User interface design

- These notes are from
  User Interface Design for Programmers by Joel Spolsky, 2001
- Controlling your environment makes you happy
  - "A user interface is designed well when it behaves exactly how the user thought it would."
- User model, program model – "Users assume the simplest model possible."
- Options and settings are normally the result of developer indecision
  - "Every time you provide the user with an option, you're asking the user to make a decision."
    - Choices that cause accidents
      - Tear-away menus, toolbars
      - Movable Windows taskbar
- Use common-sense metaphors to educate users
  - Cut – scissors
  - Zoom – magnifying glass
  - Desktop, folder metaphor
  - Tabbed layouts
  - Affordances
    - Ridged resize corner in Windows
    - door pull handles
    - hand grip on cameras
- Failed metaphors
  - The left indent, hanging indent, and first line indent on MS Word ruler
  - Multiple rows of tabs
- Consistency – the user model is partly determined from usage of other software
  "Good GUI designers use consistency intelligently, and though it may not show off their creativity as well, in the long run it makes users happier."

- The balance between allowing users to do what they want and guiding them though the process – wizards
- Design for extremes
  - People can't read
    - “Users don't have the manual, and if they did, they wouldn't read it.”
    - “In fact, users can't read anything, and if they could, they wouldn't want to.”
    - Manuals are good places for domain knowledge – like explaining accounting in the QuickBooks manual
    - There is a balance between how much text to show the user and the chance of the average user reading it
  - People can't control the mouse
    - The “mile-high” menu bar – the menu bar on Macs
    - Fonts in editing boxes – easier to edit using fixed-width fonts like Courier
  - People can't remember – the difficulty of command-line interfaces
  - It's not that people are stupid, they have better things to do
- Activity-Based Planning – what activity is the <u>user</u> trying to accomplish? The starting point of software should ask this.
- Usability testing
  - allow new people to play with your application and have them speak their thoughts
  - don't give them suggestions
  - 5-6 people is enough to find major trends
  - really more of a *learnability* study
  - downside: can be rigged easily and are often used to settle conflicts
  - can even be done with paper drawings of an interface
- Color
  “Design in black-and-white. Add color for emphasis, when your design is complete.” - Diane Wilson
- Icons – nouns make easy icons (printer, disk, etc); verbs are difficult, especially with limited space

- Internationalization
  - be aware that text takes up different sizes in different languages
    - *Note: one of the reasons Java uses layout managers*
  - different sorting, money symbols, date formats, etc
  - customs, taboo, etc – the stop sign isn't the same in all countries
  - jokes, etc – a piggybank for "save" is clever, but not intuitive

Regular Expressions

- A special syntax for describing patterns in strings
- A special topic in Theory of Computation too
- Example: a regular expression that matches the string "keith":
  `keith`
  - This also matches "keith!" and "Hey, keith, wash the dishes!"
  - To make it match only the string "keith", do this:
    `^keith$`
    - The **`^`** character signifies beginning of line (or beginning of string)
    - The **`$`** character signifies end of line (or end of string)
- Let's make one match my full name too:
  `^keith|keith trnka$`
  - The **`|`** (pipe) character means "or"
- How about any number of repetitions of my name?
  `^(keith|keith trnka)*$`
  - The parentheses are used to group things, like in math
  - The star means "repeat the expression to the left zero or more times"
  - Instead, we want it to occur one or more times:
    `^(keith|keith trnka)+$`
    - The plus means "repeat the expression to the left one or more times"
  - How about zero or one time?
    `^(keith|keith trnka)?$`
- Parentheses create something called a ***capturing group*** – the part of the string matched within the parentheses can be retrieved
  - Numbering starts left-to-right, starting at 1

    - Example
    - Non-capturing groups are created by using **`(?:...)`**
- Character classes – matches groups of characters
  `[0-9]` matches one character in the range 0 through 9
    - Union over character classes
      `[0-9[A-Z]]` matches numbers and uppercase letters
    - Alternatively, you could just do
      `[0-9A-Z]`
    - Intersection
      `[0-9A-Z&&[123]]` only matches characters 1, 2, 3
        - More useful in conjunction with negation
          `[^0-9]` matches any character except 0-9
    - Predefined character classes
        - **`.`** (dot) matches any character except line terminators (by default)
        - **`\w`** matches word characters: **`[a-zA-Z_0-9]`**
        - **`\W`** matches non-word characters: **`[^\w]`**
        - **`\s`** matches whitespace characters: **`[ \t\n\x0B\f\r]`**
        - **`\S`** matches non-whitespace characters
        - **`\d`** matches digit characters
- Special boundary matches
    - **`\b`** represents word boundaries
    - **`\B`** represents non-word boundaries
- Number of repetitions – exact or range
    - \d{3} means \d\d\d
    - \d{1,3} means \d|\d\d|\d\d\d
- Using this in Java
    - The ***String*** class has some convenient methods:

boolean	**matches**(String regex) Tells whether or not this string matches the given regular expression.
boolean	**regionMatches**(boolean ignoreCase, int toffset, String other, int ooffset, int len) Tests if two string regions are equal.

boolean	**regionMatches**(int toffset, String other, int ooffset, int len) Tests if two string regions are equal.

String	**replaceAll**(String regex, String replacement) Replaces each substring of this string that matches the given regular expression with the given replacement.
String	**replaceFirst**(String regex, String replacement) Replaces the first substring of this string that matches the given regular expression with the given replacement.
String[]	**split**(String regex) Splits this string around matches of the given regular expression.
String[]	**split**(String regex, int limit) Splits this string around matches of the given regular expression.

- If you want to do anything more complex or efficient, look at the *java.util.regex* package
  - The ***Pattern*** class represents a regular expression

static Pattern	**compile**(String regex) Compiles the given regular expression into a pattern.
static Pattern	**compile**(String regex, int flags) Compiles the given regular expression into a pattern with the given flags.
int	**flags**() Returns this pattern's match flags.
Matcher	**matcher**(CharSequence input) Creates a matcher that will match the given input against this pattern.
static boolean	**matches**(String regex, CharSequence input) Compiles the given regular expression and attempts to match the given input against it.
String	**pattern**() Returns the regular expression from which this pattern was compiled.
String[]	**split**(CharSequence input) Splits the given input sequence around matches of this pattern.
String[]	**split**(CharSequence input, int limit) Splits the given input sequence around matches of this pattern.

- The sorts of flags you can pass are **`Pattern.CASE_INSENSITIVE`**, **`Pattern.DOTALL`** (you add them to use multiple flags)
- To match things, obtain a ***Matcher*** by calling **`matcher`** on the string you'd like to match
- The ***Matcher*** class – important methods

String	**group**(int group) Returns the input subsequence captured by the given group during the previous match operation.
boolean	**matches**() Attempts to match the entire input sequence against the pattern.
String	**replaceAll**(String replacement) Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
String	**replaceFirst**(String replacement) Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.

- Note: you have to do string escaping in Java, unlike Perl ((\d{3}-|\(\d{3}\))?\d{3}-)?\d{4} matches many phone numbers "((\\d{3}-|\\(\\d{3}\\))?\\d{3}-)?\\d{4}" is the escaped version in Java