User-Centered Design
Data Entry

CS 4640
Programming Languages
for Web Applications

[The Design of Everyday Things, Don Norman, Ch 7]

Seven Principles for Making Hard Things Easy

1. Use knowledge in the world and knowledge in the head
2. Simplify task structure
3. Make things visible
4. Get the mappings right
5. Exploit the power of constraints
6. Design for error
7. When all else fails, standardize

1. Use Knowledge in the World

- New users do better if everything they need to know is in the UI
- Experienced users can be faster by having knowledge in their heads
- All users are more effective if the implementation model matches their mental model
- Avoid depending on user manuals
  - A very inconvenient part of the world

2. Simplify Task Structure

- Simplify tasks by considering the users':
  - Psychology
  - Short term memory
  - Long term memory
  - Concentration
- New technology should make tasks simpler
  - Same task with mental aids
  - Increase visibility
  - Same task with simple steps automated
  - Change the nature or structure of the task

3. Make Things Visible

- Users should quickly see
  - What they can do
  - How they can do it
  - What will happen
- The possible actions should satisfy user's goal
  - Revenue, not excise
- System state should always be obvious
- Examples
  - Collab needs to show state: semester

4. Get the Mappings Right

- Intentions (what users want) to actions (what they can do)
- Actions and effects on the software
- System state and what is visible
- Perceived system state and the users needs
- Graphics, icons and pictures are easier to understand
  - But designing graphics is hard!
  - Not a common skill among programmers

5. Exploit Power of Constraints

- Constraints stop users from entering wrong data
  - Ignore dashes in phone & credit card numbers
  - Advanced controls like the dates in travel web sites
  - Selections, as in radio boxes and dropdown lists
- Think of this as strong typing for UIs …

6. Design for Error

- Users are not perfect and will enter invalid data
- Design for invalid inputs!
  - Use constraints to avoid invalid inputs
  - Correct invalid inputs automatically
  - Make it simple and convenient for users to correct invalid inputs
  - Allow users to postpone invalid input corrections
  - Make it easy to undo

7. Standardize

- The controls are all the same—consistent
- A last resort approach because it forces knowledge to be in the head
- Notice anything funny … ?

- Standardize early or it will be too late
- Standardization only has to be learned once

Improving Data Entry

- Data Integrity
  - The state of the program depends on correct, valid input data
- Input data validation means
  - Checking before sending to software
  - Rejecting if it does not conform
- Makes users feel like suspects and treats typos like malicious behavior
- Sometimes invalid data is reasonable
  - We don't have the complete data
  - We mis-typed something
  - It doesn't matter – the rules are too restrictive

Data Immunity

- Don’t use data validation to ensure integrity
- Make the software immune from invalid data

> Most invalid data can be made valid
> by the software !

- Four types of immunization
  1. Repairing automatically
  2. Masking out invalid data
  3. Flexible rule enforcement
  4. Auditing instead of editing

1. Auto-Repair

- If you search for "thomas jfferson," google will say: "Showing results for thomas jefferson"
  - Plus a link that matches the original string
- Auto-fixing examples:
  - Convert word to numbers ("five" to "5")
  - Look for relationships ("Charlottesville, BA" to "Charlottesville, VA")
- Let the programmers be creative!

It saves money to have programmers work more and users work less

2. Mask Invalid Data

- The UI can often prevent invalid data from being entered
  - Do not allow "five" for a number – use masking to ignore all non-numeric characters
  - Fill in dashes automatically, so it doesn't matter if the users entered "123-45-6789" or "123456789"
  - Use radio buttons or dropdowns when possible

3. Flexible Rule Enforcement

- Defining good rules is hard – defining perfect rules is impossible!
- Three levels of rules:
  1. The restrictions we really want (intent)
  2. The restrictions we describe (specifications)
  3. The restrictions we implement (law)
- The three never match perfectly, and considerate people consider the intent instead of the law
- Allow some rules to be bent
- Keep a log to check later

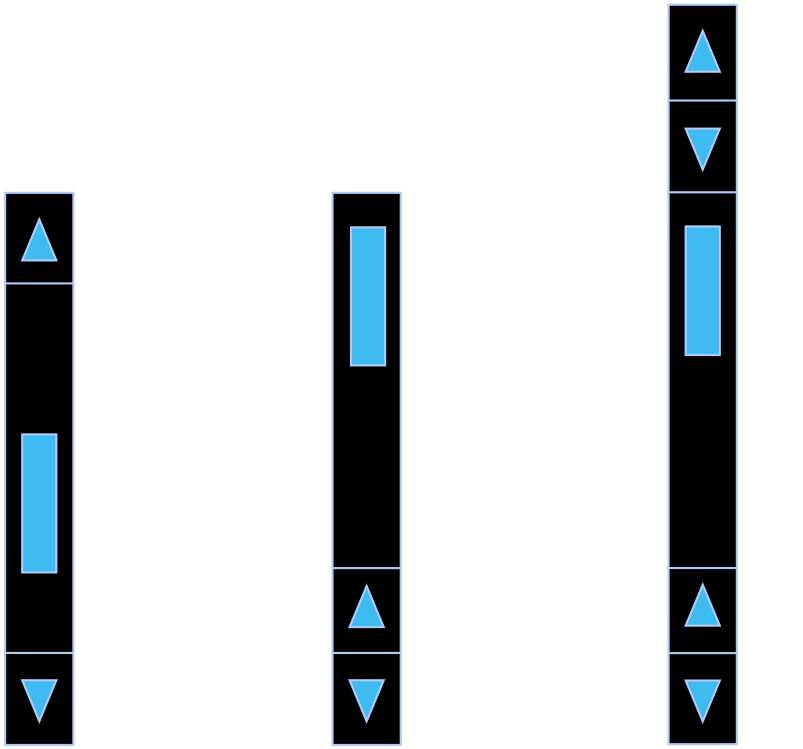
4. Audit Don't Edit

If it can't be fixed …
Do we have to bother the useres right now?

- Missing data is not a data integrity error
  - Missing data can sometimes be entered later
  - Missing data can often be inferred from existing data
    - The programmers have to work
- Mistakes can often be fixed later
  - Spelling mistakes, TurboTax's audit phase
  - Tell users about mistakes with modeless feedback
    - Modeless: feedback they do not have to respond to

Selecting Events

Keep events close together



Bad Better Good

Selection

- GUI operations have two parts:
  - Operation (verb)
  - Operands (objects)
- Command lines often use natural speaking style: verb-object
- GUIs should usually let the user select an object, then apply an operation: object-verb
  - Example: date selection
- This makes selection very important

Discrete and Contiguous Selection

- *Discrete data*: Objects are independent and need to be selected independently
  - Picture elements in a drawing tool
- *Contiguous data*: Objects are ordered in lists or matrices
  - Spreadsheet cells and words in word processors
- Whether data is discrete or contiguous sometimes depends on user needs

Summary

- Make it easy to decide what actions are possible
- Make things visible
  - Controls, choices, and the conceptual model
- Make it easy to see the state
- Use natural mappings
  - Intentions → actions
  - Actions → effect
  - Visible information → actual state
- Protect the users from mistakes
- Don’t prevent them from doing their jobs
- Users must always know what was selected before choosing an operation