

## **Design for Design for Analysis**

*Michal Young*

*Dept. of Computer and Information Science*

*University of Oregon*

*michal@cs.uoregon.edu*

Design for analysis (of which design for test is one part) means creating a design in a way that facilitates detection of errors either in the design itself or in a related artifact, usually by some automated technique. Design for design for analysis means designing notations, design approaches, and analysis techniques to support design for analysis. Design for design for analysis (DfDfA) is clearly an important aspect of the evolution of design, and there are many examples of improvements in notations, approaches, and techniques whose value lies in DfDfA, but as yet we have little systematized knowledge about DfDfA.

### **Some things we know**

We can observe several characteristics of DfDfA from the evolution of programming languages, and to a lesser extent from experience with other notations for expressing design. There is a recurring pattern of evolution that begins with recognizing an error idiom (e.g., use of an uninitialized variable in Fortran), and progress through development of techniques to detect the error (data flow analysis a la DAVE), modification of notation to make the error idiom less likely (e.g., requiring explicit variable declarations), and sometimes introduction of new notation rules that rule out the error as a matter of syntactic well-formedness. Occasionally the opportunity to commit an error is removed by taking away the chance to make a particular design decision, i.e., analysis is replaced by synthesis.

We know some of the design desiderata for DfDfA. It is important that errors be not only detectable, but also explainable. The simple initialize-before-use rule in Java, which requires initialization even on syntactic control flow paths that can never be executed, is tolerable because it is perfectly obvious to the programmer how to repair the program and avoid the error message regarding an unexecutable path. We know that properties of interest are sometimes decomposed so that one part can be subjected to an automated check, leaving another part to other detection methods. For example, database systems provide assurance that a set of concurrently executing transactions have the same effect as if they were executed in some order, but reasoning about the correctness of each transaction in isolation is left to the programmer.

We also know that progress in DfDfA, like advances that make software construction faster or complex systems easier to understand, is hard to measure. This is because software designers work at the edge of their competence. If we miraculously remove half the errors in systems of a particular size or complexity, then

designers are capable of producing larger and more complex systems until they again reach the edge of their competence. Then, software “productivity” by some measures does not improve, even as we are able to construct far more elaborate and useful software systems.

### **Some things we don’t know**

We don’t know why the whiteboard, which provides no syntactic constraints or other validation capabilities, is still the best medium for thinking about and creating and even communicating design. Would it be better (or even possible) to introduce constraints incrementally, on demand, rather than having one fixed set of syntactic and semantic rules for a given notation?

We don’t know nearly enough about the relation between analysis and explanation. Work on literate programming and other approaches to presentation of design (which includes source code) has had only a loose connection with research in analysis. There has been some work in analysis to support code inspection, but less on how analysis is or could be tied to intentional explanation.

We don’t know when it is best to combine different kinds of information in a single notation, and when it is better to have completely different ways of saying things. We know, roughly speaking, that inconsistencies are both less likely to occur and easier to explain if they involve two elements that are close to each other (e.g., on the same page), which seems to favor combining many kinds of information in a single notation. On the other hand, there are limits and costs to integrating different kinds of information, and in practice designers use a panoply of notations that suit different purposes.

We don’t understand as much as we should about the inter-dependent evolution of analysis, design notations, and design methods. Some history or archeology is in order, particularly to determine the extent to which the relatively well-known history of general purpose programming languages is representative of evolution of other software design notations, including domain-specific languages and non-executable notations, and design notations and approaches in fields outside of software.

Work on these and related questions should be part of an overall effort to systematize our knowledge about design.

### **Author Background**

Michal Young works in the area of software engineering, primarily in software test and analysis but also in software architecture and domain-specific languages.