Raising the Level of Abstraction: Deterministic Code Generation for C++

by Mark Wilson

Raising the Level of Abstraction: Deterministic Code Generator for C++
SAMPLE CHAPTER

Published by **Autumnal Software Press**
Rochester, New York, USA
ISBN:979-8-9941850-3-2
Author contact:
mark@autumnalsoftware.com

This is an early-release edition. The core ideas are complete; additional chapters and refinements will be added over time.

The Right Level of Abstraction

In the early 2000s, the software industry placed a lot of hope on UML.

The idea seemed straightforward. Instead of writing code directly, engineers would design systems using diagrams. Tools would generate code from those diagrams, and development would proceed through something called *round-trip engineering*, where diagrams and source code remained synchronized.

It didn't work out that way; these tools created duplicate sources of truth, made maintenance more difficult, and slowed development.

UML diagrams still exist, but they are rarely the primary artifact in a system. The code is the source of truth, not the diagrams. Most teams treat diagrams as documentation, not as something from which the system is built.

Understanding why is useful, because the problem was not tooling, it was bad abstraction.

The Abstraction Test

A good abstraction removes mechanical decisions you shouldn't have to make, and preserves system-level control.

Assembly language removed the need to think in machine code. High-level languages removed the need to manage registers and memory directly. Each step moved us further away from implementation details and closer to describing intent.

UML was supposed to do the same thing, but in reality, it often did not.

A typical UML class diagram describes things like:

- classes
- attributes
- methods
- inheritance
- visibility

These are programming language concepts. They are not system concepts.

If a model already specifies classes, members, methods, and relationships, then most of the work of programming has already been done. At that point, the model is not operating at a higher level of abstraction. It is simply describing the programming language in a different notation.

You may as well write the code yourself.

The Round-Trip Problem

UML based development often relied on round-trip engineering:

UML Model -> Generate Code -> Modify Code -> Synchronize Model

In theory, diagrams and code would remain synchronized, but it was actually a fragile methodology.

Developers modify code as they work, and the model quickly diverges from the implementation. Keeping both representations synchronized requires discipline and tooling that most teams cannot sustain.

The result is that the diagrams fall out of date and stop being trusted.

What Actually Works

There is a form of “round-trip” that does work in practice; it runs in the opposite direction.

Tools such as Doxygen generate diagrams from source code. The code remains the source of truth, and the diagrams are derived artifacts.

Source Code -> Documentation Tool -> Generated Diagrams

This works because it avoids maintaining two competing representations of the same system; the code is the only source of truth.

Modeling the Wrong Thing

UML models software structure. It describes how a program is organized in terms of classes and relationships, but most systems are not fundamentally about classes and inheritance.

They are about:

- data flowing through a system
- components transforming that data
- connections between those components

A model that focuses on programming constructs misses this.

A Different Kind of Model

Consider a simple system that processes sensor data. A useful description might look like this:

```
measurements:
  - Temperature
  - Humidity
queues:
  - TemperatureQueue
stages:
  - TemperatureSensor
  - Logger
```

This describes the structure of the system: what data exists, how it moves, and which components process it

The description does not mention:

- constructors
- access specifiers
- inheritance
- templates

Those details belong in the implementation.

This model operates at a higher level of abstraction. It describes the system directly.

Why This is Important

The level of abstraction determines what can be automated. If a model describes programming language constructs, there is little left to generate. The model has already done most of the work.

If a model describes system concepts, the generator can produce large amounts of useful infrastructure:

- data types
- message structures
- pipeline wiring

- supporting code

This is the key difference.

The Lesson

UML failed as a development abstraction. It attempted to model software using the same concepts as the programming language. As a result, it did not meaningfully raise the level of abstraction.

A useful modeling approach must describe the system in terms of its own concepts, not the constructs of the language used to implement it.

Looking Ahead

In the next chapter, we will start building such a model. Instead of describing classes and methods, we will describe measurements, queues, and processing stages.

From that model, we will generate code.