Elements of an extensible compiler

Ramaswamy Krishnan-Chittur

Table of contents:

1. Background:

The role of specs in a software project:

Big software projects are driven by Specs. Specs give directions for software development, and remove any ambiguities associated with if the software manages to meet its requirements or not. B-Level specifications are more than simple guidelines about the requirements and data flow of the program – Each **'Shall'** in the B-Spec is a legal binding as well. The customer can drag the software developer to the court if the software fails to meet its B-level specification. At the qualification tests the customer and the project manager verify whether each **'Shall'** is met. Suffice to say that well-defined specifications are mandatory for large software projects.

A specs-aware compiler?

Specs written down on paper, however well defined they may be, are subjective to a certain degree. The user and the project manager may interpret a spec in slightly different ways to get entirely different ideas of the requirement. Wouldn't it be better if we could develop a system where the code developed could be **'compiled and built'** only if it meets the specifications? This would significantly reduce the tests on if the software meets the requirements at various stages of the software development like integration and qualification.

This paper discusses a concept called ‘extensible compiler’ that is a means to achieving the above-said goal.

2. Theory:

Goal:

The **‘extensible compiler’** concept that is proposed in this paper is a means of developing a system where custom checks can be added during compilation time. If the code under compilation does not meet the custom check requirements, the compilation would not be successful, and as a result the compiler won’t generate a compiled binary output file for the code.

Elements of an extensible compiler:

Extensible compiler is a generic concept. However, we expect that a language that might want to support it might require the following features:

1. An interface to its compiler (to be extended to a custom compiler).
2. Capability for developing custom specifications as code (code-specs henceforth).
3. Facility to apply the specifications on the code under development.
4. Provision for the custom compiler to check the code-specs.

.NET for our work – why?

For the purpose of illustration, we have developed a prototype application called the 'extensible compiler' with the above-said goals. We have used C#/.NET for our work. There are three important reasons for this choice –

1. .NET has certain FCL classes that provide an interface to the C# compiler.
2. Code-specs can be developed as attribute classes, and employed over the code under development as declarative attributes.
3. The user can extend the system-provided C# compiler to check the code-specs, implemented as attributes, using reflection.

In the following sections, we discuss the architecture and design of an extensible compiler, with the prototype demo that we built. This document, however, discusses the structure for an application more extensive than the prototype.

3. Context Diagram:

The custom C# compiler is fed with the name of a C# project file (.csproj); [The compiler internally parses the C# project file to find out the source codes associated with the project, and absorbs them as input]. The ‘custom compile’ processing is done processing block. The output is either a compiled binary or a set of error messages.

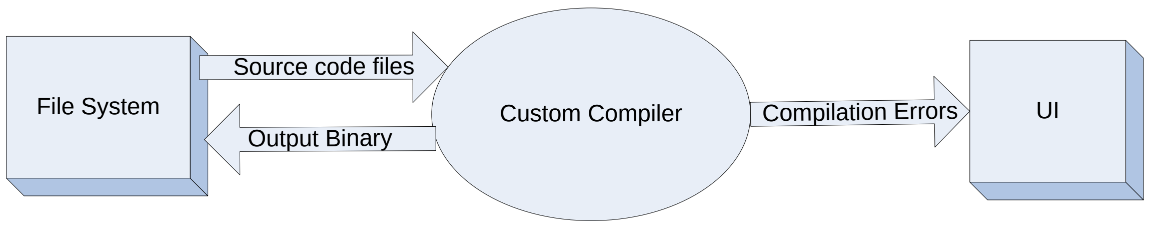

Figure 1: Context diagram

4. Module Layout:

The following diagram shows the module layout for the “custom C# compiler” project. The functioning of the different modules is described below:

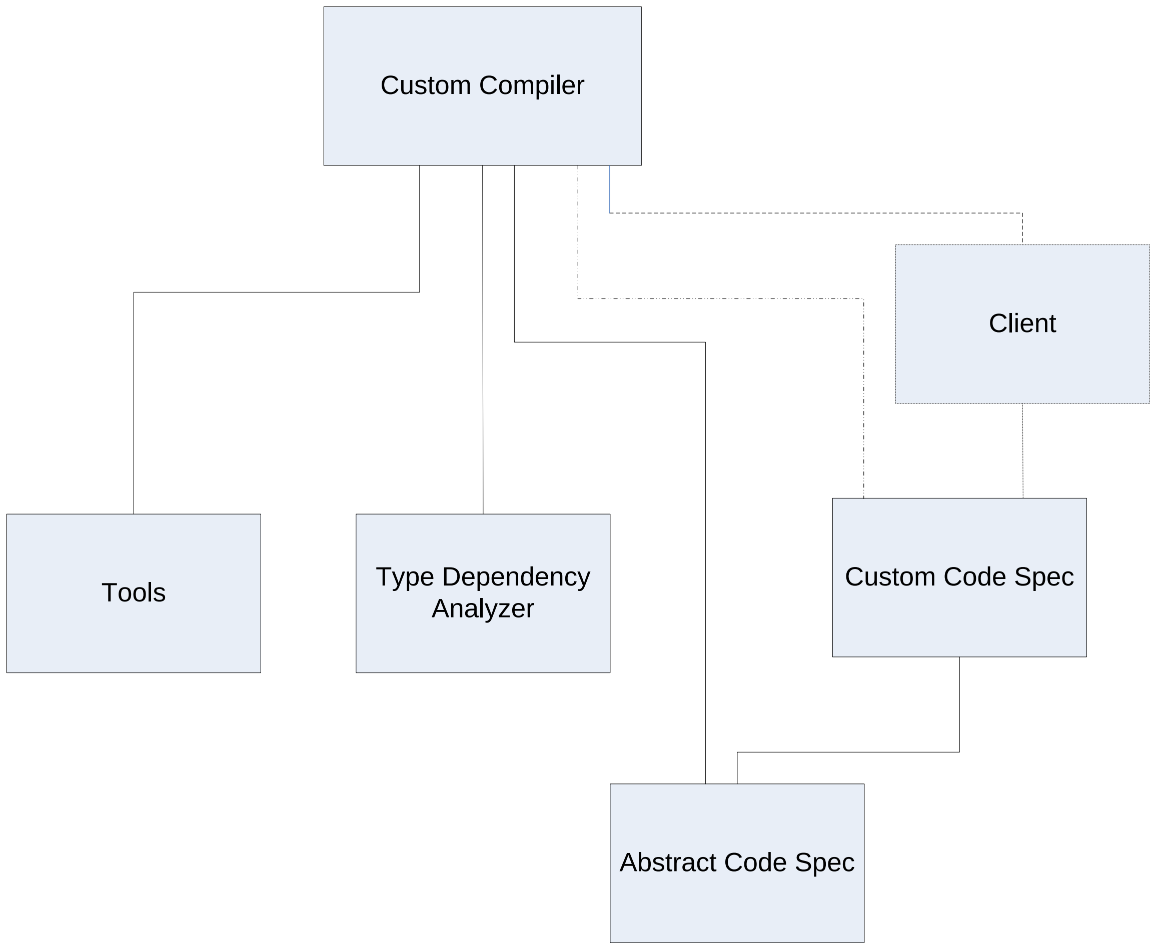

Figure 2: Module layout

Tools:

The ‘Tools’ module provides some of the commonly used utility functions – like ***test harness*** and ***code coverage***. While test suites and profiling tools are not quintessential for the system, they provide valuable service for developing software systems.

Type Dependency Analyzer:

This module is **not** vital to the **custom compiler** system; but it is a useful add-on. Apart from supporting custom checks that can be used as a concrete implementation of an organization’s process, the extensible compiler may employ a ***Type Dependency Analyzer*** as a handy tool for project managers. So a project manager can view, along with warnings and error messages, PMAs (Project Manager Alerts). The PMAs are generated by analyzing the type-dependencies of the various types defined in the assembly being compiled. These PMAs would give some pointers to the project manager regarding any possible pit falls that may be encountered.

Abstract Code Spec:

This module defines ‘abstract code specs’ – abstract attribute classes that may be applied over assemblies, types, methods etc. The **custom compiler** checks for instances inheriting from these abstract attribute types, and polymorphically calls the custom checks specified by them during custom compilation.

Custom Code Spec:

This module extends the abstract attribute code spec. The custom ***code-specs*** are implemented as concrete attribute classes, and hence can be applied over the client code. The custom compiler, aware of the abstract code specs, polymorphically calls the custom code specs to perform custom compilation checks.

Custom Compiler:

Custom compiler extends the basic C# compiler provided by .NET. The custom compiler module first compiles the source code files into memory – this is referred to as basic compilation; in case the basic compilation fails, errors are reported, and the compilation stops. If the basic compilation succeeds, the custom compiler performs custom compilation – by performing custom checks on the source code. If the custom compilation succeeds, the output binary file is generated. Otherwise, the custom compilation errors are reported. The compilation stops here.

5. Class diagram:

Each class in the “custom compiler” project may be developed to correspond to the respective module in the module layout – the class diagram is shown in diagram 3.

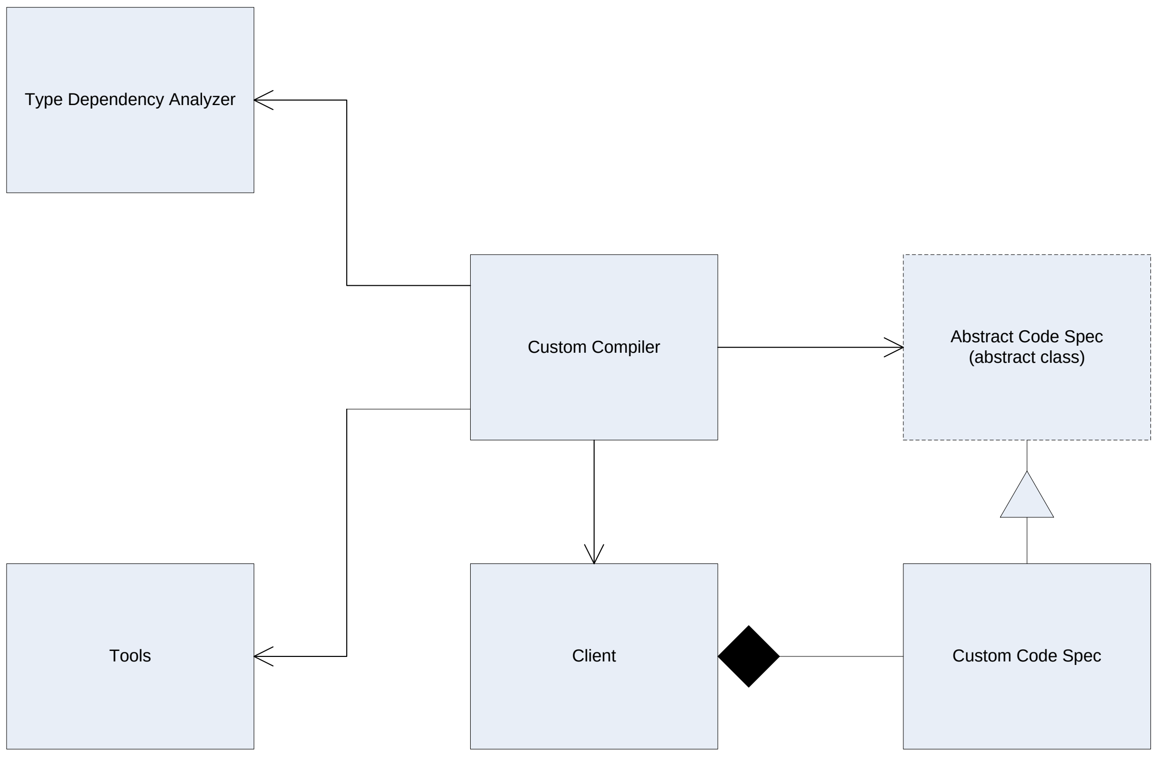

Figure 3: Class Diagram

6. Activity Diagram:

The sequence of activities for the custom compiler is simple. The source code files are fed to the compiler for basic compilation. If the compilation fails, errors are displayed and the compilation stops – else the assembly is compiled in memory. The assembly is then compiled using custom compilation checks. If the compilation succeeds, output binary is generated. Otherwise errors are displayed, and the compilation stops.

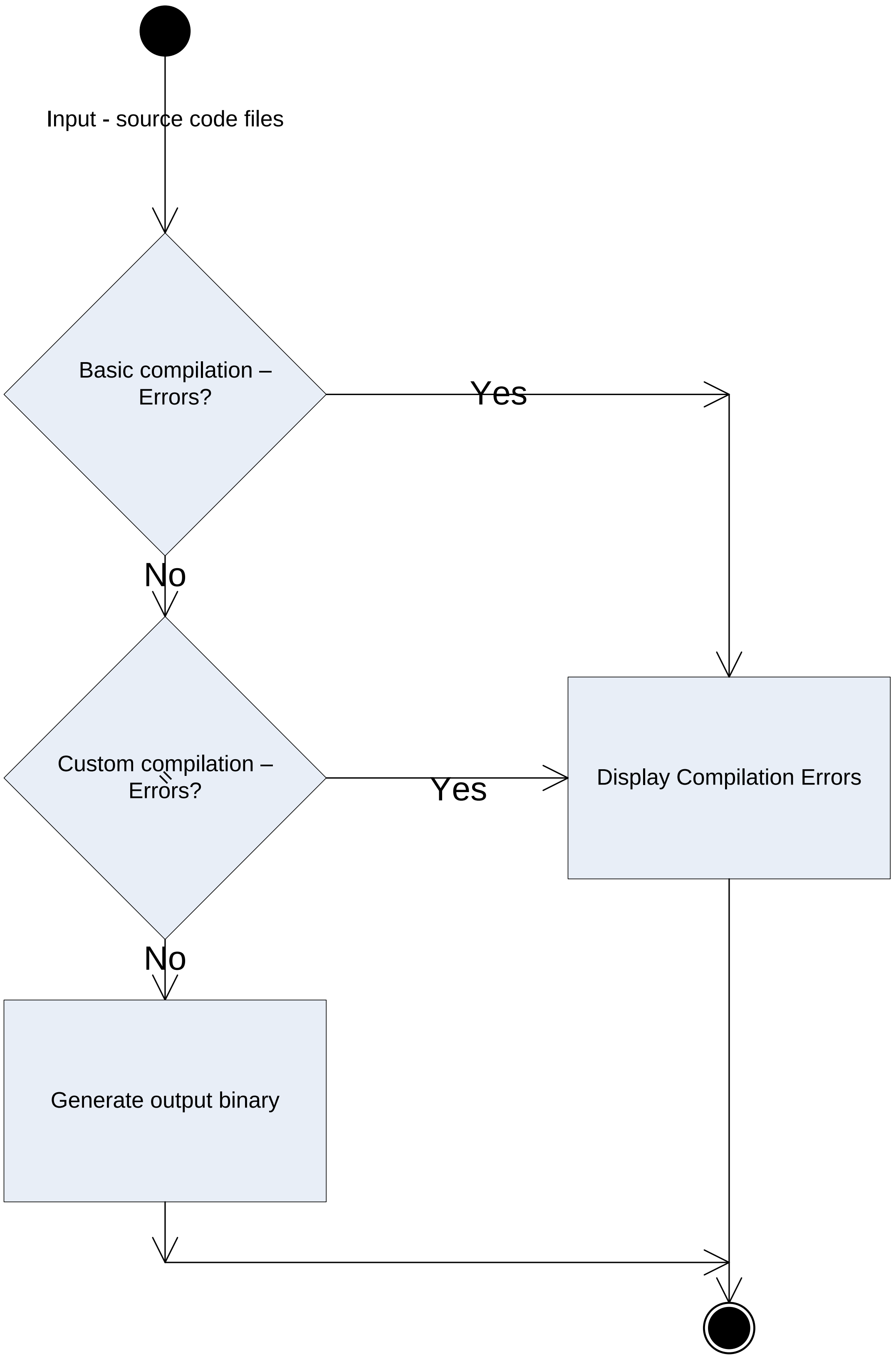
Input - source code files
Basic compilation – Errors?
Yes
No
Custom compilation – Errors?
Yes
Display Compilation Errors
No
Generate output binary

Figure 4: Activity Diagram

7. Summary:

The primary idea of the extensible compiler is to extend the basic compiler of a language with custom checks so that the requirements of a project developed under the language are ***'qualification tested'* at compilation time**. The extensible compiler may be supplied with supporting tools like *type dependency analyzers*, which provide the program manager with a clue to the hierarchical structure of the software and complexity of the design. A prototype 'extensible compiler' has been developed using C#/.NET and tested on sample projects with interesting results.