Building Blocks of Software Design

Ipek Ozkaya

SOFTWARE DESIGN INVOLVES the process of understanding the requirements and creating the artifacts that specify these requirements as the product to be built. The specification of the requirements ultimately happens in code. Intermediate abstraction mechanisms, such as domain modeling languages, software design and architecture patterns, programming paradigms, and design fragments, assist software engineers to specify requirements further into the final designs as implementations. However, in the absence of commonly agreed-upon building blocks that assist software engineers in tracing the design specification across software elements, these abstraction mechanisms become sources of unintended errors. Consequently, despite the availability of many software development lifecycle processes and implementation tool support, designs erode and drift from their intent quicker than anticipated.

Software design refers to both the process of creating the software product as well as the characteristics of the product itself. Design thinking and similar approaches assist software engineers during the design process through prototyping, testing, and experimentation of concepts. Techniques such as visualization or use of metaphors provide software engineers tools to further progress their designs. However, ultimately the vocabulary—the building blocks—that software design gets expressed and delivered to its end users is through code.

The ease of creating and building code, compared to creating and building any other engineering product that requires manufacturing steps, lies at the heart of the software complexity, design drift, rework, and catastrophic failure challenges of the software industry. The software design process gets cut short, code is not utilized effectively to design but to quickly implement the changes, creating a gap between the optimal design and the actual deployed design. In an effort to bridge this gap, we need better building blocks that assist software engineers to design and communicate the design as well as construct the design.

Code as Software Design?

In his 1992 essay, “What Is Software Design?” Reeves suggests that one of the reasons C++ as a programming language had become popular was because C++ made it easier to design software and program it at the same
Reeves also argues that the only documentation that satisfies the engineering criterion of the ability to build software to a specification is the source code as an engineering document itself. The act of constructing (manufacturing) in software is taken care of by compilers.

Reeves observes that one reason why software gets complex very quickly is a consequence of the fact that building the design in software is a simple push-button compile action, with much less overhead of manufacturing. Today, building and deploying the end product is further simplified by the increasing capabilities of DevOps automation pipelines. Ease of making and deploying changes results in a focus on delivering the changes, as opposed to making the changes with sound design. Because creating the source code as the document and its construction by the compiler is perceived to be cheaper activities, subsequent changes, iterations, and evolutions are often done reactively, many times resulting in adding unintended complexity to software.

Building blocks that are expressive enough to represent software design as well as its construction should provide ability to specify information about the components, such as how elements communicate, what states elements are in, and what states persist as well as strong type checking that allows for detecting errors. If the ultimate representation of the design becomes the code, next-generation programming languages should make a targeted effort in their expressive power for representing such design characteristics.

### Models as Software Design?

A key criticism against “the design is in the code” perspective is often the inability of the code to express runtime properties, the static and dynamic communications, and cross-cutting concerns across different elements of the software. Architecture thinking assists in capturing these kinds of specifications and system properties, and architecture modeling languages provide a vocabulary to express them. However, the high-level architecting process can be several steps removed from the act of programming. Model-based software engineering approaches use formal modeling languages that generate code to fill this gap; however, these techniques are still yet to be robust enough to represent the software product comprehensively. General-purpose modeling languages, such as UML, have failed in providing the necessary level of formalism in an effort to provide general enough design representation. Other modeling languages that provide tighter formal specifications and code generation capabilities, such as Architecture Design and Analysis Language (AADL), have limited scope.

Ironically, the needs expressed by software developers and architects from software architecture and modeling languages are no different than their needs from programming languages that they should be expressive enough to represent design. A study conducted with 48 practitioners from 40 IT companies revealed that architecture languages are not closing the software design expressiveness gap any better. The top gap software engineers found as a barrier in industry for using existing architecture modeling languages was their limited formalism to support design analysis. The most commonly cited limitations included lack of ability to express quality attribute properties, such as latency, throughput, propagation of change,
and lack of formality resulting in languages with no precise semantics.

Closing the Gap
Designing and delivering a software product, like any other engineering product, is a complex process that involves many activities, artifacts, and stakeholder communication techniques to collect, implement, and verify the requirements. Agile software development processes and the availability of better deployment tools have resulted in the reduction of errors that stem from communication barriers and delays introduced due to high-ceremony processes. The ultimate challenge in improving software products remains to be figuring out how to avoid the design errors and inconsistencies introduced among the many layers of abstraction that are essential in managing the complexity of the process.

Programming is not solely about constructing software—programming is about designing software. Thinking about the source code as the design does not imply don’t design, just code. Good architecture and abstractions are essential. Similarly, architecting is not solely about designing software, architecting is about constructing software. Software engineers increasingly require high-level programming languages that are closer to how software engineers think and design software as well as modeling languages that are closer to how detailed designs can be realized in code. While we continue to groom architects that think in code and developers that think in design, there are also opportunities for developing better programming languages that can express design and better tools that provide automated support for iterative design and design conformance.

References