1: Introduction

Software is built on abstractions. Pick the right ones, and programming will flow naturally from design; modules will have small and simple interfaces; and new functionality will more likely fit in without extensive reorganization. Pick the wrong ones, and programming will be a series of nasty surprises: interfaces will become baroque and clumsy as they are forced to accommodate unanticipated interactions, and even the simplest of changes will be hard to make. No amount of refactoring, bar starting again from scratch, can rescue a system built on flawed concepts.

Abstractions matter to users too. Novice users want programs whose abstractions are simple and easy to understand; experts want abstractions that are robust and general enough to be combined in new ways. When good abstractions are missing from the design, or erode as the system evolves, the resulting program grows barnacles of complexity. The user is then forced to master a mass of spurious details, to develop workarounds, and to accept frequent, inexplicable failures.

The core of software development, therefore, is the design of abstractions. An abstraction is not a module, or an interface, class, or method; it is a structure, pure and simple—an idea reduced to its essential form. Since the same idea can be reduced to different forms, abstractions are always, in a sense, inventions, even if the ideas they reduce existed before in the world outside the software. The best abstractions, however, capture their underlying ideas so naturally and convincingly that they seem more like discoveries.

The process of software development should be straightforward. First, you design the abstractions, from a careful consideration of the problem to be solved and its likely future variants. Then you develop its embodiments in code: the interfaces and modules, data structures and algorithms (or in object-oriented parlance, the class hierarchy, datatype representations, and methods).

Unfortunately, this approach rarely works. The problem, as Bertrand Meyer once called it, is *wishful thinking.* You come up with a collection of abstractions that seem to be simple and robust. But when you implement them, they turn out to be incoherent and perhaps even inconsistent, and they crumble in complexity as you attempt to adapt them as the code grows.

Why are the flaws that escaped you at design time so blindingly obvious (and painful) at coding time? It is surely not because the abstractions you chose were perfect in every respect except for their realizability in code. Rather, it was because the environment of programming is so much more exacting than the environment of sketching design abstractions. The compiler admits no vagueness whatsoever, and gross errors are instantly revealed by executing a few tests.

Recognizing the advantage of early application of tools, and the risk of wishful thinking, the approach known as "extreme programming" [4] eliminates design as a separate phase altogether. The design of the software evolves with the code, kept in check by the rigors of type checking and unit tests.

But code is a poor medium for exploring abstractions. The demands of executability add a web of complexity, so that even a simple abstraction becomes mired in a bog of irrelevant details. As a notation for expressing abstractions, code is clumsy and verbose. To explore a simple global change, the designer may need to make extensive edits, often across several files. And pity the reviewer who has to critique design abstractions by poring over a code listing.

An alternative approach is to attack the design of abstractions head-on, with a notation chosen for ease of expression and exploration. By making the notation precise and unambiguous, the risk of wishful thinking is reduced. This approach, known as *formal specification*, has had a number of major successes. Praxis, a British company that develops critical systems using a combination of formal specification and static analysis, offers a warranty on its products, boasts a defect rate an order of magnitude lower than the industry average, and achieves this level of quality at a comparable cost.

Why isn't formal specification used more widely then? I believe that two obstacles have limited its appeal. The notations have had a mathematical syntax that makes them intimidating to software designers, even though, at heart, they are simpler than most programming languages. A second and more fundamental obstacle is a lack of tool support beyond type checking and pretty printing. Theorem provers have advanced dramatically in the last 20 years, but still demand more investment of effort than is feasible for most software projects, and force an attention to mathematical details that don't reflect fundamental properties of the abstractions being explored.

This book presents a new approach. It takes from formal specification the idea of a precise and expressive notation based on a tiny core of simple and robust concepts, but it replaces conventional analysis based on theorem proving with a fully automatic analysis that gives immediate feedback. Unlike theorem proving, this analysis is not "complete": it examines only a finite space of cases. But because of recent advances in constraint-solving technology, the space of cases examined is usually huge—billions of cases or more—and it therefore offers a degree of coverage unattainable in testing.

Moreover, unlike testing, this analysis requires no test cases. The user instead provides a property to be checked, which can usually be expressed as succinctly as a single test case. A kind of exploration therefore becomes possible that combines the incrementality and immediacy of extreme programming with the depth and clarity of formal specification.

This volume introduces the key elements of the approach: a logic, a language, and an analysis:

- The *logic* provides the building blocks of the language. All structures are represented as relations, and structural properties are expressed with a few simple but powerful operators. States and executions are both described using *constraints* ("formulas" to the logician, and "boolean expressions" to the programmer), allowing an incremental approach in which behavior can be refined by adding new constraints.
- The *language* adds a small amount of syntax to the logic for structuring descriptions. To support classification, and incremental refinement, it has a flexible type system that has subtypes and unions, but requires no downcasts. A simple module system allows generic declarations and constraints to be reused in different contexts.
- The *analysis* is a form of constraint solving. *Simulation* involves finding instances of states or executions that satisfy a given property. *Checking* involves finding a counterexample—an instance that violates a given property. The search for instances is conducted in a space whose dimensions are specified by the user in a "scope," which assigns a bound to the number of objects of each type. Even a small scope defines a huge space, and thus often suffices to find subtle bugs.

This book is aimed at software designers, whether they call themselves requirements analysts, specifiers, designers, architects, or programmers. It should be suitable for advanced undergraduates, and for graduate students in professional and research masters programs. No prior knowledge of specification or modeling is assumed beyond a high-school–level familiarity with the basic notions of set theory. Nevertheless, it is likely to appeal more to readers with some experience in software development, and some background in modeling.

Throughout the book, I use the term “model” for a description of a software abstraction. It’s not ideal, because a software abstraction need not be a “model” of anything. But it’s shorter than “description,” and has come to have a well established (and vague!) usage.

To keep the text short and to the point, I’ve relegated discussions of trickier points and asides to question-and-answer sections that are interspersed throughout the text. For the benefit of researchers, I’ve used these sections also to explain some of the rationale behind the Alloy language and modeling approach.

In the book’s appendices you’ll find a series of exercises designed to help develop modeling and analysis skills; a reference manual for the Alloy language; a summary of the semantics of the logic; and a comparison of Alloy to some well-known alternatives.

There’s no better way to learn modeling than to do it. As you read the book, I recommend that you try out the examples for yourself, and experiment to see the effects of changes.

The Alloy Analyzer is freely available at *http://alloy.mit.edu* for a variety of platforms. It can display its results in textual and graphical form, and includes a visualization facility that lets you customize the graphical output for the model at hand.

All the examples in the book are available for download at the book’s website, *http://softwareabstractions.org*, along with other supplementary material.