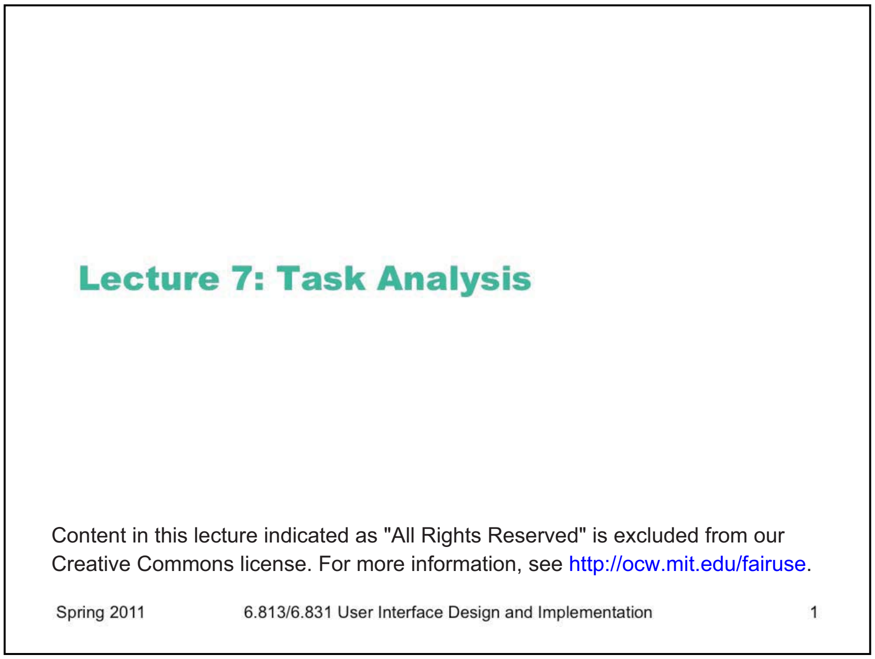
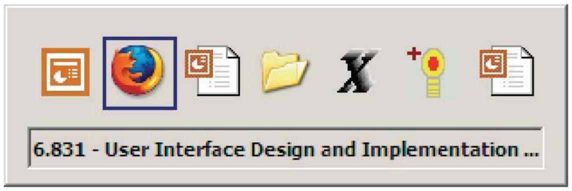


Lecture 7: Task Analysis

UI Hall of Fame or Shame?



6.831 - User Interface Design and Implementation ...

Spring 2011 6.813/6.831 User Interface Design and Implementation 2

Today's candidate for the Hall of Fame & Shame is the **Alt-Tab** window switching interface in Microsoft Windows. This interface has been copied by a number of desktop systems, including KDE, Gnome, and even Mac OS X.

For those who haven't used it, here's how it works. Pressing Alt-Tab makes this window appear. As long as you hold down Alt, each press of Tab cycles to the next window in the sequence. Releasing the Alt key switches to the window that you selected.

We'll discuss this example in class. Here are a few things to think about:

- how learnable is this interface?

- what about efficiency?

- what kinds of errors can you make, and how can you recover from them?

My comments:

The first observation to make is that this interface is designed only for keyboard interaction. Alt-Tab is the only way to make it appear; pressing Tab (or Shift-Tab) is the only way to cycle through the choices. If you try to click on this window with the mouse, it vanishes. The interface is weak on affordances, and gives the user little help in remembering how to use it.

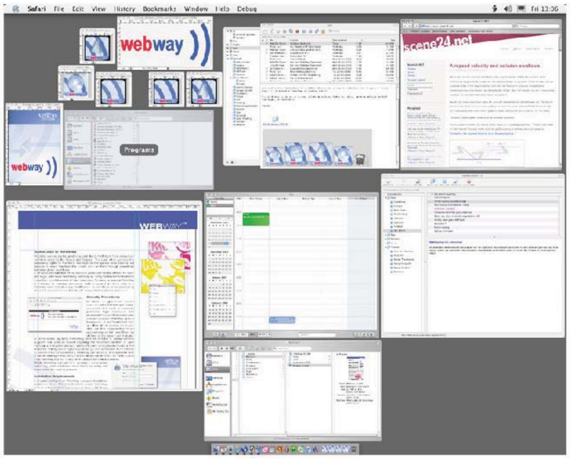
But that's OK, because the Windows taskbar is the primary interface for window switching, providing much better visibility and affordances. This Alt-Tab interface is designed as a **shortcut**, and we should evaluate it as such.

It's pleasantly **simple**, both in graphic design and in operation. Few graphical elements, good alignment, good balance. The 3D border around the window name could probably be omitted without any loss.

This interface is a **mode** (since pressing Tab is switching between windows rather than inserting tabs into text), but it's spring-loaded, happening only as long as the Alt button is held down.

Is it **efficient**? A common error, when you're tabbing quickly, is to overshoot your target window. You can fix that by cycling around again, but that's not as **reversible** as just moving backwards with a mouse. (You can also back up by holding down Shift when you press Tab, but that's not well-communicated by this interface, and it's tricky to negotiate while you're holding Alt down.)

UI Hall of Fame or Shame?



© Apple Inc. All rights reserved.

For comparison, we'll also look at the Exposé feature in Mac OS X. When you push F9 on a Mac, it displays all the open windows – even hidden windows, or windows covered by other windows – shrinking them as necessary so that they don't overlap. Mousing over a window displays its title, and clicking on a window brings that window to the front and ends the Exposé mode, sending all the other windows back to their old sizes and locations.

My comments:

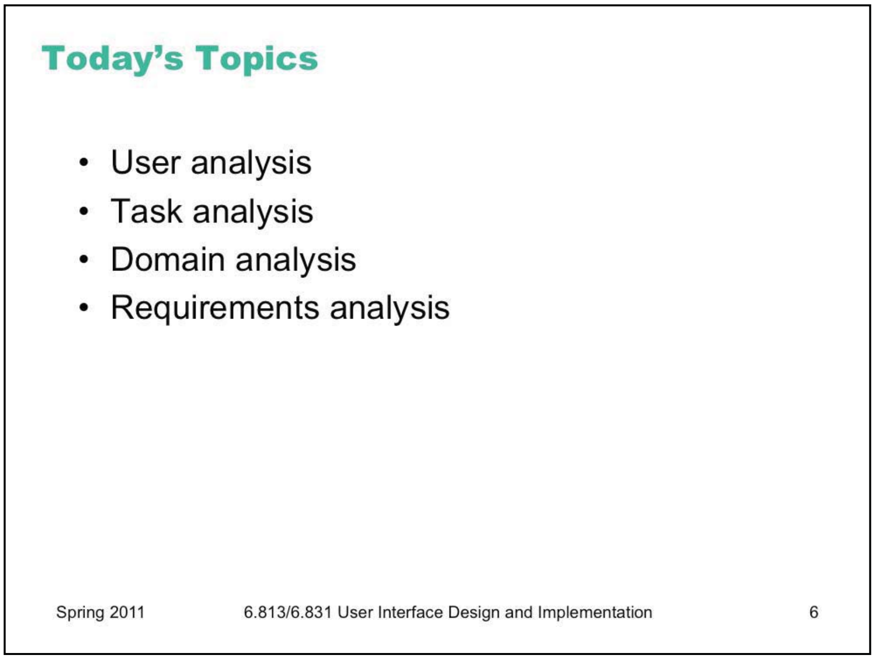
Like Alt-Tab, Exposé is also a **mode**. Unlike Alt-Tab, however, it is not spring-loaded. It depends instead on dramatic visual differences as a mode indicator – with its shrunken, tiled windows, Exposé mode usually looks a lot different than the normal desktop.

To get out of Exposé mode *without* choosing a new window, you can press F9 again, or you can click the window you were using before. That's easier to discover and remember than Alt-Tab's mechanism – pressing Escape. When I use Alt-Tab, and then decide to abort it, I often find myself cycling through all the windows trying to find my original window again. Both interfaces support **user control and freedom**, but Exposé seems to make canceling more **efficient**.

The representation of windows is much richer in Exposé than Alt-Tab (at least on Windows XP). Rather than Alt-Tab's icons (many of which are identical, when you have several documents open in the same application), Exposé uses the **window itself** as its visual representation. That's much more in the spirit of direct manipulation. (The version of Alt-Tab included in Windows Vista now shows images of the windows themselves – try it!)

Let's look at efficiency more deeply. Alt-Tab is a very linear interface – to pick an arbitrary window out of the n windows you have open, you have to press Tab O(n) times. Exposé, on the other hand, depends on pointing – so because of Fitts's Law, the cost is more like O(log n). (Of course, this analysis only considers motor movement, not visual search time; it assumes you already know where the window you want is in each interface. But Exposé probably wins on visual search, too, since the visual representation shows the window itself, rather than a frequently-ambiguous icon.)

But Alt-Tab is designed to take advantage of **temporal locality;** the windows you visited recently are at the start of the list. So even if Exposé is faster at getting to an arbitrary window, Alt-Tab really wins on one very common operation: toggling back and forth between two windows.



Today's Topics

- User analysis
- Task analysis
- Domain analysis
- Requirements analysis

We've seen that UI design is iterative – that we have to turn the crank several times to achieve good usability. How do we get started? How do we acquire information for the initial design?

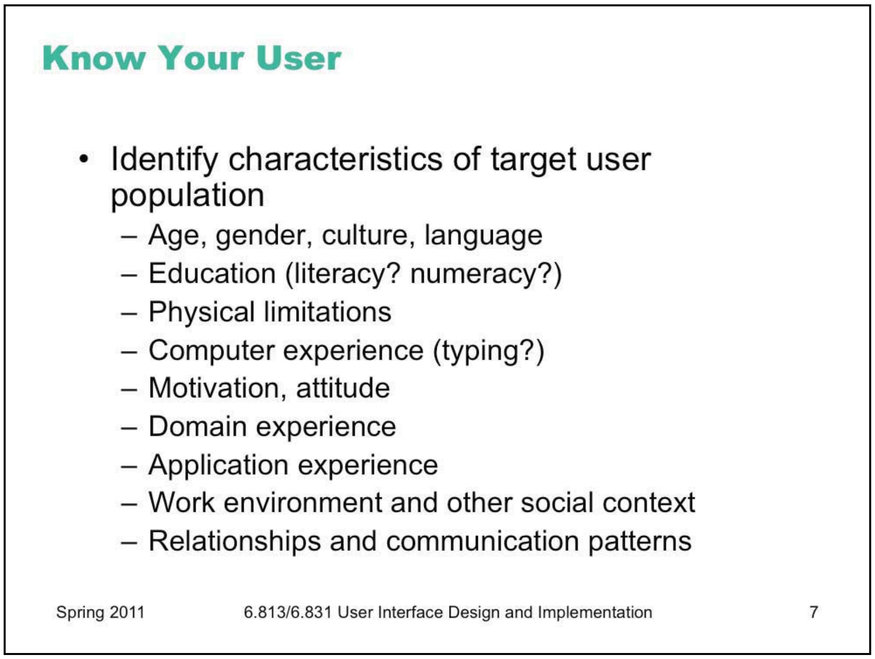
Today's lecture is about the process of collecting information about users and their tasks, which is the first step in user-centered design. We'll talk about four key steps:

User analysis: who is the user?

Task analysis: what does the user need to do?

Domain analysis: what is the context the user works in (the people and things involved)?

Requirements analysis: what requirements do the preceding three analyses impose on the design?



Know Your User

- Identify characteristics of target user population
  - – Age, gender, culture, language
  - – Education (literacy? numeracy?)
  - – Physical limitations
  - – Computer experience (typing?)
  - – Motivation, attitude
  - – Domain experience
  - – Application experience
  - – Work environment and other social context
  - – Relationships and communication patterns

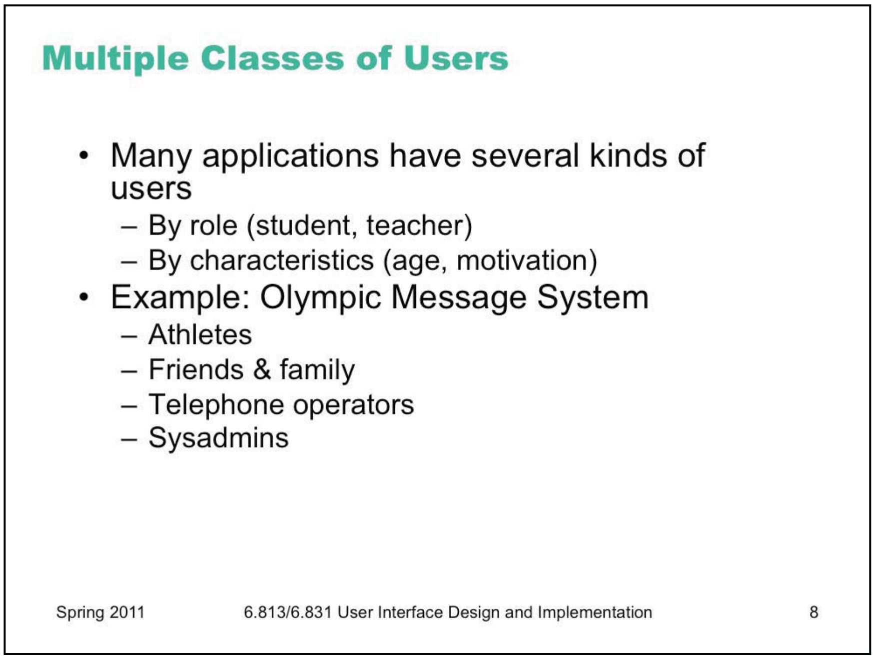
Spring 2011 6.813/6.831 User Interface Design and Implementation 7

The reason for user analysis is straightforward: since you’re not the user, you need to find out who the user actually is.

User analysis seems so obvious that it’s often skipped. But failing to do it explicitly makes it easier to fall into the trap of assuming every user is like you. It’s better to do some thinking and collect some information first.

Knowing about the user means not just their individual characteristics, but also their situation. In what environment will they use your software? What else might be distracting their attention? What is the social context? A movie theater, a quiet library, inside a car, on the deck of an aircraft carrier; environment can place widely varying constraints on your user interface.

Other aspects of the user’s situation include their relationship to other users in their organization, and typical communication patterns. Can users ask each other for help, or are they isolated? How do students relate differently to lab assistants, teaching assistants, and professors?



Multiple Classes of Users

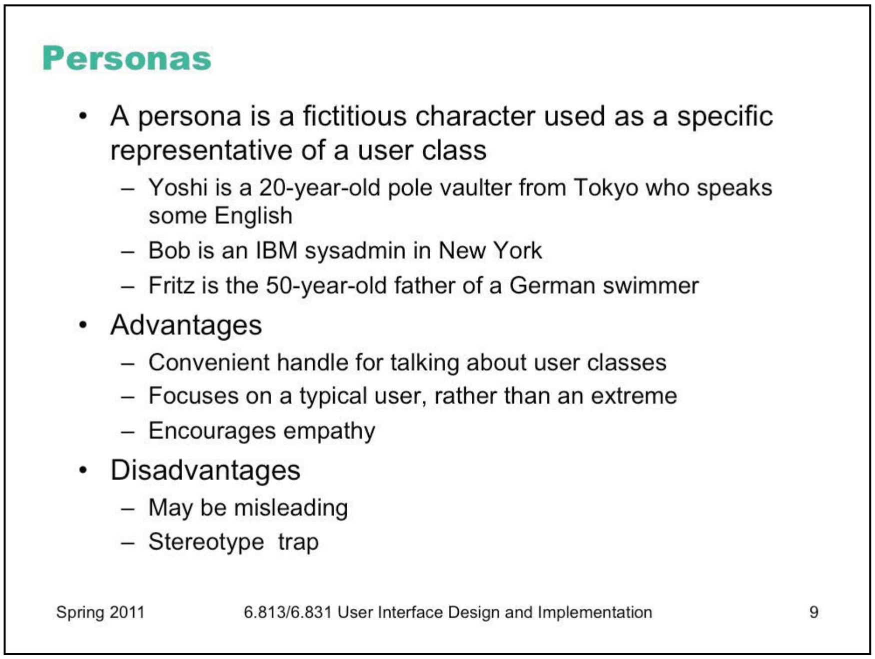
- Many applications have several kinds of users
  - By role (student, teacher)
  - By characteristics (age, motivation)
- Example: Olympic Message System
  - Athletes
  - Friends & family
  - Telephone operators
  - Sysadmins

Many, if not most, applications have to worry about multiple classes of users.

Some user groups are defined by the roles that the user plays in the system: student, teacher, reader, editor.

Other groups are defined by characteristics: age (teenagers, middle-aged, elderly); motivation (early adopters, frequent users, casual users). You have to decide which user groups are important for your problem, and do a user analysis for every class.

The Olympic Message System case study we saw in a previous lecture identified several important user classes by role.



Personas

- A persona is a fictitious character used as a specific representative of a user class
  - Yoshi is a 20-year-old pole vaulter from Tokyo who speaks some English
  - Bob is an IBM sysadmin in New York
  - Fritz is the 50-year-old father of a German swimmer
- Advantages
  - Convenient handle for talking about user classes
  - Focuses on a typical user, rather than an extreme
  - Encourages empathy
- Disadvantages
  - May be misleading
  - Stereotype trap

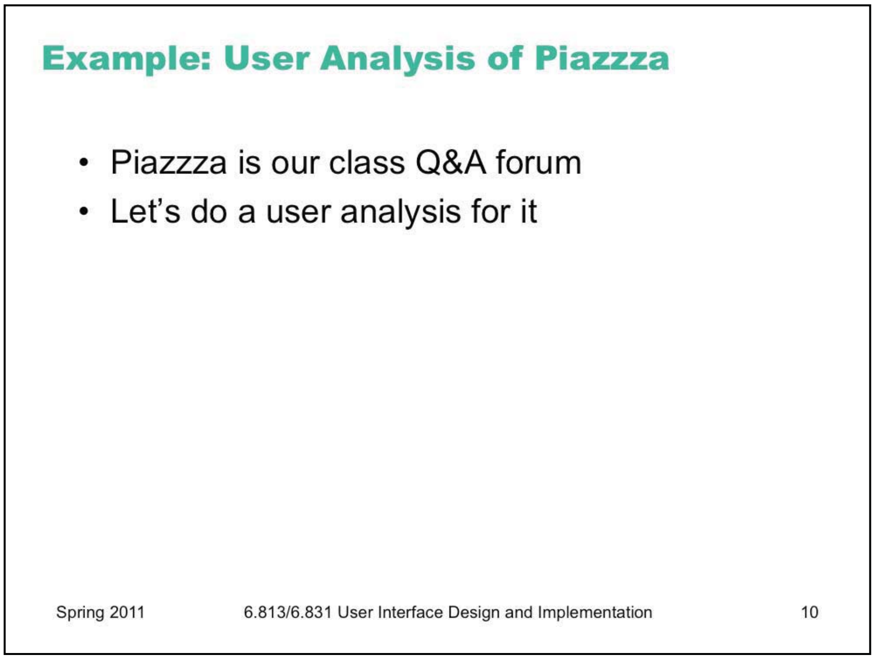
Spring 2011 6.813/6.831 User Interface Design and Implementation 9

One popular technique for summarizing user classes is to give each user class a fictional representative, with typical characteristics and often a little back story. These representatives are called *personas*.

Personas are useful shorthand for a design group; you can say things like “let’s think about how Yoshi would do this”, rather than a mouthful like “non-English-speaking athlete.” They also help focus attention on typical members of the user class, rather than extremes. And by putting a human face on a user class, albeit an imaginary one, they can encourage you to have more empathy for a user class that’s very different from your own. (Alan Cooper, *The Inmates are Running the Asylum*, 1999).

Some cautions: a badly-chosen persona, an extreme case, won't help focus on the typical user. And it's shorthand, so you're abstracting away the richness and diversity of the user class into a specific example. It's not too bad for scalar properties like age or education level, where a mean or median value has some meaning, but what do you do for categorical properties like gender? Say you have a user class that's 35% male and 65% female. Do you use a female persona, at the risk of ignoring the guys? Or do you split it into two user classes, and run the risk of designing for gender differences that really aren't relevant?

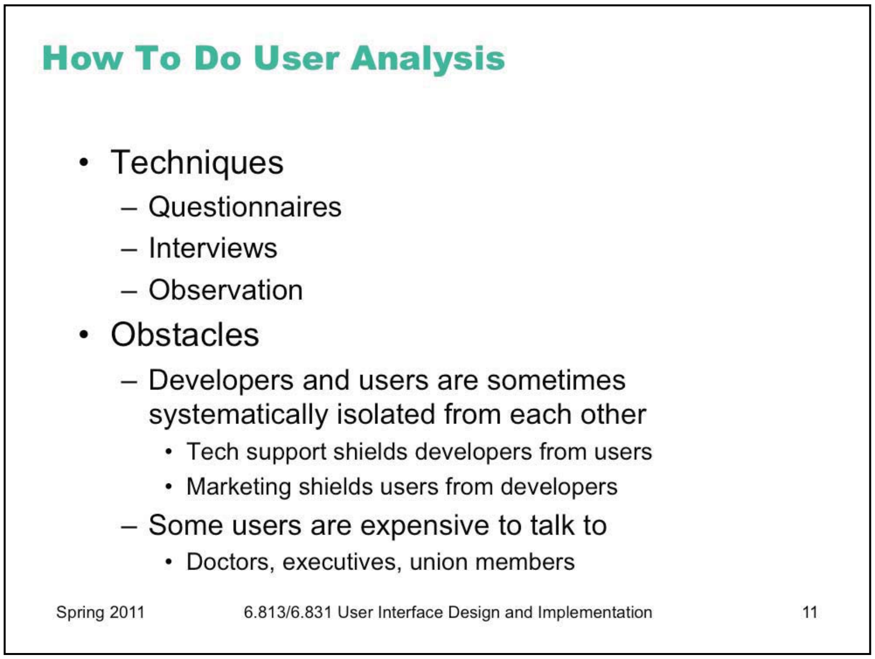
Personas are essentially stereotypes. Technically, you *want* a persona to be a stereotype; it should typify its user class. But we all know the dehumanizing effects of stereotyping, so you also want the persona to be like a human being, an *individual* that you respect and love. So each persona implicitly has two parts: the stereotypical part, and the "color" we added to make it a real person. For example, Franny is an 8-year old child (typical of some user class), who happens to be a girl, lives in Chicago, and likes drawing bunnies and mushroom clouds (but those parts aren't typical for the class, nor do they really matter for our design). Everybody on the design team has to implicitly know what part of the persona matters and what doesn't.



Example: User Analysis of Piazzza

- Piazzza is our class Q&A forum
- Let’s do a user analysis for it

Who are the users in Piazzza?



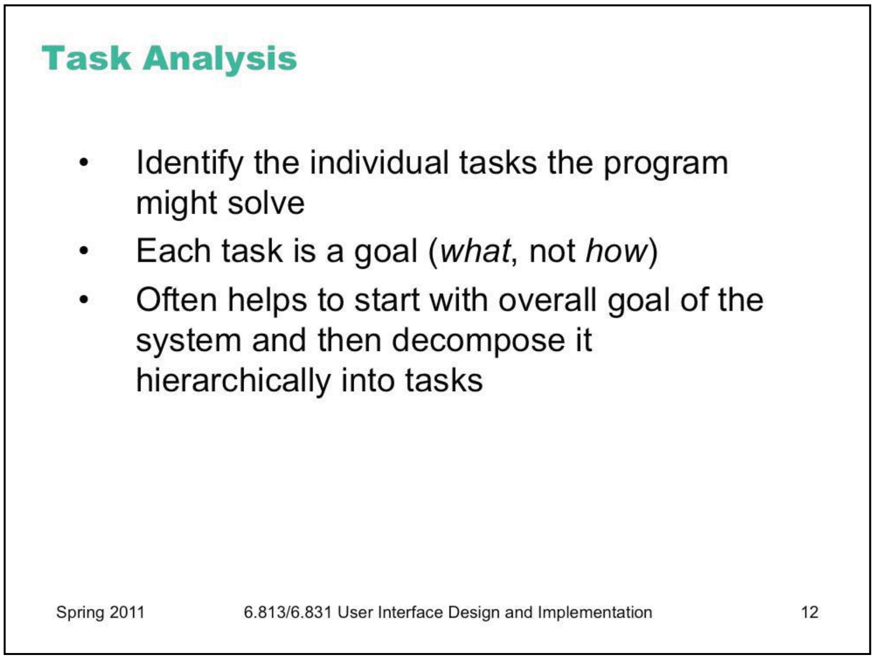
How To Do User Analysis

- Techniques
  - Questionnaires
  - Interviews
  - Observation
- Obstacles
  - Developers and users are sometimes systematically isolated from each other
    - Tech support shields developers from users
    - Marketing shields users from developers
  - Some users are expensive to talk to
    - Doctors, executives, union members

The best way to do user analysis is to find some representative users and talk to them. Straightforward characteristics can be obtained by a **questionnaire**. Details about context and environment can be obtained by **interviewing** users directly, or even better, **observing** them going about their business, in their natural habitat.

Sometimes it can be hard to reach users. Software companies can erect artificial barriers between users and developers, supposedly for their mutual protection. After all, if users know who the developers are, they might pester them with bugs and questions about the software, which are better handled by tech support personnel. The marketing department may be afraid to let the developers interact with the users – not only because geeks can be scary (and sometimes obnoxious), but also because usability discussions may make customers dissatisfied with the current product. ("I hadn't noticed it before, but that DOES suck!") But this isn't a good idea. Developers should interact with users, if only so that they learn that their users are intelligent human beings with real goals, not just idiots who can't find the Any key.

Some users are also expensive to find and talk to. Nevertheless, make every effort to collect the information you need. A little money spent collecting information initially should pay off significantly in better designs and fewer iterations.



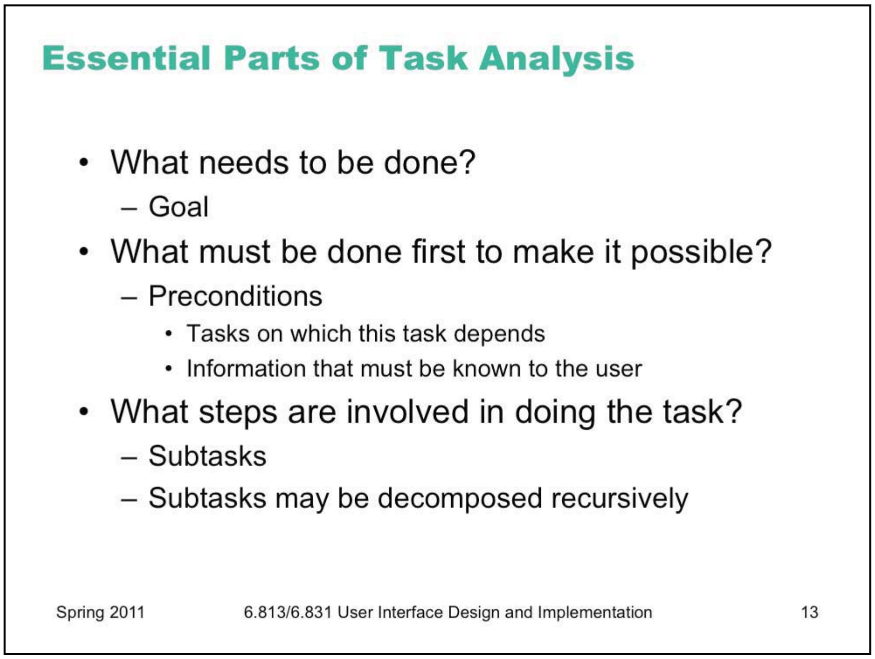
Task Analysis

- Identify the individual tasks the program might solve
- Each task is a goal (*what*, not *how*)
- Often helps to start with overall goal of the system and then decompose it hierarchically into tasks

Spring 2011 6.813/6.831 User Interface Design and Implementation 12

The next step is figuring out what tasks are involved in the problem. A task should be expressed as a goal: *what* needs to be done, not *how*.

One good way to get started on a task analysis is hierarchical decomposition. Think about the overall problem you're trying to solve. That's really the top-level task. Then decompose it into a set of subtasks, or subgoals, that are part of satisfying the overall goal.



Essential Parts of Task Analysis

- What needs to be done?
  - Goal
- What must be done first to make it possible?
  - Preconditions
    - Tasks on which this task depends
    - Information that must be known to the user
- What steps are involved in doing the task?
  - Subtasks
  - Subtasks may be decomposed recursively

Spring 2011 6.813/6.831 User Interface Design and Implementation 13

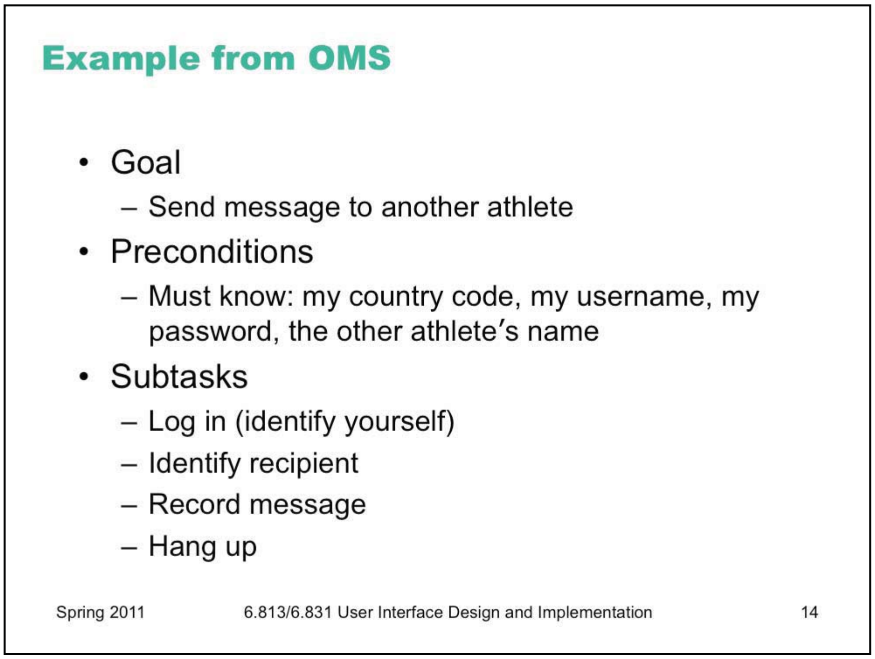
Once you've identified a list of tasks, fill in the details on each one. Every task in a task analysis should have at least these parts.

The **goal** is just the name of the task, like "send an email message."

The **preconditions** are the conditions that must be satisfied before it's reasonable or possible to attempt the task. Some preconditions are other tasks in your analysis; e.g., before you can listen to your messages in the Olympic Message System, you first have to log in. Other preconditions are **information needs**, things the user needs to know in order to do the task. For example, in order to send an email message, I need to know the email addresses of the people I want to send it to; I may also need to look at the message I'm replying to.

Preconditions are vitally important to good UI design, particularly because users don't always satisfy them before attempting a task, resulting in errors. Knowing what the preconditions are can help you prevent these errors, or at least render them harmless. For example, a precondition of starting a fire in a fireplace is opening the flue, so that smoke escapes up the chimney instead of filling the room. If you know this precondition as a designer, you can design the fireplace with an interlock that ensures the precondition will be met. Another design solution is to offer opportunities to complete preconditions: for example, an email composition window should give the user access to their address book to look up recipients' email addresses.

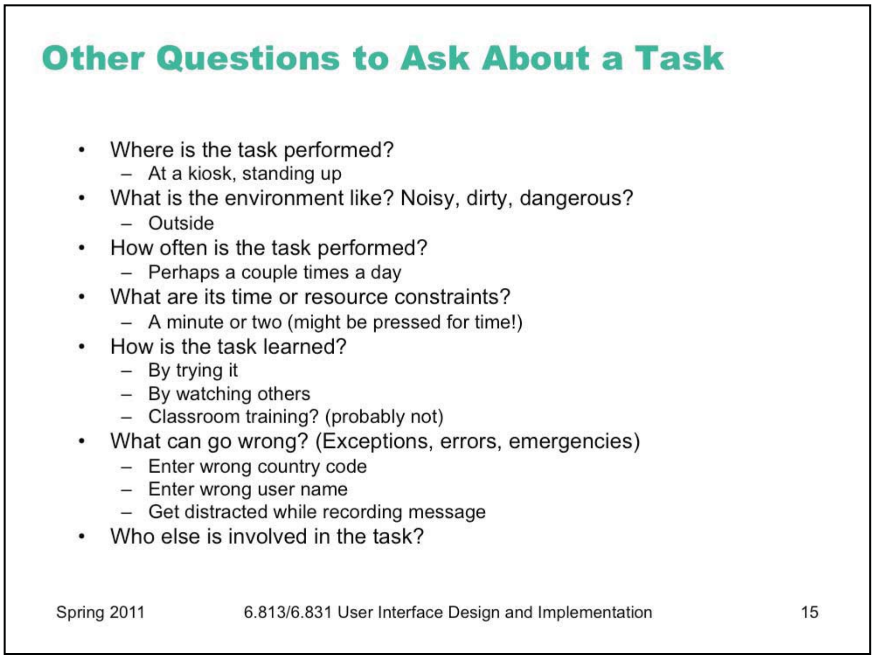
Finally, decompose the task into **subtasks**, individual steps involved in doing the task. If the subtasks are nontrivial, they can be recursively decomposed in the same manner.



Example from OMS

- Goal
  - Send message to another athlete
- Preconditions
  - Must know: my country code, my username, my password, the other athlete′s name
- Subtasks
  - Log in (identify yourself)
  - Identify recipient
  - Record message
  - Hang up

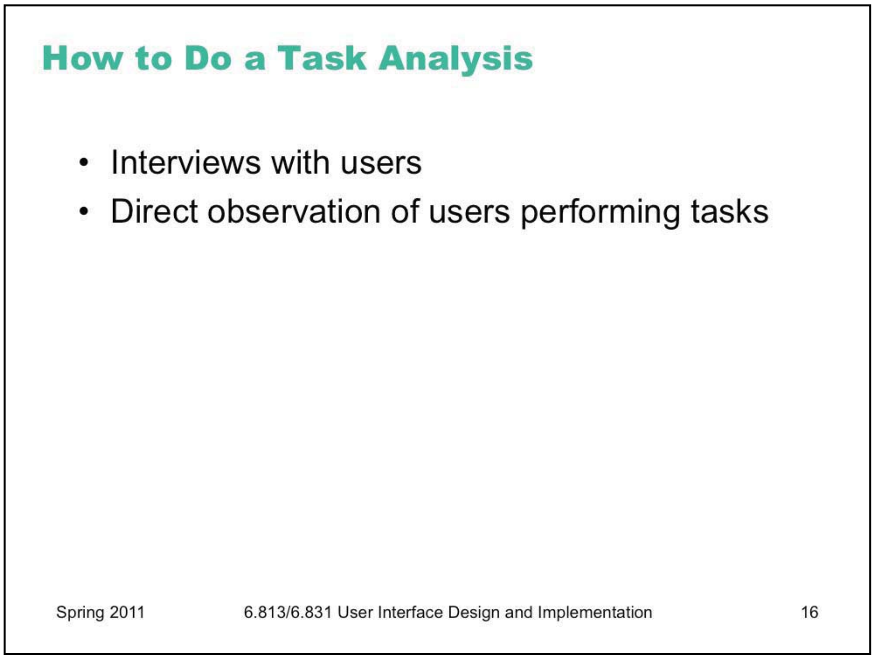
Here’s an example of a task from the Olympic Message System.



Other Questions to Ask About a Task

- Where is the task performed?
  - At a kiosk, standing up
- What is the environment like? Noisy, dirty, dangerous?
  - Outside
- How often is the task performed?
  - Perhaps a couple times a day
- What are its time or resource constraints?
  - A minute or two (might be pressed for time!)
- How is the task learned?
  - By trying it
  - By watching others
  - Classroom training? (probably not)
- What can go wrong? (Exceptions, errors, emergencies)
  - Enter wrong country code
  - Enter wrong user name
  - Get distracted while recording message
- Who else is involved in the task?

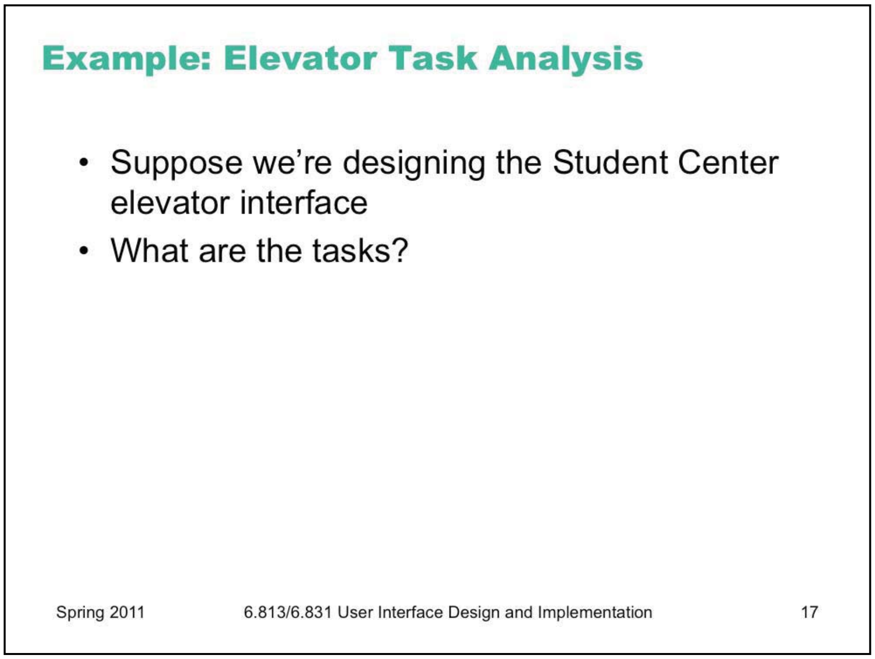
There are lots of questions you should ask about each task. Here are a few, with examples relevant to the OMS send-message task.



How to Do a Task Analysis

- Interviews with users
- Direct observation of users performing tasks

The best sources of information for task analysis are user interviews and direct observation. Usually, you'll have to observe how users *currently* perform the task. For the OMS example, we would want to observe athletes interacting with each other, and with family and friends, while they're training for or competing in events. We would also want to interview the athletes, in order to understand better their goals in the task.



Example: Elevator Task Analysis

- Suppose we're designing the Student Center elevator interface
- What are the tasks?

Spring 2011 6.813/6.831 User Interface Design and Implementation 17

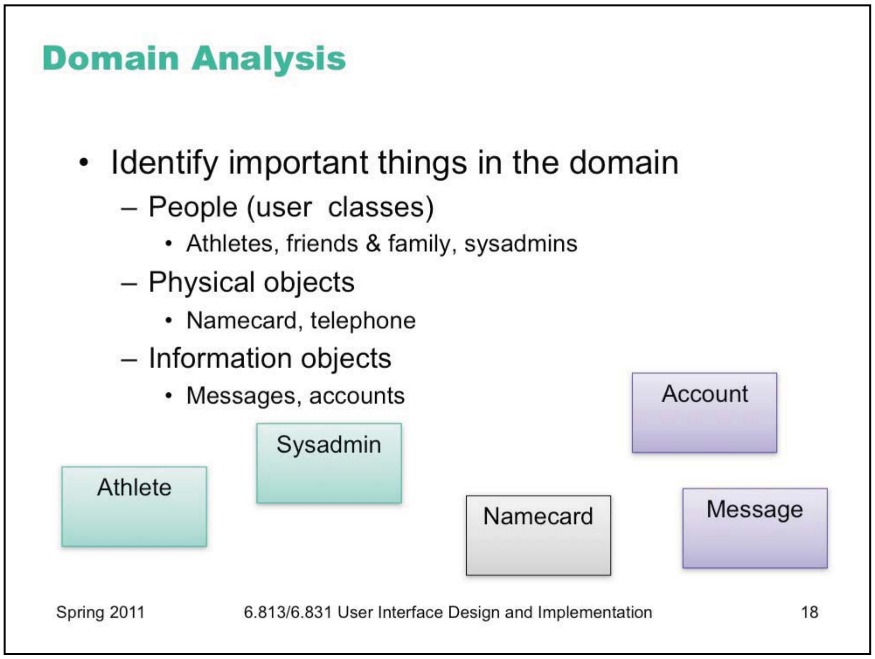
Looking at today's typical use of elevators, the high-level task is:

- **go to a floor**: for example, I want to go to floor 5. Subtasks include:
    - **decide between elevator or stairs**
        - preconditions: knowing how many flights you'd have to climb up or down, and knowing where the elevators are now (are they far away or not moving, so it's worth taking the stairs)
    - **call elevator** to request it to come to you
    - **step in**
        - preconditions: elevator has arrived and the door has opened
        - (we might omit this one from the task analysis, because it seems like an insignificant subtask; on the other hand, the system *is* affected by this action. For one thing, the user may now face a different interface, the one inside the elevator, rather than outside, so it's a significant state change from a UI perspective. For another, the system has sensors that detect when this happens, and doors open and close in order to enable this to happen.)
    - **request floor** (push a button on the panel)
    - exceptional condition: if the elevator stops moving, **call for help**
    - **step out**
        - preconditions: elevator has arrived at destination floor and doors have opened.

These are the tasks for a passenger. The elevator task we're observing is heavily automated, but elevators didn't used to be automated. An elevator with a manual driver might have had these tasks for the driver:

- **drive the elevator**
    - **view calls** (requests from other floors, which helps the driver form a plan about which floors to go to in which order)
        - which floors have calls? how many people and how much stuff are waiting at each floor?
    - **go to floor**
        - **close door**
        - **go up**
        - **go down**
        - **stop**
        - **open door**

This example isn't necessarily arguing to go back to the days of manually-controlled elevators, but user control and freedom is a good thing, and some of this control UI *is* still found in modern elevators, e.g. **open door** and and **close door** buttons. Visibility is also a good thing -- when you step on an elevator, would it be helpful to **view**



Domain Analysis

- Identify important things in the domain
  - People (user classes)
    - Athletes, friends & family, sysadmins
  - Physical objects
    - Namecard, telephone
  - Information objects
    - Messages, accounts

Account

Sysadmin

Athlete

Namecard

Message

Spring 2011 6.813/6.831 User Interface Design and Implementation 18

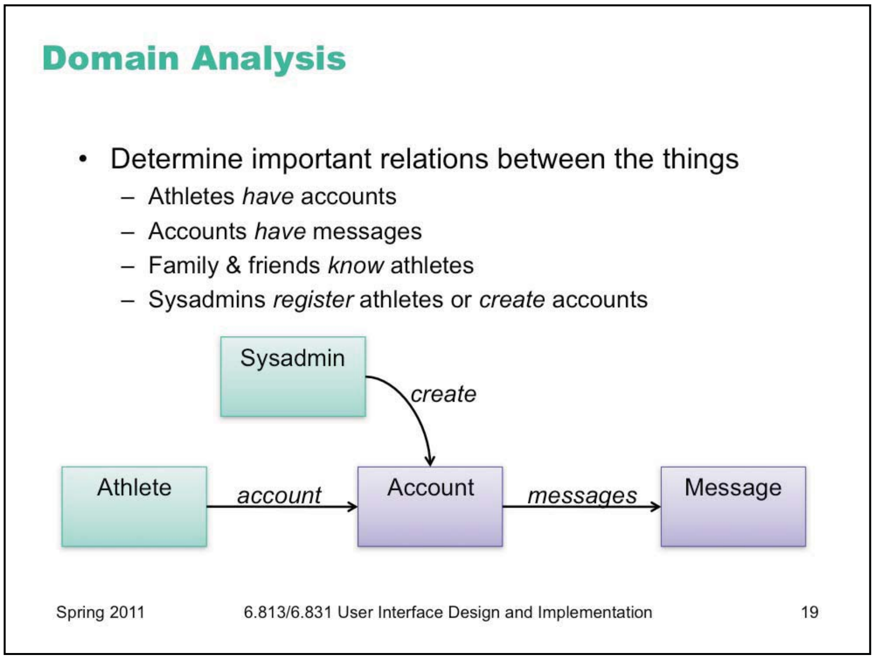
The third step is **domain analysis**, which discovers the elements of the domain and how they're related to each other. If you took 6.170 or a similar software engineering class, you did domain analysis by drawing object model diagrams or entity-relationship diagrams. That's what we'll do too.

To draw a domain diagram, you first need to identify the **entities** of the domain – the *things* that are involved. Entities include people, physical objects, and information objects. (An information object is something that consists only of bits. You can't store a chair digitally, but you can store a slide, or a memo, or a purchase order, or an email message, or an account. When you translate your domain diagram into a software system design, you might create classes or database tables that represent people and physical objects in your domain. But at the level of domain analysis, you're not designing the software yet, and people and physical objects are still physical, not digital.)

User classes defined by role should certainly be entities; user classes defined by characteristics generally aren't, because variation in characteristics doesn't change how they fundamentally relate to other entities. In the OMS example, *athletes* have namecards and *sysadmins* don't (role-based classes); but *young* or *male* isn't a useful distinction (characteristic-based classes).

Sometimes you need to include people in the domain model that you haven't identified as user classes, because they're involved in the system but aren't actually users of the interface you're designing. For example, an IM client needs to represent Buddies, but they aren't a user class. A hospital information management system needs to represent Patients even if they won't actually touch the UI.

Draw each kind of entity as a labeled box, as shown here. Note that you should think about these boxes as representing a *set* of objects – so the Athlete box is the set of all athletes in OMS.



Domain Analysis

- Determine important relations between the things
  - Athletes *have* accounts
  - Accounts *have* messages
  - Family & friends *know* athletes
  - Sysadmins *register* athletes or *create* accounts

Sysadmin

create

Athlete

account

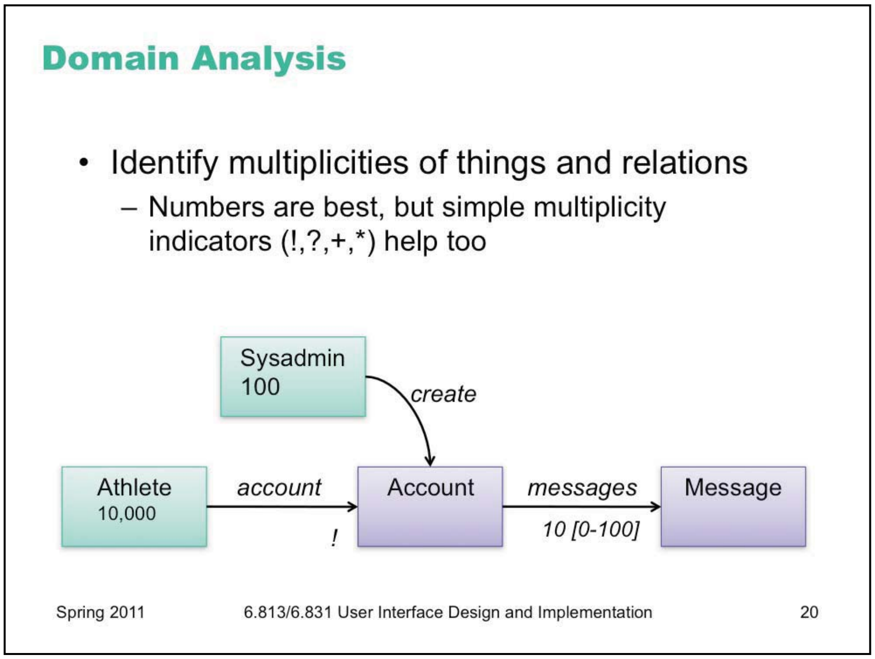
Account

messages

Message

Next, determine the **relationships** between the entities that matter to your problem, and draw them as edges. Here are some examples from the OMS. Relationships are usually labeled as verbs (create, know), but with generic relationships like "have" (also called "has-a"), it's more readable to label it with a noun, analogous to a field or property name (the athlete's "account", the account's "messages").

Another kind of relationship is classification, or "is-a". You can use this to show that several entity classes are subclasses of a larger one – e.g., Lecturers and TAs are subclasses of Instructors.



Domain Analysis

- Identify multiplicities of things and relations
  - Numbers are best, but simple multiplicity indicators (!,?,+,*) help too

Spring 2011 6.813/6.831 User Interface Design and Implementation 20

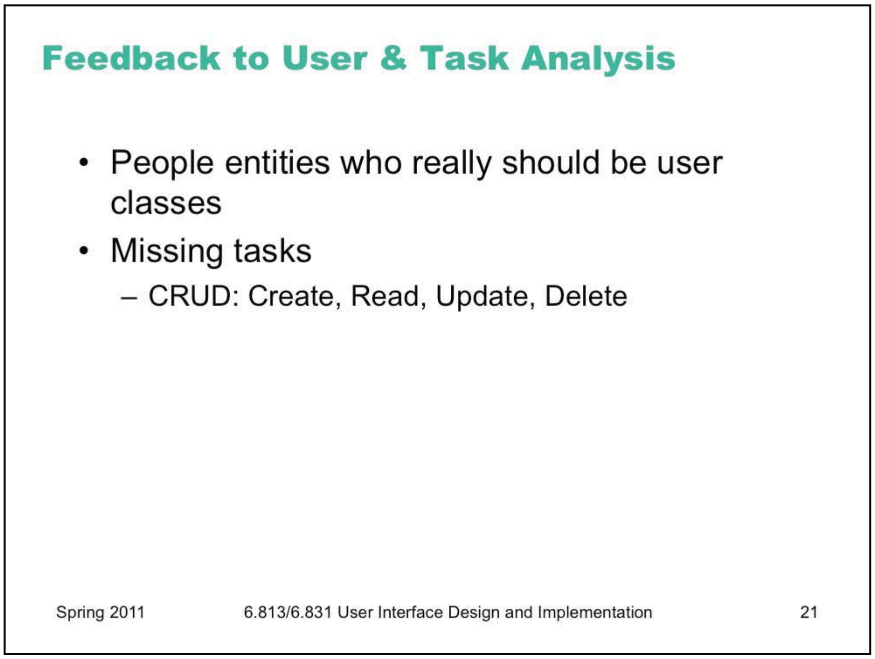
Finally, add **multiplicities** showing the sizes of entity sets and relationships. The multiplicity of an entity set is the number of members of that set in the system; for example, there were about 10,000 athletes in OMS. The multiplicity of a relation is the number of targets per source; for example, an athlete has exactly one account, and each account might have 10 messages stored in it.

Why do we want multiplicities? They will be useful in later design, both for thinking about your backend (10 million messages have to be stored much differently than 100 messages) and your user interface (10,000 messages must be displayed and manipulated much differently than 10 messages).

If there's some uncertainty in your estimate, or if the multiplicity will vary in actual use, then show a typical value plus a range. In the diagram here, we're showing that we expect a typical account to have 10 messages, with as few as 0 messages but possibly as many as 100. The best multiplicity estimates are based on actual data or observation. If you're building an email system, for example, find out how large users' inboxes actually *are*, by measuring existing email practices.

Note that you don't have to mark every multiplicity, because many of them can be deduced from other multiplicities. For example, what is the multiplicity of the Message set, given the other information on the diagram?

You can abbreviate common multiplicities with symbols: **!** means exactly 1, **?** means 0 or 1, **+** means 1 or more, and ***** means 0 or more. Use **+** and ***** when more precise estimates aren't likely to change the system design, either the backend or the UI.



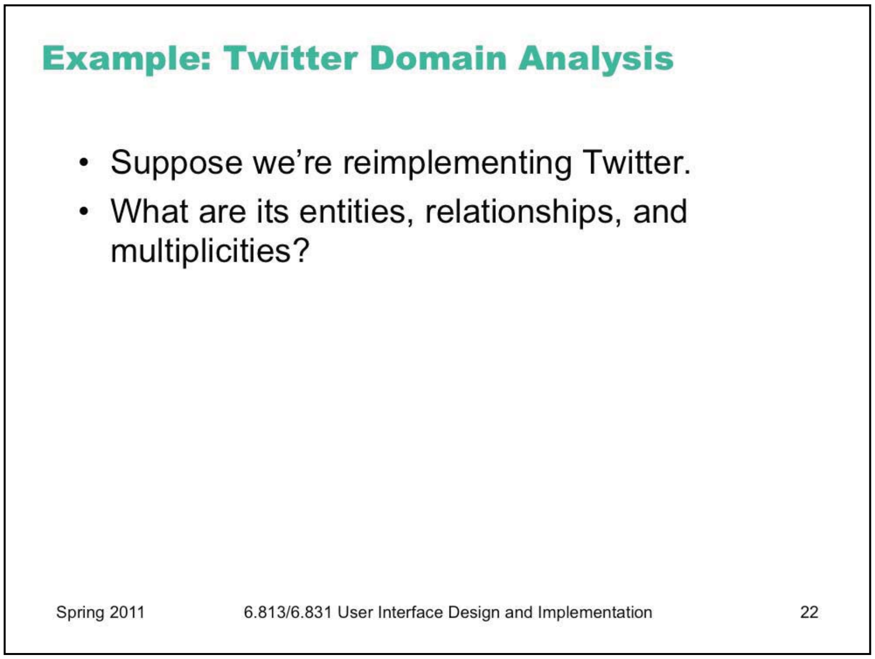
Feedback to User & Task Analysis

- People entities who really should be user classes
- Missing tasks
  - CRUD: Create, Read, Update, Delete

After doing some domain analysis, you can use it to think about whether your user and task analysis was complete. For example, you may have identified new people entities who really should be user classes of your system. Maybe Patients *should* be users of the hospital information system, and a user interface should be created that supports their tasks.

Your domain analysis may have also identified physical or information objects that don't seem to be involved in any of the tasks you specified. That could be a sign that your domain analysis is broader or more detailed than you really need, or it could be a sign that you missed some tasks.

One heuristic that you can use for information objects is CRUD. For every information object, consider whether you need low-level tasks for Creating, Reading (viewing information about), Updating (changing the information), and Deleting the objects. Consider Messages in the OMS example: friends and family need to record messages (Create), athletes need to listen to them (Read), and athletes need to delete them (Delete). We're missing the Update task. Maybe that's because messages should be immutable, like face-to-face speech is; once something comes out of your mouth, you can't modify it. But maybe editing a message is something we could consider in the design. In any case, by checking for CRUD, we might find tasks we didn't observe in the task analysis.

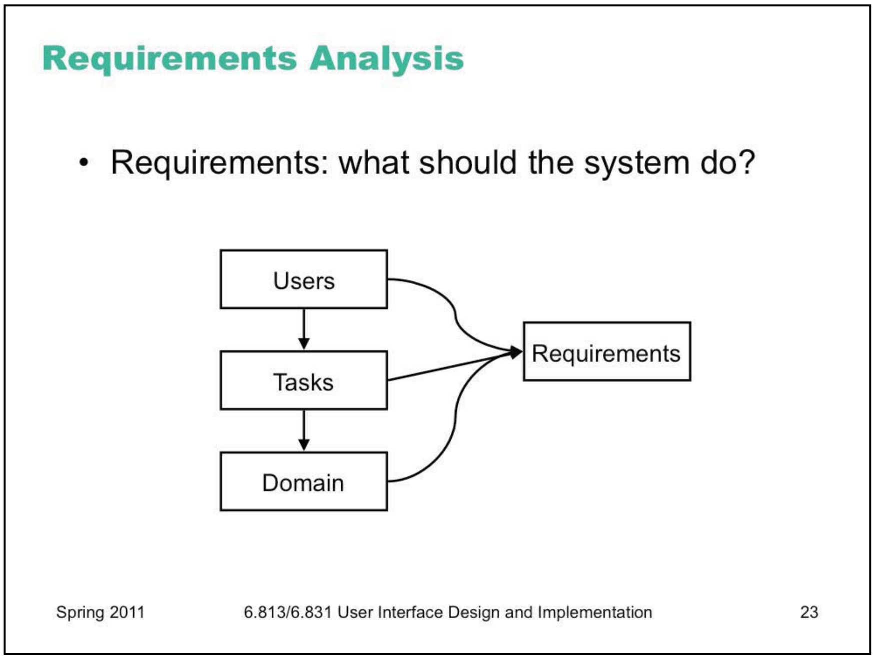


Example: Twitter Domain Analysis

- Suppose we're reimplementing Twitter.
- What are its entities, relationships, and multiplicities?

Clearly **Tweets** are an entity. Also **Users**, and their **Accounts**.

Relationships include Account **says** Tweet, and Account **follows** Account.



Requirements Analysis

- Requirements: what should the system do?

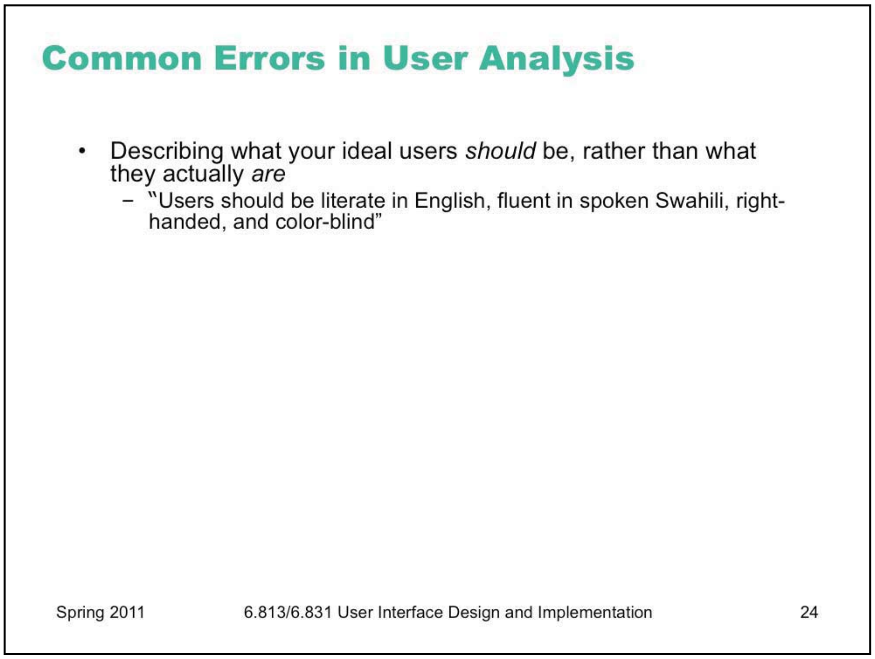
User, task, and domain analysis feed into a more general process called **requirements analysis**, which creates a description of the system's desired functionality and other nonfunctional properties (like performance, security, and capacity).

Without user and task analysis, requirements are incomplete. User and task analysis contribute additional functionality (tasks that users need to do which may not be evident from the domain analysis alone) as well as nonfunctional requirements about **usability** (like how efficient certain tasks should be, or how learnable, or how memorable) and about other properties of the system as well (e.g., accommodation for users' physical limitations, like impaired vision). For example, here are some of the requirements in the OMS system that might come out of user and task analysis:

•Support twelve languages (because athletes, friends & family don't all speak the same language)

•Support non-touchtone phones (because friends & family don't all have them)

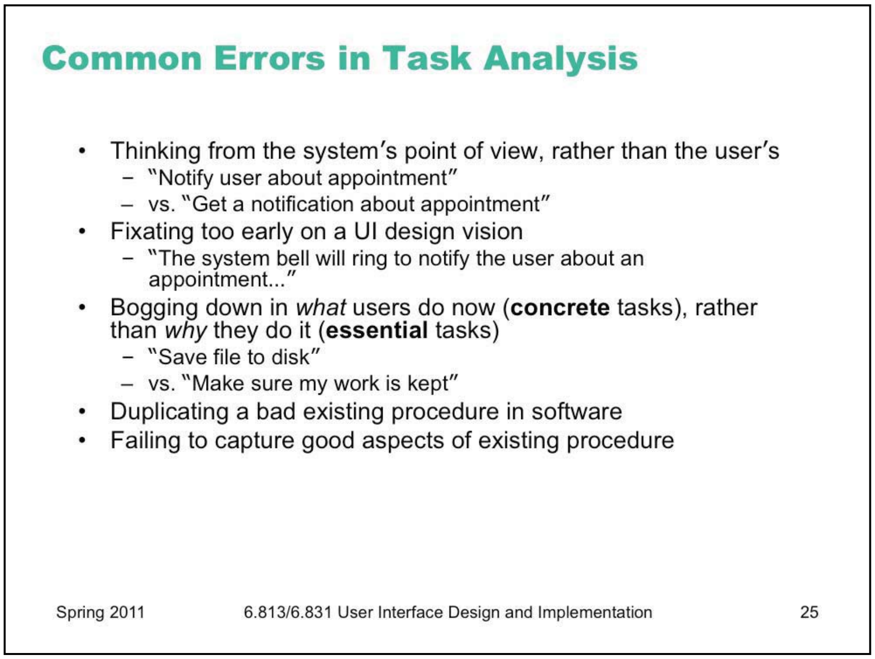
•Check Messages task should take less than 30 seconds (because athletes may be pressed for time)



Common Errors in User Analysis

- Describing what your ideal users *should* be, rather than what they actually *are*
  - “Users should be literate in English, fluent in spoken Swahili, right-handed, and color-blind”

Many problems in user and task analysis are caused by jumping too quickly into a requirements mindset. In user analysis, this sometimes results in wishful thinking, rather than looking at reality. Saying “OMS users *should* all have touchtone phones” is stating a **requirement**, not a characteristic of the existing users. One reason we do user analysis is to see whether these requirements are actually satisfied, or whether we’d have to add something to the system to make sure it’s satisfied. For example, maybe we’d have to offer touchtone phones to every athlete’s friends and family…



Common Errors in Task Analysis

- Thinking from the system′s point of view, rather than the user′s
  - "Notify user about appointment"
  - vs. "Get a notification about appointment"
- Fixating too early on a UI design vision
  - "The system bell will ring to notify the user about an appointment..."
- Bogging down in *what* users do now (**concrete** tasks), rather than *why* they do it (**essential** tasks)
  - "Save file to disk"
  - vs. "Make sure my work is kept"
- Duplicating a bad existing procedure in software
- Failing to capture good aspects of existing procedure

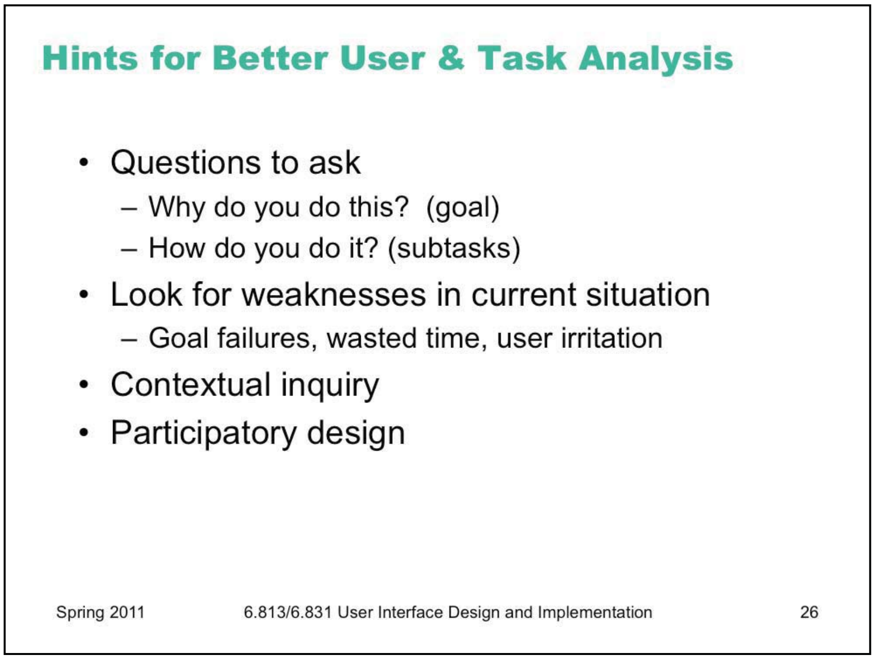
Spring 2011 6.813/6.831 User Interface Design and Implementation 25

The requirements mindset can also affect task analysis. If you're writing down tasks from the system's point of view, like "Notify user about appointment", then you're writing **requirements** (what the system should do), not **tasks** (what the user's goals are). Sometimes this is merely semantics, and you can just write it the other way; but it may also mean you're focusing too much on what the system *can* do, rather than what the user *wants*. Tradeoffs between user goals and implementation feasibility are inevitable, but you don't want them to dominate your thinking at this early stage of the game.

Task analysis derived from observation may give too much weight to the way things are currently done. A task analysis that breaks down the steps of a current system is **concrete**. For example, if the *Log In* task is broken down into the subtasks *Enter username* and *Enter password*, then this is a concrete task relevant only to a system that uses usernames and passwords for user identification. If we instead generalize the *Log In* task into subtasks *Identify myself* and *Prove my identity*, then we have an **essential** task, which admits much richer design possibilities when it's time to translate this task into a user interface.

A danger of concrete task analysis is that it might preserve tasks that are inefficient or could be done a completely different way in software. Suppose we did a task analysis by observing users interacting with paper manuals. We'd see a lot of page flipping: "Find page N" might be an important subtask. We might naively conclude from this that an online manual should provide really good mechanisms for paging & scrolling, and that we should pour development effort into making those mechanisms as fast as possible. But page flipping is an artifact of physical books! It would pay off much more to have fast and effective searching and hyperlinking in an online manual. That's why it's important to focus on **why** users do what they do (the essential tasks), not just what they do (the concrete tasks).

An incomplete task analysis may fail to capture important aspects of the existing procedure. In one case, a dentist's office converted from manual billing to an automated system. But the office assistants didn't like the new system, because they were accustomed to keeping important notes on the paper forms, like "this patient's insurance takes longer than normal." The automated system provided no way to capture those kinds of annotations. That's why interviewing and observing real users is still important, even though you're observing a concrete task process.



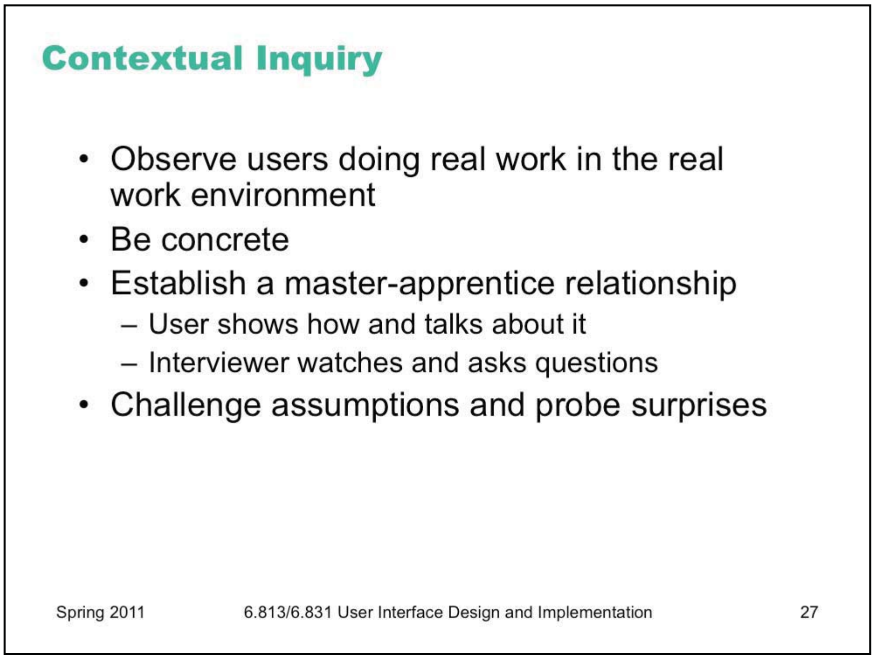
Hints for Better User & Task Analysis

- Questions to ask
  - Why do you do this? (goal)
  - How do you do it? (subtasks)
- Look for weaknesses in current situation
  - Goal failures, wasted time, user irritation
- Contextual inquiry
- Participatory design

When you're interviewing users, they tend to focus on the what: "first I do this, then I do this…" Be sure to probe for the **why** and **how** as well, to make your analysis more abstract and at the same time more detailed.

Since you want to improve the current situation, look for its weaknesses and problems. What tasks often fail? What unimportant tasks are wasting lots of time? It helps to ask the users what annoys them and what suggestions they have for improvement.

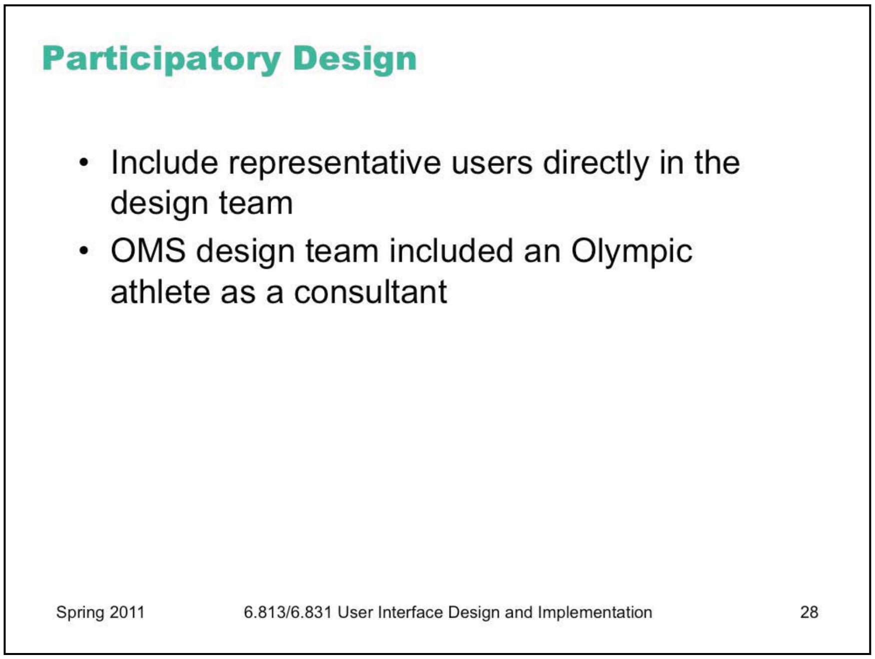
There are two other techniques for making user and task analysis more effective: contextual inquiry and participatory design, described in more detail on the next slides.



Contextual Inquiry

- Observe users doing real work in the real work environment
- Be concrete
- Establish a master-apprentice relationship
  - User shows how and talks about it
  - Interviewer watches and asks questions
- Challenge assumptions and probe surprises

Contextual inquiry is a technique that combines interviewing and observation, in the user’s actual work environment, discussing actual work products. Contextual inquiry fosters strong collaboration between the designers and the users. (Wixon, Holtzblatt & Knox, “Contextual design: an emergent view of system design”, CHI ’90)

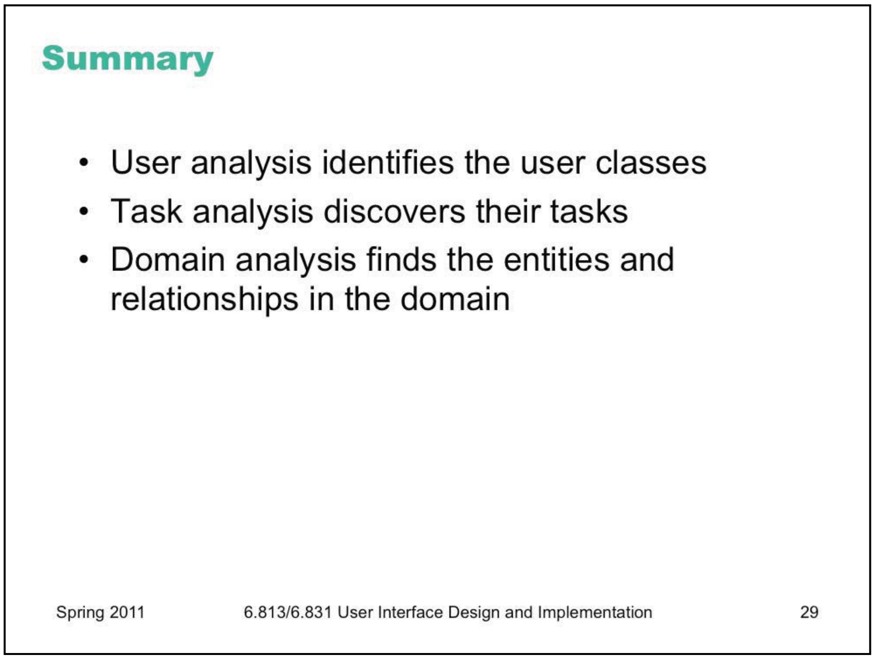


Participatory Design

- Include representative users directly in the design team
- OMS design team included an Olympic athlete as a consultant

Spring 2011 6.813/6.831 User Interface Design and Implementation 28

Participatory design includes users directly on the design team – participating in the task analysis, proposing design ideas, helping with evaluation. This is particularly vital when the target users have much deeper domain knowledge than the design team. It would be unwise to build an interface for stock trading without an expert in stock trading on the team, for example.



Summary

- User analysis identifies the user classes
- Task analysis discovers their tasks
- Domain analysis finds the entities and relationships in the domain