Build System Issues in Multilanguage Software

Transcript

Presented September 25, 2012 at the 28th IEEE International Conference on Software Maintenance (ICSM) in Riva del Garda, Italy
Build System Issues in Multilanguage Software

Andrew Neitsch, University of Alberta
Kenny Wong, University of Alberta
Michael W. Godfrey, University of Waterloo

http://andrew.neitsch.ca/build-issues-icsm2012

Giuliano Antoniol: [I’m Giuliano] Antoniol, and I’m chairing the session Analysis of Build Systems.

We have three papers so we need to stick on the 25-minute mark, because otherwise we cannot make it. And the first paper, it’s called, Andrew Neitsch, Kenny Wong, Mike Godfrey—Build System Issues in Multilanguage Software.

Andrew Neitsch: Thank you.

There’s a URL here if you’d like to follow along in the slides.
Build systems

Transform source code into running programs

Build systems. They’re things that transform source code into running programs. So you might have a Makefile that turns C source files into a binary you can actually run. Now, a lot of you are thinking, isn’t this …
Solved problem?

Java builds in Eclipse

Slide 3

... a solved problem? And in many ways, it is. If you’re building Java software in Eclipse, there’s a Build Automatically menu option. It’s checked by default. All changes to your source files are automatically and instantly rebuilt.
Solved problem?

C/C++ builds in Eclipse

And Eclipse also has support for building C and C++. It’s not instant, but it’s still totally automatic.
Solved problem?
Not when mixing Java and C/C++

However, if you try and mix the two, it turns out that Eclipse doesn’t have any support for integrating Java and C++. You can sort of hack it with a Makefile that pretends to be one of Eclipse’s auto-generated Makefiles, but it goes from being a totally automatic solved problem to one where you have to be an expert in both Java and C build systems and build tools before you can write a line of application code that uses the two of them together.

This is empirical evidence that, while this may be a solved problem, in some contexts, that’s not necessarily the case for . . .
Multilanguage software

Software written in multiple programming languages, in which the parts written in different languages are both necessary and interdependent in the implementation of the software.

Slide 6

...multilanguage software. We need to distinguish between the case where there’s a package that uses multiple programming languages and there’s a package that uses multiple programming languages together. For example, if there’s a database server written in C, we wouldn’t consider it to be multilanguage just because it happens to have a Java client library available. However, something like Firefox where the networking stuff is all done in C++ and the UI is all done in JavaScript, and the two parts are constantly talking to each other, we would consider that multilanguage. So
we have this definition to distinguish between the two cases.
Research questions

1. What are the major issues in building multilanguage software?
2. How can they be addressed?
3. Why do they occur?

Slide 7

Now, our research questions are: What are the major issues in building multilanguage software? How can they be addressed? And, why do they occur?
Qualitative methodology

1. Select case studies
2. Try to build them
3. Note build problems encountered and build system features that prevent problems
4. Compare and analyze commonalities

So, to address this question, we used a qualitative case study research methodology. We select five case studies, we try to build them, we note the build problems we encounter and the build system features that prevent problems. Then, once we have those observations, we compare and analyze the commonalities.
Key finding

There are commonalities among build problems that could be systematically addressed.

Slide 9

Our key finding is that there are commonalities among build problems that could be systematically addressed. So it is actually hard to build multilanguage software. If you just take a random piece of multilanguage software, you’re going to run into problems. But it’s not that every package has its own isolated problems that need to be fixed in very specific ways. There are general themes and commonalities among the build problems that could be systematically addressed.
Slide 10

The investigations we did to arrive at this finding are . . .
Multilanguage case studies

Five open-source multilanguage packages selected at random from Ubuntu

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>synopsis</td>
<td>Source code documentation</td>
<td>C++, Python</td>
</tr>
<tr>
<td>python3.0</td>
<td>Programming language</td>
<td>C, Python</td>
</tr>
<tr>
<td>gnat-gps</td>
<td>IDE</td>
<td>Ada, Python, C</td>
</tr>
<tr>
<td>axiom</td>
<td>Computer algebra</td>
<td>Lisp, Scratchpad, C</td>
</tr>
<tr>
<td>ruby-prof</td>
<td>Profiler</td>
<td>Ruby, C</td>
</tr>
</tbody>
</table>

We took five open-source multilanguage packages. We selected them at random from Ubuntu. We could have just used Firefox and stuff like that but we wanted to look at some things that hadn’t been looked at before. So, there are five packages. There’s a source code documentation tool, there’s the Python interpreter, there’s the gnat-gps interactive development environment for Ada, there’s a computer algebra system, and a Ruby profiler. So there’s a variety of languages, and these are of various different sizes, and gnat-gps is actually industrially-developed, and axiom used to...
be an IBM product but is now open-source. These are the five systems we looked at.
Case study questions

- Package purpose
- Multilanguage architecture
- Build system
- Build and rebuild problems

For each of them we performed deep case studies. We looked at what’s the purpose of the package, what does it do, what’s it for, what’s the multilanguage architecture—which parts are in which languages, and how do they interact with each other? And then we analyzed the build system, both its implementation and structure. And finally we tried to build these systems, and we encountered problems, and we analyzed what the causes of the problems were and proceeded on.
I only have time to give one example, so I’m going to go over the IDE called gnat-gps. It’s primarily for Ada source code, and it’s a fairly typical-looking IDE. There’s a project navigator on the left, and there’s a big editor on the right, and then there’s the toolbars and menus for doing all sorts of other things. For example there’s this Build → Clean menu.
Excerpt of gnat-gps’s conceptual architecture

Now the conceptual architecture—this is an excerpt of it. And I’ve colour-coded it where Ada is green, Python is purple, and C is gray. Ada is an object-oriented language. It’s got objects and classes and that sort of thing. So, the general conceptual architecture is, there are a bunch of classes, and they all call each other and there’s a web of objects interacting. There’s some C code, but it’s more low-level stuff that doesn’t call back into the Ada source code, and there’s also some Python stuff. There’s the scripting module, and most of the other modules in gnat-gps register themselves
with the scripting module. And the scripting module loads Python scripts and gives them access to the full object model of the interactive development environment. Some of them are kind of trivial, like changing shortcut keys, but some of them are actually key features. This Build → Clean menu, that you just saw, that’s implemented in Python, and these Python scripts have full access to the object model.
So how does it actually build? To describe the build system, we’re using a Build-Time View diagram. It shows three different architectural views of the system, with the artifacts in those different views, and the relationships between them.

There’s a bit of Python source code, there’s a bit of C source code, but most of the source code is this big tangled hierarchy of directories. They all have some Ada sources, and some of them also have some C source code. Each of them has a Makefile, and an Ada project file, and any particular module includes the Makefiles and the
project files for all of the directories that
the arrows are pointing to, so there’s a very
tangled hierarchy going on here.
And, the way it’s built is, in each source directory, [there are] Ada sources and C sources. [First,] an Ada compiler comes along, possibly references some external libraries, and it produces some intermediate object files.
Then, the C compiler takes the C sources, and compiles them, possibly using some external libraries as well, and produces more intermediate object files.
And once this happens with every directory, then, all the object files are linked together by the linker to produce an executable. So at this point you can actually run the IDE, but it’ll be missing some features until the rest of it is built.
Python source code is just text, so it’s copied to the execution environment.
And Source Navigator is built by a C compiler to produce some auxiliary executables, and gnat-gps runs them and reads their output.
So this is the build system for this case study. Well, it’s the build system in theory—
in practice you run into all sorts of problems when you actually try to build it. We have red warning triangles where, when you actually try to build this, the build breaks, you have to do some debugging, you have to figure out some stuff to actually get it to work. And some of these are actually covering multiple problems. We didn’t actually get this build system to actually work in the end. What happened was, I was running into so many problems using the original vendor build system on Ubuntu. But Ubuntu also provides binaries, and they have patches to make the software build
specifically on Ubuntu. And, after running into a bunch of problems, I took a look at what Ubuntu does—how does Ubuntu patch this to actually work?
And, surprisingly, what I found was, instead of patching the build system, the Ubuntu maintainer just threw it away and wrote his own. And he replaced that big tangled hierarchy of all these directories, of all these Makefiles including each other, with a build system that just had a single Makefile and a single Ada project file for the main source code. So, all the Ada sources are compiled in one step, all the C sources are compiled in one step, and they’re all linked together, and there are still some problems, but it’s a build system that actually works.

We performed five deep case studies like
this, where we examined the purpose of the package, its architecture, its build system structure, and then the actual problems we encountered.
Patterns and anti-patterns

Now, we synthesized our observations across all these different case studies into what we call build patterns and anti-patterns. Whenever we saw similar build problems in different packages, we put them on the same row of a table as different red triangles, and then we looked at other packages and we said, “Why doesn’t this package have the same problem as well?” And, often we identified what we call build features that avoided those problems. And then, we synthesized all these observations into these patterns and anti-patterns.

### Slide 17

---

Patterns and anti-patterns

<table>
<thead>
<tr>
<th>build problem</th>
<th>build feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>△</td>
<td></td>
</tr>
<tr>
<td>□</td>
<td></td>
</tr>
</tbody>
</table>

#### Anti-Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>synopsis</th>
<th>python3.0</th>
<th>gnat-gps</th>
<th>axiom</th>
<th>ruby-prof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filename Collision</td>
<td>△</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Installation Required</td>
<td>△</td>
<td>□</td>
<td>△</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unverified Third-Party Software</td>
<td>△</td>
<td>□</td>
<td>△</td>
<td>□</td>
<td></td>
</tr>
<tr>
<td>Ignored Error</td>
<td>□</td>
<td></td>
<td>□</td>
<td>△</td>
<td></td>
</tr>
<tr>
<td>Incorrect Dependencies</td>
<td>□</td>
<td>△</td>
<td>□</td>
<td>□</td>
<td>△</td>
</tr>
</tbody>
</table>

#### Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Build-Free Extensibility</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Object-Oriented Builds</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>
Anti-pattern: Incorrectly-specified rebuild dependencies

Consequences
Up-to-date artifacts are needlessly rebuilt and/or out-of-date artifacts are not rebuilt

Remedies
Use a tool to extract rebuild dependencies such as header files

Conclusion
Not a major issue in this study—more for rebuilds, not builds

So, I’m going to go over a few of them.

The first one is incorrect dependencies. It’s commonly thought that if you have build problems, it’s because your dependencies aren’t specified correctly. For example, you’ve got the wrong header files listed in your Makefiles or something like that. However, we found that, when you actually define what those dependencies are, they’re rebuild dependencies. The consequences of having incorrect dependencies in your build system are that you waste time rebuilding stuff that doesn’t need to be rebuilt, or you make changes that don’t get rebuilt when
you execute the build system.

And there’s a fairly well-known remedy. You use a tool to just automatically extract the dependencies instead of specifying them yourself. However, we found that a major issue in building this software wasn’t with rebuilds, it was getting it to build initially in the first place. And the dependencies specified didn’t matter as much.
Anti-pattern: Unverified third-party software

Consequences
The build fails due to third-party software problems, but the current package seems at fault.

Evidence
Unbuildable case study vs. autotools

Remedies
Test third-party functionality at configuration-time, especially if there are known bugs.

Another anti-pattern I want to talk about is unverified third-party software. This is when you’re trying to build one particular package, and it relies on functionality from a third-party package. And the third-party package has some sort of problem that prevents your build from working at all, but it makes it seem like there’s a problem in your package. We saw this in a few different case studies, where the build failed, and when we debugged it, it turned out to be a problem with a third-party package, and sometimes it would turn out that buried deep in the documentation was a little note:
“Oh, by the way, this doesn’t work with some versions of this other program.”

So, when we examined other case studies to find out what they did, we found that, before autotools will build any piece of C software, it will try to build a “Hello, world” program with the compiler, and run it, and if it can’t build and run a “Hello, world” program, it will just put up an error message and it will say, “I’m not going to, I can’t build your software, until you give me a valid C compiler.” And the remedy is to use a similar test like that before the build actually starts, especially when there are known bugs.
One package that mentions somewhere that it had some problems with a third-party library had patches for fourteen different versions of that library in its source code, but at no point did it actually try running that third-party tool to see whether the functionality it needed worked.
Pattern: Build-free extensibility

Description  The project can be extended and customized without rebuilding the software

Evidence  Difficult-to-build case studies

Consequences  Build systems are avoided

Remedies  Incorporate dynamic extensibility, such as scripting

Slide 20

Now, a pattern instead of an anti-pattern. It is build-free extensibility.

This is when a project can be customized and extended without rebuilding the software. Now, if you think about it, every time you write some Python source code, or you write a Python module, you’re actually extending what Python can do. But you don’t need to rebuild Python every time you write a little bit of Python. That would be very annoying. And Python had one of the better build systems we’ve looked at in this case study, but also when there are difficult-to-build case studies, this pattern...
stops most people from having to actually deal with these sorts of build problems. As long as people can get binaries from somewhere, if there are well-thought-out extension and customization mechanisms, then they’re able to still do a lot of what they would normally do with a build system but without having to use the build.

So someone still has to be able to build the software, you still have to be able to produce binaries, but it’s an end-run around most people encountering those build problems.

And the remedy is to incorporate dynamic extensibility mechanisms such as scripting
into your software.
Causes of build problems

- Secondary nature of builds
- Technical debt

Could be addressed through build tools and build frameworks, and object-oriented builds

---

Slide 21

So, what cases build problems? Well, in one sense they’re kind of secondary. When a developer sits down to work every day, are they going to fix a major bug, are they going to add features that customers are demanding, or are they going to make the build system nicer? Well, they’re probably going to do the more important stuff, and a lot of projects are comfortable living with the technical debt of having an imperfect build system.

So the patterns and anti-patterns, while individual practitioners can apply them, we think that they would be better used when
incorporated into build tools and build frameworks to systematically address these problems. Also, object-oriented builds, where you are able to create build plugins, and distribute them, could make it less secondary. So if there’s a problem in your build script, and you have to change a shell fragment in a Makefile that’s only ever going to fix that Makefile, it’s not a very high priority. However, if you can solve the problem in a general way and release it as a plugin to be used by all sorts of build systems in all sorts of other projects, then it becomes less secondary.
Leaking abstractions

The build system has properties whose presence may only be explainable by reference to the application and implementation domains.

Another potential cause we identified is something I call leaking abstractions. Now this is when the build system has properties whose presence may only be explainable by reference to the application and implementation domains.

You’d expect that the build system for a particular piece of software might have some specialized functionality for maybe creating a code generator that that’s only used by it. But this is talking about other sorts of abstractions. And I’m going to give an example that shows what I mean.
Leaking abstractions

difficult to build, but business goals help explain build structure

Slide 23

This is the build system for gnat-gps. It has this very complicated hierarchy of all these different source directories that all include each other, and it was very error-prone, and the Ubuntu maintainer just threw it away and replaced it with a single Makefile, called it “evil” and “brittle” and— So, why is this here?

Is this because the developers who made this had never read Recursive Make Considered Harmful? No. When you just look at the build system, this seems like a very bad idea. But when you take a step up and you look at the software project itself,
it turns out that this is a method of enforcing architectural constraints. The company that develops this software wants the ability to take individual directories, and once the functionality inside them is mature, to release them as independent Ada libraries. Now, they develop Ada tools and Ada libraries; having mature Ada libraries with useful functionality makes Ada more attractive for development. So, this structure is hard to build, but in the context of business goals of the company that makes it, it explains a lot. So this abstraction that we want individual pieces to be able to have very explicit dependencies, and be compilable separately, leaks down into the
build system from the sort-of business
domain and the implementation domain.
Leaking abstractions: Python

# python -c 'import this'
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
**Simple is better than complex.**
Complex is better than complicated.
**Flat is better than nested.**
Sparse is better than dense.
Readability counts.
Special cases aren’t special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
...

Another example is Python. If you type “import this” into Python, you get a little poem about how great Python is. It’s basically the design philosophy of Python. Two of the things it says are, “Flat is better than nested,” and “Simple is better than complex.” Part of the Python build system is implemented as Makefiles and shell scripts. However, those parts, when you look at when, have a lot of the same properties of idiomatic Python code. People are choosing to use flat structures instead of nested structures, so there’s just one Makefile instead of a hierarchy. And there
are very simple commands instead of complex ones. So, some of these good features from the mindset of the Python developer show up in the build system as well.
Leaking abstractions: Python

# python -c 'import this'
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren’t special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
...

Slide 24b

However, there’s this one thing in here: “Explicit is better than implicit.” And, usually this is good; sometimes it’s going to be kind of annoying, like typing ‘self’ in every class definition. And in Python there’s this general avoidance of ‘magic’ and sometimes you have to do stuff by hand something like Ruby would do automatically for you. And, this same negative aspect of Python’s development style is found in the build system, inside the Makefiles. Instead of using a tool to extract header files, they explicitly have a list of all of them that they maintain by hand. And
instead of having one list for different parts with different object files that depends on different header files there’s just one list that’s used for everything to keep it simple. And unfortunately this list is kind of wrong. So when you change almost any header file in Python, it forces a whole rebuild of all of Python. However, there are a few header files that are missed in this list, and if you change them, then your changes are just totally ignored. So, this set of abstractions, the developer mindset from the application implementation domain, is leaking down into the build system and affecting how that’s developed as well.

This relates to previous work by McIntosh
and others, who have evidence that a centralized build ownership style reduces overall build system maintenance. So, they’re saying that, when you have a team of build experts who work only on the build system and not on other stuff, then that reduces your overall build maintenance. And, so, this idea of leaking abstractions is suggesting a possible cause for that, in that, when there’s a team of people who work only on the build system, they’re not being exposed to these application implementation domain concepts and adding them to the build system where they may not be appropriate.
Conclusions

So, conclusions.
1. What are the major issues in building multilanguage software?

Getting the software to build at all is the major issue.
2. How can they be addressed?

Build patterns and anti-patterns
Systematically addressed through build tools and frameworks

Now, how can build problems be addressed? Well, there are commonalities—so build patterns and anti-patterns, as I said, individual practitioners could use them, but they’d be better used if they were incorporated into build tools and frameworks to fix them systematically.
3. Why do they occur?

Secondary nature of builds, technical debt, leaking abstractions

And then, why do build problems occur? Well, they’re sort of secondary in nature for projects. People are ok with the technical debt of having an imperfect build system. And there’s this idea of leaking abstractions that people are thinking more about the software, the application and the implementation when they’re writing the build system, and these other concepts are leaking.
So, to summarize, build systems aren’t necessarily a solved problem in the context of multilanguage software. We took five open-source case studies. We performed deep case studies on each of them, looking at the purpose, the architecture, the build system, and then the build problems, and we synthesized all of our observations into build patterns and anti-patterns.

Thank you. [Applause]

Giuliano Antoniol: So thanks for the presentation and for being on time. Question—Bram?
Bram Adams: I like the term “leaking abstractions,” it’s really a nice thing, because I think Mozilla for example has the same problem.

Andrew Neitsch: Ok.

Bram Adams: It’s universal. And with that, there’s a question, the question is, did you also look at how the build system evolves of these multilanguage—?

Andrew Neitsch: No. I just looked at one snapshot of each. There were a couple where I looked at multiple versions when the version I selected we actually just couldn’t build at all.
Mike Godfrey: It’s next year’s paper.

Andrew Neitsch: Yes.

Giuliano Antoniol: The gentleman there.

Roman Suvorov: Also with respect to leaking abstractions, don’t you think it’s necessary for the build system experts to know the overall architecture? It’s almost impossible to really separate the two. Especially given how performance is critical to most build systems?

Andrew Neitsch: Eclipse doesn’t care for Java. Eclipse has the exact same generated build system no matter what you’re building. It doesn’t care if it’s a database or
a web browser. It works fine, so there doesn’t seem to be a need for database or web browser abstractions to just be sucked in there too.

Giuliano Antoniol: The gentleman over there, please,

Chris Bird: I came a few minutes late, so I apologize if you’ve already answered this. Are these all Make-based build systems, or?

Andrew Neitsch: They all ended up using Make. They were randomly selected, but they all used Make. They also used other tools, in part, but they all had a bit of Make in them.
Chris Bird: So, is there room for—so I guess two questions: are there existing build systems other than make that handle this problem better? I mean obviously you didn’t look at them, but that you’ve heard about, or—is there room for that? I mean, I hear all the time about attempts to replace Make, but nothing really holds, and yet it seems like multilanguage projects are growing in popularity.

Andrew Neitsch: So, something like Ruby has a replacement for Make called Rake, which actually works really well, but, there’s a chain of libraries that ends up actually building one small part of it, and that generates one Makefile. It’s just weird
— and that was the part that had the problems! Because they leave all the dependencies out of the Makefile, because they’re just passing it off to some other tool, and— there is room for it. That particular package had only one automatically-generated Makefile from Rake and it could be replaced. So there is room to replace these.

Giuliano Antoniol: Other questions?

Audience member: Just clarifications— axiom or open-axiom?

Andrew Neitsch: axiom. And the particular version we used doesn’t run at all on the version of Ubuntu because there’s a . . .
Audience member: yup

Andrew Neitsch: ok

Giuliano Antoniol: Ok. So thanks again.

[applause]