
18. What does JAD mean?
19. How do you do a JAD Meeting?
20. What does RAD mean?
21. How does an iteration differ from a prototype?

. . . . . . . . ( end of section Requirements and Use cases) <<Contents | End>>

Abbreviations

1. **TBA**: "To Be Announced".
2. **TBD**: "To Be Done".

Links

Notes -- Analysis [a1.html] [a2.html] [a3.html] [a4.html] [a5.html] -- Choices [c1.html] [c2.html] [c3.html] --
Data [d1.html] [d2.html] [d3.html] [d4.html] -- Rules [r1.html] [r2.html] [r3.html]
Projects [project1.html] [project2.html] [project3.html] [project4.html] [project5.html] [projects.html]
Field Trips [F1.html] [F2.html] [F3.html]
[about.html] [index.html] [schedule.html] [syllabus.html] [readings.html] [review.html] [glossary.html] [grading/]

End