7 Cognitive Dimensions
Usability claim red flags

“highly usable”

“simple”

“intuitive”

“user friendly”

“natural”

“easy to use”

What do you mean when you use these terms? Can you justify it?

Lesson: use the same care for usability claims that you do for technical claims
What Are Cognitive Dimensions?

A **terminology** for discussing the **usability** of your language (or tool)

A **lightweight tool** to support language **design** and **evaluation**

A **framework** for thinking about **usability tradeoffs**

… from research on psychology of programming (Thomas Green)

… widely used in HCl-related fields, especially end-user programming
What Are CDs Good For?

Gives **names to usability concepts**
- effectively discuss and communicate design rationale
- raises awareness, makes them enumerable

Lightweight, **informal design aid**
- easy to learn, easy to apply
- applicable at any stage of the design
- emphasizes trade-offs between dimensions

Lightweight, **qualitative evaluation**
- motivate design decisions, trade-offs
- identify strengths and weaknesses
What Are the Limitations?

Not a metric
- can’t tell you if your language is good or not
- can’t measure the effect of a design decision

Some attempts to operationalize:
- Representation Benchmarks (Yang et al., 1997)
- Physics of Notation (Moody, 2009)
  not really useful … too low-level and easy to abuse

Not complete – often see “new” dimensions proposed

Imprecise – room for interpretation, disagreement
All the Cognitive Dimensions

- **abstraction gradient**: availability and necessity of abstraction mechanisms
- **closeness of mapping**: similarity to problem domain
- **consistency**: self-similarity – similar semantics expressed by similar syntax
- **diffuseness/terseness**: verbosity of the language
- **error-proneness**: does the notation invite or prevent certain mistakes?
- **hard mental operations**: is reading/writing overly arduous? requires paper and pencil?
- **hidden dependencies**: are important links between entities visible?
- **provisionality**: degree of commitment to programming actions
- **premature commitment**: are decisions forced too early? are actions reversible?
- **progressive evaluation**: ability to check and run parts of an incomplete solution
- **role-expressiveness**: are the roles of components obvious?
- **secondary notation**: ability to carry extra information outside of formal syntax
- **viscosity**: resistance to change
- **visibility & juxtaposability**: ability to view all components easily
Abstraction Gradient

*Availability and necessity of abstraction mechanisms*

Questions:
- can the user define their own abstractions?
- are there abstractions that *must* be learned? if so, how many?

---

<table>
<thead>
<tr>
<th>abstraction starved</th>
<th>abstraction hungry</th>
</tr>
</thead>
<tbody>
<tr>
<td>no user-level abstractions</td>
<td>provides abstraction mechanisms</td>
</tr>
</tbody>
</table>
| forces learning many abstractions | }
Closeness of Mapping

*Similarity of notation to problem domain*

Questions:

• are there existing notations in the domain that can be reused?
• do language constructs correspond to domain concepts?
• are the most important concepts built in?

*Usually a focus of DSLs!*

Interacts with abstraction gradient:

• what are the abstraction mechanisms in the domain?
• but often the point is to provide more abstraction
Consistency

Self-similarity – similar semantics expressed by similar syntax

Python lists

\[
\begin{align*}
\text{xs} &= [1, 2, 3] \\
\text{length} &= \text{len}(\text{xs}) \\
\text{indexing} &= \text{xs}[2] \\
\text{adding} &= \text{xs}.\text{append}(4) \\
\text{is empty?} &= \text{not} \ \text{xs}
\end{align*}
\]

“There should be one, and preferably only one, obvious way to do it.”
Diffuseness/Terseness

Verbosity of the language

\[ \text{life} \leftarrow \{ \uparrow 1 \omega \land 3 4 = +/\mathcal{1} 0 \mathcal{1} \cdot \theta \mathcal{1} 0 \mathcal{1} \cdot \mathcal{1} \subset \omega \} \]

Conway’s Game of Life in APL

import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSLog (@"Hello, World!");
    [pool drain];
    return 0;
}

Hello, World! in Objective C
Error Proneness

Does the notation invite or prevent certain mistakes?

\[
\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash D(e_1, e_2) : D(T_1, T_2)}
\]

Type systems

\[
\text{if } (x = y) \{ \\
\text{...} \\
\} \\
\text{C “gotcha”}
\]

data Pt = Pt Int Int

Haskell name spaces

Structured editors
Hard Mental Operations

Is reading/writing overly arduous? requires paper and pencil?

: fib ( n -- fib )
  0 1 rot 0 ?do over + swap loop drop ;

Fibonacci numbers in Forth

Shell scripts

Makefiles
Hidden Dependencies

Are important links between entities visible?

<table>
<thead>
<tr>
<th>Participation</th>
<th>Exam #1</th>
<th>Exam #2</th>
<th>Project</th>
<th>Exam #1 %</th>
<th>Exam #2 %</th>
<th>Adj. Exam %</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>17</td>
<td>15</td>
<td>10</td>
<td>0.34</td>
<td>0.375</td>
<td>0.3825</td>
<td>70.3</td>
</tr>
<tr>
<td>18</td>
<td>27</td>
<td>26</td>
<td>10</td>
<td>0.54</td>
<td>0.65</td>
<td>0.62</td>
<td>84.8</td>
</tr>
<tr>
<td>17</td>
<td>42</td>
<td>29</td>
<td>8</td>
<td>0.84</td>
<td>0.725</td>
<td>0.8075</td>
<td>84.6</td>
</tr>
<tr>
<td>18</td>
<td>49</td>
<td>37</td>
<td>10</td>
<td>0.98</td>
<td>0.925</td>
<td>0.9775</td>
<td>99.1</td>
</tr>
<tr>
<td>18</td>
<td>44</td>
<td>17</td>
<td>8</td>
<td>0.88</td>
<td>0.425</td>
<td>0.6775</td>
<td>81.1</td>
</tr>
<tr>
<td>18</td>
<td>50</td>
<td>35</td>
<td>10</td>
<td>1</td>
<td>0.875</td>
<td>0.9625</td>
<td>98.5</td>
</tr>
</tbody>
</table>

One way links in spreadsheets
Progressive Evaluation

Ability to check and run parts of an incomplete solution

Motivation for:

- Haskell `undefined`
- Agda holes
- Dynamic typing?

...ability to write an incomplete solution?
Role Expressiveness

Are the roles of components obvious?

\[
\text{life} \leftarrow \{ \uparrow 1 \circ v. \land 3 \ 4=+/, \sim 1 \ 0 \ 1 \ . \odot \sim 1 \ 0 \ 1 \ . \ominus \subset \omega \} 
\]

Conway’s Game of Life in APL

Network diagram

LEGO Mindstorms Scratch
Secondary notation

Ability to carry extra information outside of formal syntax
Viscosity

Resistance to change

Questions:

• How difficult is it to change a program? (refactoring)
• How difficult is it to add new features? (extensibility)

Lots of traditional PL and SE work addresses this dimension

Common negative example: moving elements in graphical editors
Tradeoffs Between Cognitive Dimensions

- Premature commitment
  - Increases cost
  - Can increase

- Viscosity
  - Increases
  - Can increase

- Abstraction
  - Can increase
  - Can reduce

- Hidden dependencies
  - Can increase

- Visibility
  - Can decrease

- Secondary notation
Closing Thoughts

Gives **names** to **usability concepts**

- effectively discuss and communicate design rationale
- raises awareness, makes them enumerable

Not perfect, but anecdotally useful

- accepted in HCI and end-user programming

Any method is better than no method?

- helps to externalize the design process
- supports reflection, analysis, etc.
How to Learn More

Green and Petre
*Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework*
JVLC 1996

Green and Blackwell
*Cognitive Dimensions of Information Artefacts: a tutorial*
1998

http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/