The UNIVERSITY of VERMONT

Benjamin Fenster
CS-205 (Damon)
20 December 2004

Design

I. User Interface Design
   a. Introduction
      i. Since the user is central, so too should the User Interface
      ii. Need to consider UI all along, can't just slap a GUI on at the end
      iii. Presenting a view of how the world works! Should match how it actually works.
   b. Goals
      i. Ease of Learning
         1. Aimed mostly at new users.
         2. For complex software, needs to be easy to learn new features
         3. For a monthly task the user may *forget* how it was done last month – needs to be easy to "relearn"
         4. Affordances: An affordance tells you how to do something
            a. Architectural Example: Style of door handle tells you whether you need to push, pull, or turn
            b. In Software: Almost anything that looks like a real-world object gives you an immediate sense of how it's supposed to work
         5. Consistency: Apply design concepts from other applications
            a. "Open" goes on the "File" menu
            b. "Properties" goes at the bottom of the context menu
            c. Use consistent fonts
         6. Clear Explanations (in tool tips, et cetera)
         7. Metaphors (e.g. "Desktop"). Get a Metaphor that models how the application works, even to non-geeks
      ii. Efficiency of Use
         1. Tradeoff with "Ease of Learning" Not always possible to have both
         2. Need to know a lot about your users
         3. Create minimal effort actions
         4. Low mental load
      iii. Aesthetics
         1. Helps sales, users feel better
         2. Read Tufte (Urban Planning professor at Yale)
         3. "High Guru of Information Presentation"
   c. Cognitive Background
      i. "Model Human Processor" (empirically derived, 1980)
      ii. Idea: The human mind is like a computer.
      iii. Inputs: Eyes, Ears (Perception)
      iv. Perceptual
         1. Only auditory and visual
         2. Information is handled in "chunks" of variable "size"
         3. Visual
            a. Can remember about 17 images for about $^{1}/_{5}{}^{th}$ second each
            b. Movement and gross patterns perceived first
            c. Broad special sense
            d. Visual information encoded in chunks (color, size, texture, even *font* – can tell newspaper from a distance from its title, letters)
         4. Auditory
            a. About 5 sounds, 1.7 seconds each
            b. Filter out common repeated sounds (e.g. a nearby subway)
      v. Cognitive
         1. Thinking
         2. Pattern matching (match chunks to patterns in long-term memory)
         3. Takes about 70 nanoseconds
      vi. Working Memory

        1. “Registers”
        2. Store about 7 chunks in working memory
        3. Like DRAM. If you think about it, it lasts longer. Without thought, lasts about 7 seconds.
    vii. Motor
        1. “Muscle Memory” Don’t need to provide complex commands once the muscles know how to do the little things.
        2. Can send “higher-level” commands. Don’t have to control each tiny detail anymore.
    viii. See [CS-295-2005-1-LECTURE] from next semester
d. Concerns
    i. Big Thing: Limit to 7 “chunks” (plus, the user’s probably devoting some attention elsewhere so you don’t even get the full 7 chunks)
    ii. Six degree focal area, or about 2 inches of the screen
    iii. Group common items together.
    iv. Help in chunking. Group related things together.
    v. Colors and fonts are perceived for free – use color and font consistently.
    vi. Avoid flashing. Each flash generates an extra chunk!
    vii. Sounds that are repeated often may be ignored
    viii. Stick to sequences. User can match the old pattern.
    ix. Keep things in the same place on the screen (muscle memory)
    x. Start from User Tasks
        1. Not the same as Use Cases, which are a single sequence of actions
        2. Tasks have branches and decisions
        3. Layout controls based on tasks
e. Prototypes
    i. Too expensive to evaluate an actual application
    ii. Want to develop a UI prototype
    iii. Wizard of Oz Prototype: Some human being is controlling what happens
    iv. High Fidelity
        1. Expensive to create a prototype in the real language, but can reuse code
        2. Cheap to use something like VB but nothing’s really reusable.
    v. Low Fidelity
        1. Down to hand drawn pictures of the application
        2. Keeps evaluators focused on the overall design, not on details like font
        3. Get the feel of whether you need to click 100 times to accomplish a task
f. Evaluation
    i. Do it Yourself
        1. *Always* do this first.
        2. Helps identify really aggravating stuff. If you don’t like it, nobody will.
        3. Doesn’t help at all with “Ease of Learning”
    ii. Colleague Evaluation
        1. First step in discovering ease of learning.
        2. Don’t even consider showing it to a customer if you’re embarrassed to show it to a co-worker
    iii. Real Users
        1. Mostly done for brand new users / ease of learning evaluation
        2. Sometimes experts are used to test new versions.
        3. Good to have pairs of users so you can hear their conversation. Then if something’s confusing you get to hear *why* (how did the user think it would be done).
        4. Want to know *why* she clicked the wrong thing so you’ll know what to fix.
        5. Don’t use pairs who know each other since you won’t get the information you need. You’ll get stuff like, “This is just how X worked, remember?”
    iv. Heuristic Evaluation

1. Just a checklist of issues to consider. The more heuristic evaluations you've done the more stuff you're likely to find.
2. Simple and Natural Dialog. Keep the focus on the real flow of the task.
3. Graphic Design & Color
    a. Consistency!
    b. Data-to-Ink Ratio. W ant lots of information for a given amount of "ink." Less is more: Ditch the clutter
4. Speak the User's Language
    a. In a specialized domain, use the right terminology
    b. For everyday people use everyday language
    c. Phrase everything from the user's perspective. Say, "You have bought…" Don't say, "We have sold you…"
    d. Consistency. If you say "Turn On" then say "Turn Off," not "Deactivate"
5. Minimize User's Memory Load. Only 7 items!
6. Consistency
    a. Visual
    b. Textual
    c. Conceptual. Keep similar activities looking similar.
7. Feedback! Meaningful, immediate
8. Clearly Marked Exits
    a. Let the user escape easily *whenever* s/he wants.
    b. Always, always, always offer a Cancel.
9. Provide Shortcuts. Helps experienced users.
10. Good Error Messages.
    a. What was wrong? How to fix it?
    b. Allow easy recovery from simple errors. (Opera erases the URL if you type it wrong – way to drive users mad!)
11. Prevent Errors
    a. Don't use modes! hard to figure out what mode you're in.
    b. Make it impossible to create an error if you can.
12. Help & Documentation. Much better now with context sensitive help.

v. GOMS
1. Extremely different.
2. Absolutely no feedback on ease of learning or aesthetics.
3. Predicts the actual time it will take an expert to do a tas.
4. Goals
    a. What do you want? To delete a word.
    b. Goal is: "I want to delete a word."
5. Operations
    a. Click on a spot
    b. Press a key
    c. Goals at one level can be used as operations at the next level
6. Methods. Sequence of operations

vi. Selection rules.
vii. Have empirical data for how long it takes to do something. Just add up those times based on all the required actions.
viii. Works really well, actually.

II. System Design
a. Overall design of the system. Not code design!
b. Not even looking at the class level. Worried about larger *components*.
c. Goals
    i. What are we building?
    ii. Once you've identified the components, have a first cut at isolating tasks to individual developers.
d. Good System Designs

    i. Easy to implement
    ii. Changes (even broad changes) may be easy
    iii. Can use as much existing code as possible.
    iv. Your code will be easy to reuse in the future
    v. Needs to be easy to understand
    vi. Most performance issues come from design. Address performance constraints!
    vii. Limit with respect to external dependencies (e.g. Oracle release version)

e. Abstractions
    i. Whole purpose is to have something understandable. Need to abstract to the point of understanding the program as a whole.
    ii. Need to abstract to the point of understanding the program as a whole.
    iii. "Too many details" is just as bad as "too many"
    iv. We *don't* want an approximation. Want to be perfectly accurate about everything you say, just don't say all the details.
    v. Abstraction is *not* the design – the design is too huge. Just helps with understanding the design.
    vi. Design Abstractions (broadly useful)
        1. Functional Structure
            a. Break the problem into pieces. Boxes and arrows.
            b. How do the pieces interact?
            c. Establishes clear borders that become the interfaces between components later.
            d. Each person understands everything about one component; now they can see the interactions
        2. Data Flow
            a. When there's a sequence of steps that are chained together (payroll, compilers), we want to focus on those flows
            b. Works best when there's lots of the same kind of data (e.g. "stream like")
            c. Want multiple processing steps. "Sort" isn't a good step
            d. May have multiple streams coming together
            e. Highlights possible bottlenecks
        3. Data Structure
            a. Processing components interact via central data
            b. Database applications, word processors
            c. Compiler may create a parse tree, then modify it in each step.
            d. Large data structure/s, smaller components to access and modify the data.
            e. Helps reveal issues (bottlenecks, constraints) on how data will be used and modified
        4. Communication
            a. Different components communicating
            b. Example: UI events
            c. Several related abstractions here
                i. Who's talking to whom?
                ii. Flow of information
                iii. Order of messages
            d. Good for understanding protocol
            e. Is there a communications bottleneck?
            f. Figure out what components need to remember / understand
    vii. Using Abstractions
        1. Use annotations / labels / et cetera
        2. Don't put so much information that it becomes useless
        3. Each abstraction gives one view of the final product
        4. Might have a couple hundred pages of small (2-page) abstractions
        5. [CS-205-2004-3-LECTURE-08:29], [CS-205-2004-3-LECTURE-08:30]

f. Combining Designs
    i. Top Down. At each level have a coherent whole
    ii. Bottom up
        1. Start from the necessary low-level components.
        2. You know what you've got to work with
        3. With top-down may end up with some necessary low-level components that don't seem to fit in anywhere
    iii. Do both! Build the components you know you'll need and fit them into the top-level design as you create it.
    iv. Designers rarely invent new design paradigms. Will use pieces of existing designs or the design from the old version of the same product.
g. Architectural Styles
    i. Only a handful. Designers typically just work within a single style.
    ii. Describe how the components fit together
    iii. Big push to publish a catalog of existing designs
    iv. Pennsylvania Standard Bridge Design
        1. Only about 25 designs for bridges to cross little gullies.
        2. Find existing designs based on the physical characteristics of the area
    v. Mary Shaw initiated the push to build a similar catalog for software
    vi. More formal than traditional system design
    vii. Boxes and arrows → Components and connectors
    viii. Components: Black box with public interface, free network part, or whatever
    ix. Analysis
        1. Formal expression allows automated analysis
        2. Can identify deadlocks, bottlenecks, et cetera
        3. For a specific question of importance, this makes sense. In general it's probably not worth it.
    x. Boundaries between styles aren't always clear – some overlap
    xi. Different styles can solve the same problem
h. Some Architectural Styles
    i. Pipe and Filter
        1. Classic UNIX Shell Programming
        2. Components accept inputs, spit out outputs
        3. Connectors are simple – sequential
        4. Components know nothing about one another
        5. data → grep → sort → uniq → awk
        6. Advantages
            a. Easy to understand
            b. Very strong encapsulation – easy to maintain
            c. Wide reuse of components
        7. Disadvantage
            a. Not good for performance at all
            b. Some problems just aren't meant for this design
        8. Can reuse formalization of each component
            a. Thus can prove behaviors
            b. Can analyze throughput
    ii. Layered Systems
        1. Example: Network layers
        2. Advantages: Clean separation of layers, like pipe and filter
        3. Disadvantages
            a. Performance is terrible
            b. The boundary between layers can change over time.
    iii. Repositories
        1. Have a central data store
        2. All transactions are done through the data store

3. Advantages: Performance! One of the few styles for which performance is an advantage
4. Disadvantages
    a. Data store needs to be static
    b. Everybody needs to use the same database

iv. Interpreters
1. Can you treat the series of steps in the application as a language?
2. Search becomes a finite automaton to process data
3. Photoshop applies this to editing photos
4. Creating a Virtual Machine!
5. Advantages
    a. Pretty good performance overall
    b. Can add new features easily
    c. Likewise easy to add high-level tasks
    d. Can wrap existing code as low-level instructions
6. Disadvantages
    a. Costs a lot to get started
    b. Really hard to test – all compilers are

v. Implicit Invocation
1. Use events to communicate
2. Keep all actors anonymous (can't depend on each other that way)
3. May be asynchronous
4. Example: Tools that "watch" for changes in data and take some action
5. Advantages: Flexible, maintainable
6. Disadvantages
    a. Performance (can't share partial data)
    b. Can't make guarantees for individual components (don't know what other players are involved)

vi. Communicating Processes

vii. Client / Server
1. Everybody knows who the server is
2. Server knows nothing about who's out there until they connect
3. Advantages
    a. People "get" this style easily
    b. Single point of control (though may establish a set of servers that become "communicating processes")

viii. Stateless Client / Server
1. Don't remember anything once the transaction's done
2. Advantage: Easy error recovery
3. Disadvantages
    a. Very hard to maintain state
    b. Hard to add new features (new encoding)

ix. Reactive System
1. Advantages: Easy to understand
2. Disadvantage: Not widely used
3. Very strong analysis ability. Can prove stuff

x. Data Abstraction
1. Coding style, not architectural style
2. Basically a mistake that it was ever included.

xi. There are many other styles.

xii. If you know the styles, something may seem like the obvious answer to a particular problem. Skim through the catalog and see!

III. Sharing Designs

a. Need to communicate details of your beautiful design.
b. Write it down or say it.
c. Oral Presentation

        i. Good to give a general overview
        ii. Hard to include much detail
    d. Written presentation
        i. More binding
        ii. More detailed
    e. Ability to present designs is at *least* as important as the ability to write more code.
    f. Higher-ups don't care about your code, they care how well you communicate.
    g. Context
        i. Start by giving some context
        ii. What are the goals? What needs to be optimized (performance, cost, …)
        iii. Give the audience a chance to critique the things that really matter
    h. Audience
        i. *The* thing to consider
        ii. Know your audience. What do they need to know?
        iii. Development Team
            1. Easiest talk to give
            2. What are they doing, how does it fit into the rest of the project?
        iv. Other Technical People
            1. What are you doing and why is it interesting?
            2. What can they steal?
            3. What can they contribute to your work?
            4. Sell how great it would be to work on this project – you're recruiting.
        v. Marketing
            1. What does it do?
            2. How is it better? How is it worse? What are the risks? These are the three key questions.
            3. Need to know honest answers.
            4. Risks
                a. Can hedge a little more. Marketing thinks Engineering underestimates. Engineering thinks marketing overestimates.
                b. Don't oversell the risks or they'll oversell it.
                c. "This feature may not get included."
                d. "We may not hit this delivery date."
        vi. Technical Support
            1. Late in the cycle
            2. What does it do?
            3. How does that differ from the previous release?
            4. Where is it likely to break? (Confusing, can get wrong result if these parameters are missing)
        vii. Quality Assurance
            1. What does it do?
            2. What's been implemented
            3. Where might we break it?
            4. Very different mindset here.
        viii. Customers
            1. Toughest one.
            2. What does it do?
            3. They're looking for confidence (in large part in the developers)
            4. Want to know that you understand the details
            5. If you're going to present risks, clear it with sales / marketing.
    i. Presentation
        i. Use abstractions, but only if they help answer a question for the audience
        ii. Three ways to present abstractions
            1. Text. Describe
            2. Picture. Show.
            3. Formal Description. Define.

iii. Functional Abstractions
   1. Boxes and Arrows work well.
   2. Use distinguishable arrow types.
   3. In written documents, describe components separately too
iv. Data Flow
   1. Also important
   2. Flow chart is a common representation
   3. Don't need much accompanying text if you have good labels
v. Data Structure. Typically a textual list of "stuff in the database"
vi. Communication
   1. Who's in control and who's reacting?
   2. Message sequence chart
   3. Good way to quickly describe / understand the protocol.

j. Speaking Mechanics
   i. Prepare slides carefully
   ii. Use phrases / short sentences, not big narratives on slides
   iii. Use white space / color / font effectively. Don't make it too busy.
   iv. Use high contrast colors.
   v. Talk to the audience, not the computer or the projection.
   vi. Don't point to the computer screen. Point to the projection.
   vii. Look like you want to be there.
   viii. Longer is not better. You want people to stay and listen.

k. Writing Mechanics
   i. Don't make huge narratives
   ii. Use bulleted lists!
   iii. Use bold / italics effectively
   iv. Use white space to separate key points
   v. Re-read the document. Double-check the spell checker manually.
   vi. Passive voice makes it even more boring than it's bound to be already.