Lecture 6: Usability of programming languages

For many years, it seemed that conventional text programming would eventually be replaced by ***visual programming languages***, where program behavior is defined by drawing diagrams (many proposals resembled software engineering diagrams, such as those in UML – flow charts, object interaction, state charts etc). At a time when software development methods involved creating a complete specification in diagram form, then employing programmers to convert those into code, it seemed as though programming could be completely automated. However the fallacy of this reasoning was the same error made when FORTRAN (Formula Translation) was considered to be 'automatic programming' – any representation that defines the program behavior in sufficiently precise detail to be compiled will be more like programming than like design. Drawing highly detailed diagrams is often more laborious than writing highly detailed text, so it isn't the case that diagrams will always have superior usability relative to text.

Many elements of the modern WIMP interface originated in programming language research. There are also some good examples of programming languages that have been designed for use by special groups – ***end user programmers*** who are not professionally trained in programming, or educational programming languages that illustrate programming language principles using graphical display elements. It should be clear that different languages are good for different purposes, and for use by different people. These often include a broad mix of visual and textual (or even physical and tangible) elements, selected to meet specific needs.

Cognitive Dimensions of Notations

The usability principles by which we describe what kind of activities a language in being used for, and what kinds of visual representation can be useful or not useful for those activities, have been collected into guidance for language designers, under the name ***Cognitive Dimensions of Notations*** (CDs). Just as many innovations in programming language user interfaces have led to radically different approaches to

user interfaces, CDs are one of the most appropriate theoretical frameworks for analysis of completely new content manipulation styles.

The CDs are presented as a ***vocabulary*** for design discussion. Many of the dimensions reflect common usability factors that experienced designers might have noticed, but did not have a name for. Giving them a name allows designers to discuss these factors easily. Furthermore, CDs are based on the observation that there is no perfect user interface any more than a perfect programming language. Any user interface design reflects a set of design ***trade-offs*** that the designers have had to make. Giving designers a discussion vocabulary means that they can discuss the trade-offs that result from their design decisions. The nature of the trade-offs is reflected in the structure of the dimensions. It is not possible to create a design that has perfect characteristics in every dimension – making improvements along one dimension often results in degradation along another.

An example dimension is called ***viscosity***, meaning resistance to change. In some notations, small conceptual changes can be very expensive to make. Imagine changing a variable from int to long in a large Java program. The programmer has to find every function to which that variable is passed, check the parameter declarations, check any temporary local variables where it is stored, check any calculations using the value, and so on. The idea of what the programmer needs to do is simple, but achieving it is hard. This is viscosity. There are programming languages that do not suffer from this problem, but they have other problems instead – trade-offs. This means that language designers must be able to recognize and discuss such problems when planning a new language. The word “viscosity” helps that discussion to happen.

CDs are relevant to a wide range of content manipulation systems – audio and video editors, social networking tools, calendar and project management systems, and many others. These systems all provide a ***notation*** of some kind, and an ***environment*** for viewing and manipulating the notation. Usability is a function of both the notation and the environment.

Representative cognitive dimensions

The following list gives brief definitions of the main dimensions, and examples of the questions that can be considered in order to determine the effects that these dimensions will have on different user activities.

Premature commitment: constraints on the order of doing things.

When you are working with the notation, can you go about the job in any order you like, or does the system force you to think ahead and make certain decisions first? If so, what decisions do you need to make in advance? What sort of problems can this cause in your work?

Hidden dependencies: important links between entities are not visible.

If the structure of the product means some parts are closely related to other parts, and changes to one may affect the other, are those dependencies visible? What kind of dependencies is hidden? In what ways can it get worse when you are creating a particularly large description? Do these dependencies stay the same, or are there some actions that cause them to get frozen? If so, what are they?

Secondary notation: extra information in means other than formal syntax. Is it possible to make notes to yourself, or express information that is not really recognised as part of the notation? If it was printed on a piece of paper that you could annotate or scribble on, what would you write or draw? Do you ever add extra marks (or colors or format choices) to clarify emphasis or repeat what is there already? If so, this may constitute a helper device with its own notation.

Viscosity: resistance to change. When you need to make changes to previous work, how easy is it to make the change? Why? Are there particular changes that are especially difficult to make? Which ones?

Visibility: ability to view components easily. How easy is it to see or find the various parts of the notation while it is being created or changed? Why? What kind of things is difficult to see or find? If you need to compare or combine different parts, can you see them at the same time? If not, why not?

Closeness of mapping: closeness of representation to domain. How closely related is the notation to the result that you are describing? Why? (Note that if this is a sub-

device, the result may be part of another notation, not the end product). Which parts seem to be a particularly strange way of doing or describing something?

Consistency: similar semantics are expressed in similar syntactic forms. Where there are different parts of the notation that mean similar things, are the similarity clear from the way they appear? Are there places where some things ought to be similar, but the notation makes them different? What are they?

Diffuseness: verbosity of language. Does the notation

a) let you say what you want reasonably briefly, or b) is it long-winded? Why? What sorts of things take more space to describe?

Error-proneness: the notation invites mistakes. Do some kinds of mistake seem particularly common or easy to make? Which ones? Do you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?

Hard mental operations: high demand on cognitive resources. What kind of things requires the most mental effort with this notation? Do some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What are they?

Progressive evaluation: work-to-date can be checked at any time. How easy is it to stop in the middle of creating some notation, and check your work so far? Can you do this any time you like? If not, why not? Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not? Can you try out partially-completed versions of the product? If not, why not?

Provisionality: degree of commitment to actions or marks. Is it possible to sketch things out when you are playing around with ideas, or when you aren't sure which way to proceed? What features of the notation help you to do this? What sort of things can you do when you don't want to be too precise about the exact result you are trying to get?

Role-expressiveness: the purpose of a component is readily inferred. When reading the notation, is it easy to tell what each part is for? Why? Are there some parts that are particularly difficult to interpret? Which ones? Are there parts that you really don't

know what they mean, but you put them in just because it's always been that way? What are they?

Abstraction: types and availability of abstraction mechanisms. Does the system give you any way of defining new facilities or terms within the notation, so that you can extend it to describe new things or to express your ideas more clearly or succinctly? What are they? Does the system insist that you start by defining new terms before you can do anything else? What sort of things? These facilities are provided by an abstraction manager - a redefinition device. It will have its own notation and set of dimensions.

Notational activities

When users interact with content, there are a limited number of activities that they can engage in, when considered with respect to the way the structure of the content might change. A CDs evaluation must consider which classes of activity will be the primary type of interaction for all representative system users. If the needs of different users have different relative priorities, those activities can be emphasised when design trade-offs are selected. The basic list of activities includes:

Search

Finding information by navigating through the content structure, using the facilities provided by the environment (e.g. finding a specific value in a spreadsheet). The notation is not changing at all, though the parts of it that the users sees will vary. Visibility and hidden dependencies can be important factors in search.

Incrementation

Adding further content without altering the structure in any way (e.g. adding a new formula to a spreadsheet). If the structure will not change, then viscosity is not going to be very important.

Modification

Changing an existing structure, possibly without adding new content (e.g. changing a spreadsheet for use with a different problem).

Transcription

Copying content from one structure or notation to another notation (e.g. reading an equation out of a textbook, and converting it into a spreadsheet formula).

Exploratory design

Combining incrementation and modification, with the further characteristic that the desired end state is not known in advance. Viscosity can make this kind of activity far more difficult. This is why good languages for hacking may not be strictly typed, or make greater use of type inference, as maintaining type declarations causes greater viscosity. Loosely typed languages are more likely to suffer from hidden dependencies (a trade-off with viscosity), but this is not such a problem for exploratory design, where the programmer can often hold this information in his head during the relatively short development timescale.