The Architecture of
Open Source Applications
Elegance, Evolution, and a Few Fearless Hacks

CMake

Bill Hoffman and Kenneth Martin

In 1999 the National Library of Medicine engaged a small company called Kitware to develop a better way to configure, build, and deploy complex software across many different platforms. This work was part of the Insight Segmentation and Registration Toolkit, or ITK[1]. Kitware, the engineering lead on the project, was tasked with developing a build system that the ITK researchers and developers could use. The system had to be easy to use, and allow for the most productive use of the researchers' programming time. Out of this directive emerged CMake as a replacement for the aging autoconf/libtool approach to building software. It was designed to address the weaknesses of existing tools while maintaining their strengths.

In addition to a build system, over the years CMake has evolved into a family of development tools: CMake, CTest, CPack, and CDash. CMake is the build tool responsible for building software. CTest is a test driver tool, used to run regression tests. CPack is a packaging tool used to create platform-specific installers for software built with CMake. CDash is a web application for displaying testing results and performing continuous integration testing.

5.1. CMake History and Requirements

When CMake was being developed, the normal practice for a project was to have a configure script and Makefiles for Unix platforms, and Visual Studio project files for Windows. This duality of build systems made cross-platform development very tedious for many projects: the simple act of adding a new source file to a project was painful. The obvious goal for developers was to have a single unified build system. The developers of CMake had experience with two approaches of solving the unified build system problem.

One approach was the VTK build system of 1999. That system consisted of a configure script for Unix and an executable called `pcmaker` for Windows. `pcmaker` was a C program that read in Unix Makefiles and created NMake files for Windows. The binary executable for `pcmaker` was checked into the VTK CVS system repository. Several common cases, like adding a new library, required changing that source and checking in a new binary. Although this was a unified system in some sense, it had many shortcomings.

The other approach the developers had experience with was a `gmake` based build system for TargetJr. TargetJr was a C++ computer vision environment originally developed on Sun workstations. Originally TargetJr used the `imake` system to create Makefiles. However, at some point, when a Windows port was needed, the `gmake` system was created. Both Unix compilers and Windows compilers could be used with this `gmake`-based system. The system required several environment variables to be set prior to running `gmake`. Failure to have the correct environment caused the system to fail in ways that were difficult to debug, especially for end users.

Both of these systems suffered from a serious flaw: they forced Windows developers to use the command line. Experienced Windows developers prefer to use integrated development environments (IDEs). This would encourage Windows developers to create IDE files by hand and contribute them to the project, creating the dual build system again. In addition to the lack of IDE support, both of the systems described above made it extremely difficult to combine software projects. For example, VTK had very few modules for reading images mostly because the build system made it very difficult to use libraries like libtiff and libjpeg.

It was decided that a new build system would be developed for ITK and C++ in general. The basic constraints of the new build system would be as follows:

- Depend only on a C++ compiler being installed on the system.
- It must be able to generate Visual Studio IDE input files.
- It must be easy to create the basic build system targets, including static libraries, shared libraries, executables, and plugins.
- It must be able to run build time code generators.
- It must support separate build trees from the source tree.
- It must be able to perform system introspection, i.e., be able to determine automatically what the target system could and could not do.
- It must do dependency scanning of C/C++ header files automatically.
- All features would need to work consistently and equally well on all supported platforms.

In order to avoid depending on any additional libraries and parsers, CMake was designed with only one major dependency, the C++ compiler (which we can safely assume we have if we're building C++ code). At the time, building and installing scripting languages like Tcl was difficult on many popular UNIX and Windows systems. It can still be an issue today on modern supercomputers and secured computers with no Internet connection, so it can still be difficult to build third-party libraries. Since the build system is such a basic requirement for a package, it was decided that no additional dependencies would be introduced into CMake. This did limit CMake to creating its own simple language, which is a choice that still causes some people to dislike CMake. However, at the time the most popular embedded language was Tcl. If CMake had been a Tcl-based build system, it is unlikely that it would have gained the popularity that it enjoys today.

The ability to generate IDE project files is a strong selling point for CMake, but it also limits CMake to providing only the features that the IDE can support natively. However, the benefits of providing native IDE build files outweigh the limitations. Although this decision made the development of CMake more difficult, it made the development of ITK and other projects using CMake much easier. Developers are happier and more productive when using the tools they are most familiar with. By allowing developers to use their preferred tools, projects can take best advantage of their most important resource: the developer.

All C/C++ programs require one or more of the following fundamental building blocks of software: executables, static libraries, shared libraries, and plugins. CMake had to provide the ability to create these products on all supported platforms. Although all platforms support the creation of those products, the compiler flags used to create them vary greatly from compiler to compiler and platform to platform. By hiding the complexity and platform differences behind a simple command in CMake, developers are able to create them on Windows, Unix and Mac. This ability allows developers to focus on the project rather than on the details of how to build a shared library.

Code generators provide added complexity to a build system. From the start, VTK provided a system that automatically wrapped the C++ code into Tcl, Python, and Java by parsing the C++ header files, and automatically generating a wrapping layer. This requires a build system that can build a C/C++ executable (the wrapper generator), then run that executable at build time to create more C/C++ source code (the wrappers for the particular modules). That generated source code must then be compiled into executables or shared libraries. All of this has to happen within the IDE environments and the generated Makefiles.

When developing flexible cross-platform C/C++ software, it is important to program to the features of the system, and not to the specific system. Autotools has a model for doing system introspection which involves compiling small snippets of code, inspecting and storing the results of that compile. Since CMake was meant to be cross-platform it adopted a similar system introspection technique. This allows developers to program to the canonical system instead of to specific systems. This is important to make future portability possible, as compilers and operating systems change over time. For example, code like this:

```
#ifdef linux
// do some linux stuff
#endif
```

Is more brittle than code like this:

```
#ifdef HAS_FEATURE
// do something with a feature
#endif
```

Another early CMake requirement also came from autotools: the ability to create build trees that are separate from the source tree. This allows for multiple build types to be performed on the same source tree. It also prevents the source tree from being cluttered with build files, which often confuses version control systems.

One of the most important features of a build system is the ability to manage dependencies. If a source file is changed, then all products using that source file must be rebuilt. For C/C++ code, the header files included by a `.c` or `.cpp` file must also be checked as part of the dependencies. Tracking down issues where only some of the code that should be compiled actually gets compiled as a result of incorrect dependency information can be time consuming.

All of the requirements and features of the new build system had to work equally well on all supported platforms. CMake needed to provide a simple API for developers to create complicated software systems without having to understand platform details. In effect, software using CMake is outsourcing the build complications to the CMake team. Once the vision for the build tool was created with the basic set of requirements, implementation needed to proceed in an agile way. ITK needed a build system almost from day one. The first versions of CMake did not meet all of the requirements set out in the vision, but they were able to build on Windows and Unix.

5.2. How CMake Is Implemented

As mentioned, CMake's development languages are C and C++. To explain its internals this section will first describe the CMake process from a user's point of view, then examine its structures.

5.2.1. The CMake Process

CMake has two main phases. The first is the "configure" step, in which CMake processes all the input given to it and creates an internal representation of the build to be performed. Then next phase is the "generate" step. In this phase the actual build files are created.

Environment Variables (or Not)
In many build systems in 1999, and even today, shell level environment variables are used during the build of a project. It is typical that a project has a PROJECT_ROOT environment variable that points to the location of the root of the source tree. Environment variables are also used to point to optional or external packages. The trouble with this approach is that for the build to work, all of these external variables need to be set each time a build is performed. To solve this problem CMake has a cache file that stores all of the variables required for a build in one place. These are not shell or environment

variables, but CMake variables. The first time CMake is run for a particular build tree, it creates a `CMakeCache.txt` file which stores all the persistent variables for that build. Since the file is part of the build tree, the variables will always be available to CMake during each run.

The Configure Step

During the configure step, CMake first reads the `CMakeCache.txt` if it exists from a prior run. It then reads `CMakeLists.txt`, found in the root of the source tree given to CMake. During the configure step, the `CMakeLists.txt` files are parsed by the CMake language parser. Each of the CMake commands found in the file is executed by a command pattern object. Additional `CMakeLists.txt` files can be parsed during this step by the `include` and `add_subdirectory` CMake commands. CMake has a C++ object for each of the commands that can be used in the CMake language. Some examples of commands are `add_library`, `if`, `add_executable`, `add_subdirectory`, and `include`. In effect, the entire language of CMake is implemented as calls to commands. The parser simply converts the CMake input files into command calls and lists of strings that are arguments to commands.

The configure step essentially "runs" the user-provided CMake code. After all of the code is executed, and all cache variable values have been computed, CMake has an in-memory representation of the project to be built. This will include all of the libraries, executables, custom commands, and all other information required to create the final build files for the selected generator. At this point, the `CMakeCache.txt` file is saved to disk for use in future runs of CMake.

The in-memory representation of the project is a collection of targets, which are simply things that may be built, such as libraries and executables. CMake also supports custom targets: users can define their inputs and outputs, and provide custom executables or scripts to be run at build time. CMake stores each target in a `cmTarget` object. These objects are stored in turn in the `cmMakefile` object, which is basically a storage place for all of the targets found in a given directory of the source tree. The end result is a tree of `cmMakefile` objects containing maps of `cmTarget` objects.

The Generate Step

Once the configure step has been completed, the generate step can take place. The generate step is when CMake creates the build files for the target build tool selected by the user. At this point the internal representation of targets (libraries, executables, custom targets) is converted to either an input to an IDE build tool like Visual Studio, or a set of Makefiles to be executed by `make`. CMake's internal representation after the configure step is as generic as possible so that as much code and data structures as possible can be shared between different built tools.

An overview of the process can be seen in Figure 5.1.

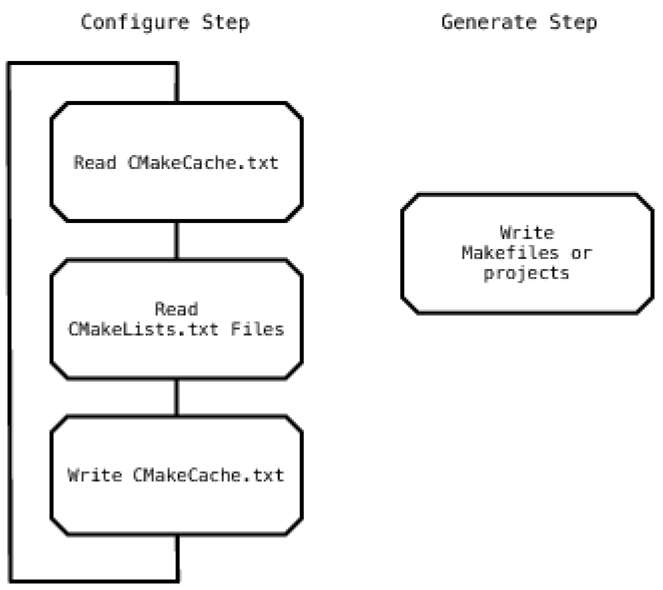
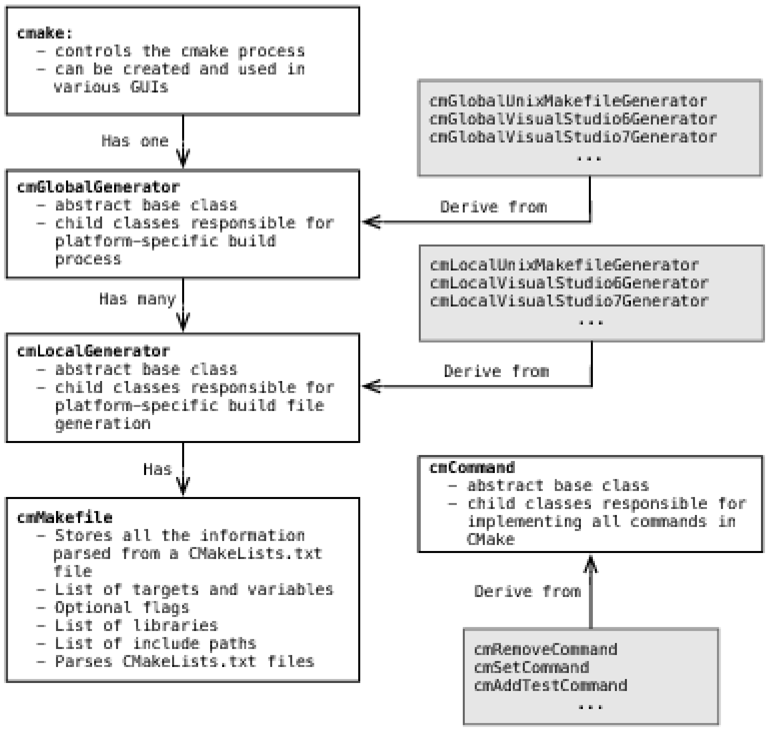


Figure 5.1: Overview of the CMake Process

5.2.2. CMake: The Code

CMake Objects

CMake is an object-oriented system using inheritance, design patterns and encapsulation. The major C++ objects and their relationships can be seen in Figure 5.2.



cmake:
– controls the cmake process
– can be created and used in various GUIs
Has one
cmGlobalGenerator
– abstract base class
– child classes responsible for platform-specific build process
Has many
cmLocalGenerator
– abstract base class
– child classes responsible for platform-specific build file generation
Has
cmMakefile
– Stores all the information parsed from a CMakeLists.txt file
– List of targets and variables
– Optional flags
– List of libraries
– List of include paths
– Parses CMakeLists.txt files
cmGlobalUnixMakefileGenerator
cmGlobalVisualStudio6Generator
cmGlobalVisualStudio7Generator
...
Derive from
cmLocalUnixMakefileGenerator
cmLocalVisualStudio6Generator
cmLocalVisualStudio7Generator
...
Derive from
cmCommand
– abstract base class
– child classes responsible for implementing all commands in CMake
Derive from
cmRemoveCommand
cmSetCommand
cmAddTestCommand
...

Figure 5.2: CMake Objects

The results of parsing each `CMakeLists.txt` file are stored in the `cmMakefile` object. In addition to storing the information about a directory, the `cmMakefile` object controls the parsing of the `CMakeLists.txt` file. The parsing function calls an object that uses a lex/yacc-based parser for the CMake language. Since the CMake language syntax changes very infrequently, and lex and yacc are not always available on systems where CMake is being built, the lex and yacc output files are processed and stored in the `Source` directory under version control with all of the other handwritten files.

Another important class in CMake is `cmCommand`. This is the base class for the implementation of all commands in the CMake language. Each subclass not only provides the implementation for the command, but also its documentation. As an example, see the documentation methods on the `cmUnsetCommand` class:

```
virtual const char* GetTerseDocumentation()
{
    return "Unset a variable, cache variable, or environment variable.";
}

/**
 * More documentation.
 */

virtual const char* GetFullDocumentation()
{
    return
      "  unset(<variable> [CACHE])\n"
```

```
        "Removes the specified variable causing it to become undefined.  "
        "If CACHE is present then the variable is removed from the cache "
        "instead of the current scope.\n"
        "<variable> can be an environment variable such as:\n"
        "  unset(ENV{LD_LIBRARY_PATH})\n"
        "in which case the variable will be removed from the current "
        "environment.";
}
```

Dependency Analysis

CMake has powerful built-in dependency analysis capabilities for individual Fortran, C and C++ source code files. Since Integrated Development Environments (IDEs) support and maintain file dependency information, CMake skips this step for those build systems. For IDE builds, CMake creates a native IDE input file, and lets the IDE handle the file level dependency information. The target level dependency information is translated to the IDE's format for specifying dependency information.

With Makefile-based builds, native make programs do not know how to automatically compute and keep dependency information up-to-date. For these builds, CMake automatically computes dependency information for C, C++ and Fortran files. Both the generation and maintenance of these dependencies are automatically done by CMake. Once a project is initially configured by CMake, users only need to run `make` and CMake does the rest of the work.

Although users do not need to know how CMake does this work, it may be useful to look at the dependency information files for a project. This information for each target is stored in four files called `depend.make`, `flags.make`, `build.make`, and `DependInfo.cmake`. `depend.make` stores the dependency information for all the object files in the directory. `flags.make` contains the compile flags used for the source files of this target. If they change then the files will be recompiled. `DependInfo.cmake` is used to keep the dependency information up-to-date and contains information about what files are part of the project and what languages they are in. Finally, the rules for building the dependencies are stored in `build.make`. If a dependency for a target is out of date then the depend information for that target will be recomputed, keeping the dependency information current. This is done because a change to a .h file could add a new dependency.

CTest and CPack

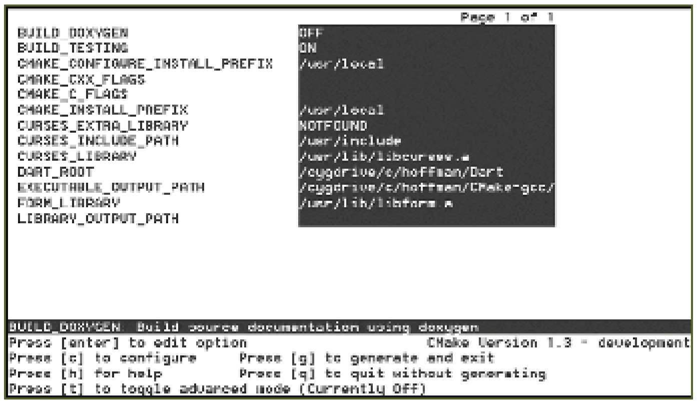
Along the way, CMake grew from a build system into a family of tools for building, testing, and packaging software. In addition to command line `cmake`, and the CMake GUI programs, CMake ships with a testing tool CTest, and a packaging tool CPack. CTest and CPack shared the same code base as CMake, but are separate tools not required for a basic build.

The `ctest` executable is used to run regression tests. A project can easily create tests for CTest to run with the `add_test` command. The tests can be run with CTest, which can also be used to send testing results to the CDash application for viewing on the web. CTest and CDash together are similar to the Hudson testing tool. They do differ in one major area: CTest is designed to allow a much more distributed testing environment. Clients can be setup to pull source from version control system, run tests, and send the results to CDash. With Hudson, client machines must give Hudson ssh access to the machine so tests can be run.

The `cpack` executable is used to create installers for projects. CPack works much like the build part of CMake: it interfaces with other packaging tools. For example, on Windows the NSIS packaging tool is used to create executable installers from a project. CPack runs the install rules of a project to create the install tree, which is then given to a an installer program like NSIS. CPack also supports creating RPM, Debian `.deb` files, `.tar`, `.tar.gz` and self-extracting tar files.

5.2.3. Graphical Interfaces

The first place many users first see CMake is one of CMake's user interface programs. CMake has two main user interface programs: a windowed Qt-based application, and a command line curses graphics-based application. These GUIs are graphical editors for the `CMakeCache.txt` file. They are relatively simple interfaces with two buttons, configure and generate, used to trigger the main phases of the CMake process. The curses-based GUI is available on Unix TTY-type platforms and Cygwin. The Qt GUI is available on all platforms. The GUIs can be seen in Figure 5.3 and Figure 5.4.

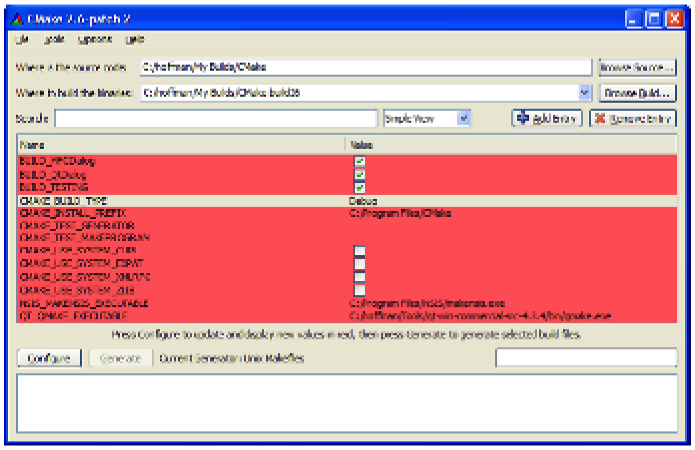


```
                                                             Page 1 of 1
 BUILD_DOXYGEN                        OFF
 BUILD_TESTING                        ON
 CMAKE_CONFIGURE_INSTALL_PREFIX       /usr/local
 CMAKE_CXX_FLAGS
 CMAKE_C_FLAGS
 CMAKE_INSTALL_PREFIX                 /usr/local
 CURSES_EXTRA_LIBRARY                 NOTFOUND
 CURSES_INCLUDE_PATH                  /usr/include
 CURSES_LIBRARY                       /usr/lib/libcurses.a
 DART_ROOT                            /cygdrive/c/hoffman/Dart
 EXECUTABLE_OUTPUT_PATH               /cygdrive/c/hoffman/CMake-gcc/
 FORM_LIBRARY                         /usr/lib/libform.a
 LIBRARY_OUTPUT_PATH




BUILD_DOXYGEN  Build source documentation using doxygen
Press [enter] to edit option                      CMake Version 1.3 - development
Press [c] to configure       Press [g] to generate and exit
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Figure 5.3: Command Line Interface



Where is the source code: C:/hoffman/My Builds/CMake

Where to build the binaries: C:/hoffman/My Builds/CMake build26

Search: Simple View

Name	Value
BUILD_MFCDialog	✓
BUILD_QtDialog	✓
BUILD_TESTING	✓
CMAKE_BUILD_TYPE	Debug
CMAKE_INSTALL_PREFIX	C:/Program Files/CMake
CMAKE_TEST_GENERATOR	
CMAKE_TEST_MAKEPROGRAM	
CMAKE_USE_SYSTEM_CURL	☐
CMAKE_USE_SYSTEM_EXPAT	☐
CMAKE_USE_SYSTEM_XMLRPC	☐
CMAKE_USE_SYSTEM_ZLIB	☐
NSIS_MAKENSIS_EXECUTABLE	C:/Program Files/NSIS/makensis.exe
QT_QMAKE_EXECUTABLE	C:/hoffman/Tools/qt-win-commercial-src-4.3.4/bin/qmake.exe

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Unix Makefiles

Figure 5.4: Graphics-based Interface

Both GUIs have cache variable names on the left, and values on the right. The values on the right can be changed by the user to values that are appropriate for the build. There are two types of variables, normal and advanced. By default the normal variables are shown to the user. A project can determine which variables are advanced inside the `CMakeLists.txt` files for the project. This allows users to be presented with as few choices as necessary for a build.

Since cache values can be modified as the commands are executed, the process of converging on a final build can be iterative. For example, turning on an option may reveal additional options. For this reason, the GUI disables the "generate" button until the user has had a chance to see all options at least once. Each time the configure button is pressed, new cache variables that have not yet been presented to the user are displayed in red. Once there are no new cache variables created during a configure run, the generate button is enabled.

5.2.4. Testing CMake

Any new CMake developer is first introduced to the testing process used in CMake development. The process makes use of the CMake family of tools (CMake, CTest, CPack, and CDash). As the code is developed and checked into the version control system, continuous integration testing machines automatically build and test the new CMake code using CTest. The results are sent to a CDash server which notifies developers via email if there are any build errors, compiler warnings, or test failures.

The process is a classic continuous integration testing system. As new code is checked into the CMake repository, it is automatically tested on the platforms supported by CMake. Given the large number of compilers and platforms that CMake supports, this type of testing system is essential to the development of a stable build system.

For example, if a new developer wants to add support for a new platform, the first question he or she is asked is whether they can provide a nightly dashboard client for that system. Without constant testing, it is inevitable that new systems will stop working after some period of time.

5.3. Lessons Learned

CMake was successfully building ITK from day one, and that was the most important part of the project. If we could redo the development of CMake, not much would change. However, there are always things that could have been done better.

5.3.1. Backwards Compatibility

Maintaining backwards compatibility is important to the CMake development team. The main goal of the project is to make building software easier. When a project or developer chooses CMake for a build tool, it is important to honor that choice and try very hard to not break that build with future releases of CMake. CMake 2.6 implemented a policy system where changes to CMake that would break existing behavior will warn but still perform the old behavior. Each `CMakeLists.txt` file is required to specify which version of CMake they are expecting to use. Newer versions of CMake might warn, but will still build the project as older versions did.

5.3.2. Language, Language, Language

The CMake language is meant to be very simple. However, it is one of the major obstacles to adoption when a new project is considering CMake. Given its organic growth, the CMake language does have a few quirks. The first parser for the language was not even lex/yacc based but rather just a simple string parser. Given the chance to do the language over, we would have spent some time looking for a nice embedded language that already existed. Lua is the best fit that might have worked. It is very small and clean. Even if an external language like Lua was not used, I would have given more consideration to the existing language from the start.

5.3.3. Plugins Did Not Work

To provide the ability for extension of the CMake language by projects, CMake has a plugin class. This allows a project to create new CMake commands in C. This sounded like a good idea at the time, and the interface was defined for C so that different compilers could be used. However, with the advent of multiple API systems like 32/64 bit Windows and Linux, the compatibility of plugins became hard to maintain. While extending CMake with the CMake language is not as powerful, it avoids CMake crashing or not being able to build a project because a plugin failed to build or load.

5.3.4. Reduce Exposed APIs

A big lesson learned during the development of the CMake project is that you don't have to maintain backward compatibility with something that users don't have access to. Several times during the development of CMake, users and customers requested that CMake be made into a library so that other languages could be bound to the CMake functionality. Not only would this have fractured the CMake user community with many different ways to use CMake, but it would have been a huge maintenance cost for the CMake project.

Footnotes

1. `http://www.itk.org/`