

Build System Issues in Multilanguage Software

MSc Seminar Transcript

Edited for grammar and clarity

Andrew Neitsch

University of Alberta
Department of Computing Science

August 31, 2012

Build System Issues in Multilanguage Software

<http://andrew.neitsch.ca/msc>

Slide 1

Hello everyone. Thanks for coming today.
I'm going to present my master's thesis
work which is titled, "Build System Issues
in Multilanguage Software."

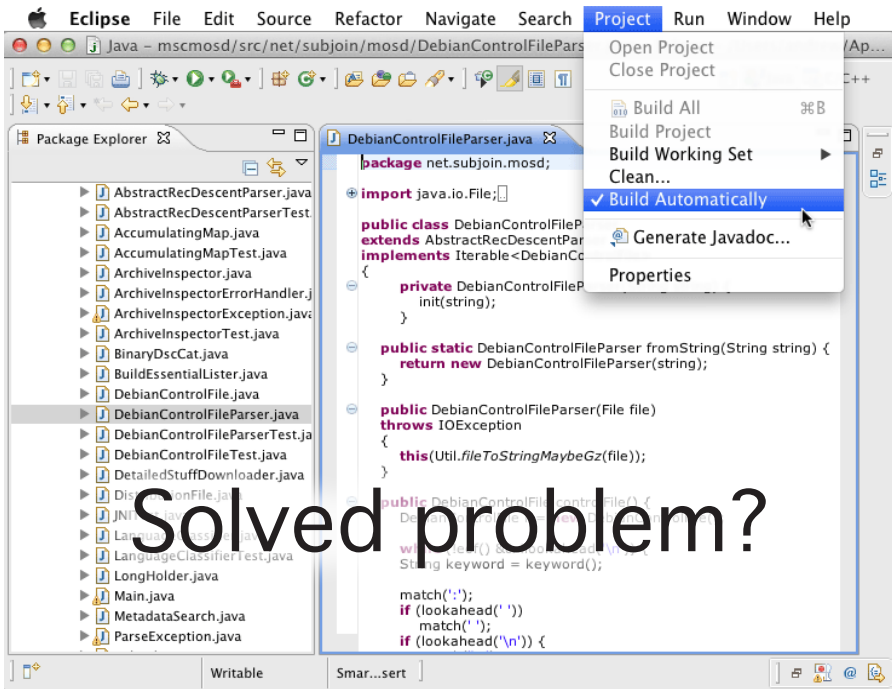
Build Systems:

Transform source code
into running programs

Slide 2

The first phrase in the title of my thesis is “Build Systems,” and build systems are things that transform source code into running programs. An example would be a Makefile that takes some C source code and produces an executable that you can run.

Now, you’re probably thinking, “Isn’t that a solved problem?”



Slide 3

Yeah, it kind of is, in some ways. For example, if you program Java code in Eclipse there's a Build Automatically menu option. It's checked by default, and every time you save a change to a source file, it just instantly recompiles the change. It's great. It runs. It's totally automatic.

So if this is a solved problem, why did I bother writing a thesis on it?

Build System Issues in Multilanguage Software:

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

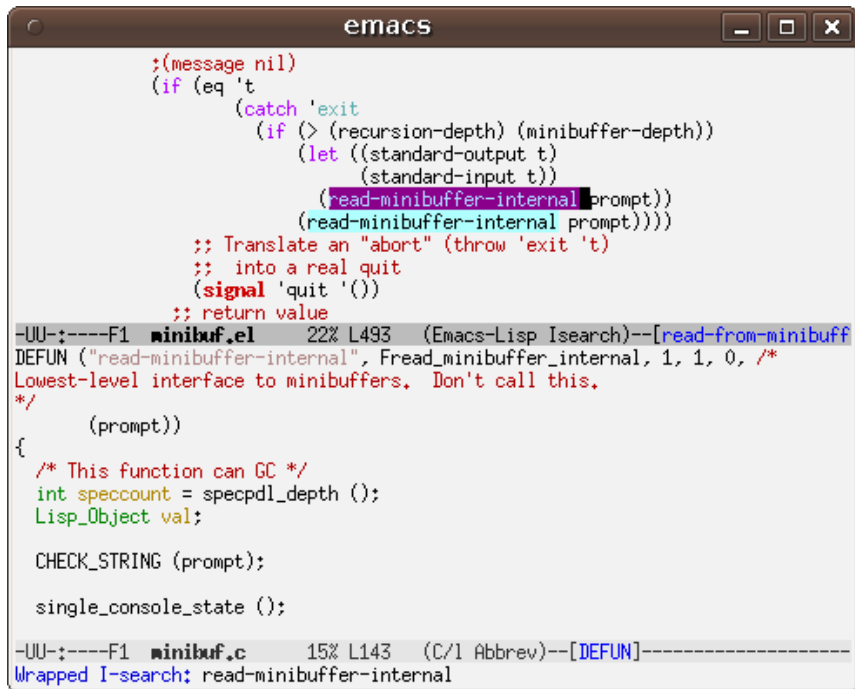
Slide 4

Well, there's another phrase in the title of the thesis, namely "multilanguage software." I'm going to look at build systems in the context of multilanguage software.

I have a definition of multilanguage software here. It's a little technical, but what it's intended to do is to distinguish between the case where there's a package that has source code in multiple programming languages, and there's a package that has source code in multiple programming languages where those programming languages are used together.

An example of something that has multiple programming languages, but isn't actually multilanguage software, would be a database that's written in C, for which there's a Java client library available. In that case, the database is actually written in C, it's not multilanguage software, even though there is both C and Java source code.

The definition I give to capture that is, "Software written in multiple programming languages, in which the parts written in different languages are both necessary and interdependent in the implementation."



```
;(message nil)
(if (eq 't
      (catch 'exit
        (if (> (recursion-depth) (minibuffer-depth))
            (let ((standard-output t)
                  (standard-input t))
              (read-minibuffer-internal prompt))
            (read-minibuffer-internal prompt))))
    ;; Translate an "abort" (throw 'exit 't)
    ;; into a real quit
    (signal 'quit '())
    ;; return value
- UU-:----F1 minibuf.el 22% L493 (Emacs-Lisp Isearch)--[read-from-minibuff
DEFUN ("read-minibuffer-internal", Fread_minibuffer_internal, 1, 1, 0, /*
Lowest-level interface to minibuffers. Don't call this.
*/
      (prompt))
{
  /* This function can GC */
  int speccount = specpdl_depth ();
  Lisp_Object val;

  CHECK_STRING (prompt);

  single_console_state ();

- UU-:----F1 minibuf.c 15% L143 (C/1 Abbrev)--[DEFUN]-----
Wrapped I-search: read-minibuffer-internal
```

Slide 5

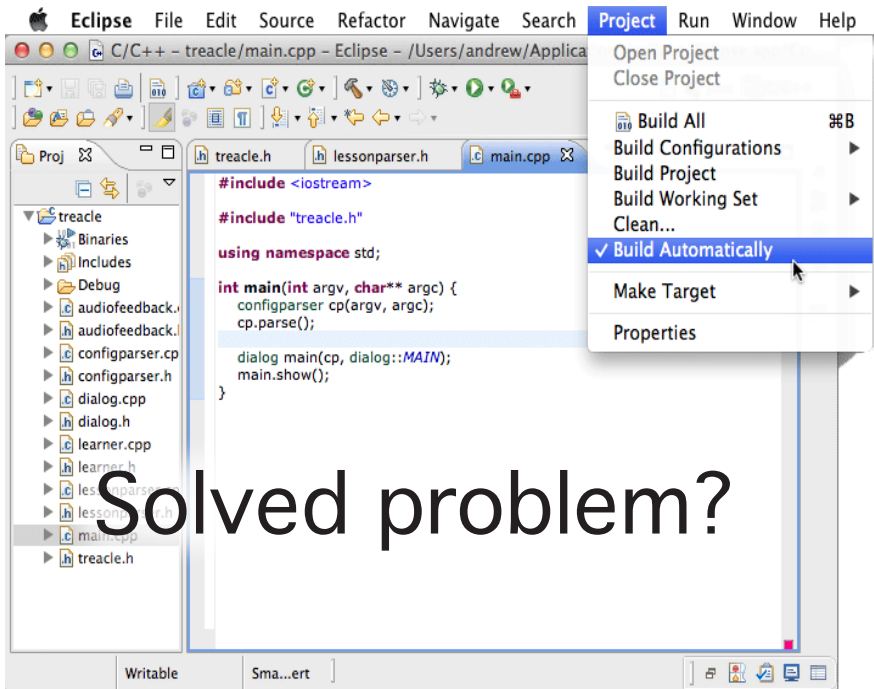
Here's an example of multilanguage software. This is the emacs editor, and it's looking at its own source code right now. Part of it's written in Lisp up here, and part of it's written in C down there, and the interesting thing is, this Lisp can freely call to C, and this C can freely call to Lisp. Any particular function can be implemented in either language. When you're developing code for emacs—whatever it is you're working on—you can use whichever language is best suited to the task at hand. At the individual function level you can change the implementation language of

functions just by rewriting them and moving them from a C file to a Lisp file or vice-versa, and recompiling. You don't need to change anything else, you just move functions around and it all works. It's got this really nice property that you can use whatever language is best-suited for the task.

Now, getting back to build systems—the build system for this, that enables this to happen, is really complicated. emacs actually has to include its own Lisp interpreter written in C to allow this to happen. And if you wanted to write a similar system, that mixed C and Lisp, before you could write a single line of

application code, you'd have to come up with a complicated build system that can mix both of these together.

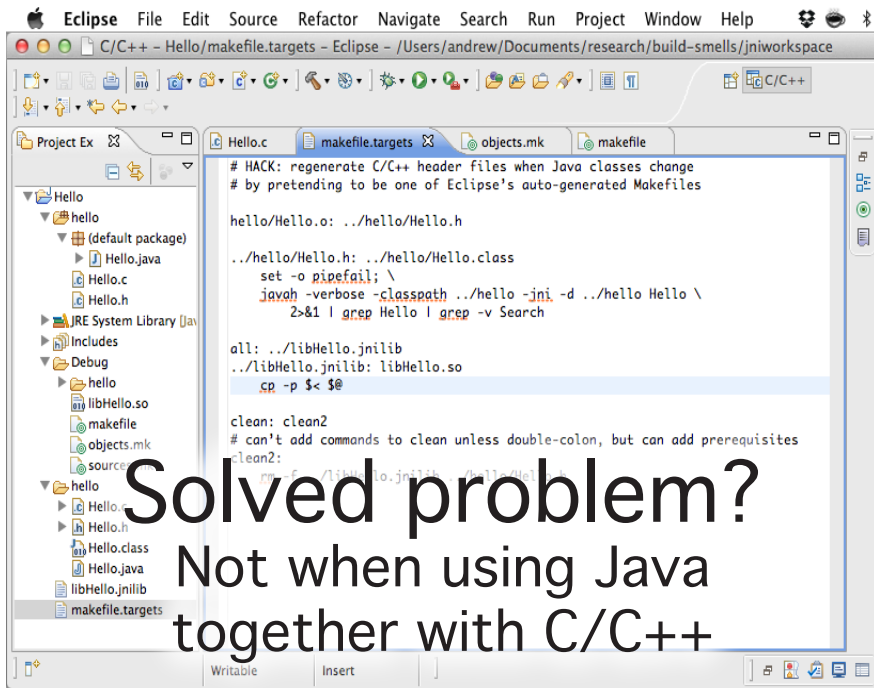
So, is building a solved problem for multilanguage software?



Slide 6

This is Eclipse again. This time it's building some C++ source code, and again there's a "Build Automatically" menu option that's checked by default. It's a little more complicated because you've got to wait for a bit for it to build—it's not instant the way Java is—but it still sort of seems like a solved problem.

Solved problem?



Slide 7

However, the moment you start using Java and C together, it gets really complicated, because while Eclipse has support for building Java, and it has support for building C, it doesn't have any support at all for building both of them together. You can sort of hack up the build system—there's a Makefile here that pretends to be part of Eclipse's Makefiles, and it adds some extra rules so that, when a Java file changes, the corresponding C header files are regenerated, but, it's ... it's really complicated. It goes from being a completely automatic, almost-instant

process, to being something where you have to understand the internals of both Java and C build systems, and you have to be able to integrate them together yourself.

So when you go from single-language to multilanguage software, building the software goes from being totally automatic to an extremely manual expert-level process.

Build systems for multilanguage
software are error-prone.

Slide 8

So, my thesis statement is that build systems for multilanguage software are error-prone. All the support for that statement is to come. And, there's a second part of the statement that ...

Build systems for multilanguage software are error-prone.

But there are commonalities that could be systematically addressed.

Slide 9

although build systems for multilanguage software are error-prone, there are commonalities among the problems that could be systematically addressed. That is, while build systems are error-prone for multilanguage software, it's not that each particular multilanguage package has its own unique problems; there are commonalities that could be systematically addressed.

Research questions

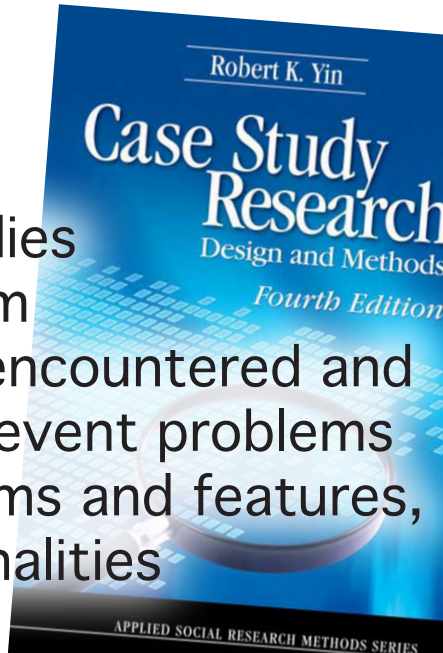
My specific research questions are

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

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

Qualitative Methodology

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



Slide 11

My thesis uses a qualitative methodology. I'm following this *Case Study Research: Design and Methods* book for guidelines. The basic procedure is to select some case studies, try to build them, note the build problems that I encounter and the build system features that prevent problems, and then I systematically analyze the commonalities among all the problems and features to produce my findings.

Contributions

- Filename-based selection procedure
- Five deep case studies of open-source multilanguage packages
- Build patterns and anti-patterns
- Error-proneness finding
- (Anti-)pattern uses, implications
- Abstraction “leakage” finding

Slide 12

The contributions of my thesis are, first of all, a filename-based selection procedure to actually find multilanguage software; five deep case studies of open-source multilanguage packages, where I analyze how each multilanguage package is built; the commonalities among the build problems for these case studies are analyzed into build patterns and anti-patterns; the finding I mentioned earlier that build systems for multilanguage software are error-prone; I discuss the uses and implications of these patterns and anti-patterns; and I also talk about what I call “leaking abstractions.”

Contributions

- Filename-based selection procedure
- Five deep case studies of open-source multilanguage packages
- Build patterns and anti-patterns
- Error-proneness finding
- (Anti-)pattern uses, implications
- Abstraction “leakage” finding

Slide 13

You'll see this contributions slide again because it's also the table of contents for this talk. To begin: the filename selection procedure.

Selection Goal: Five Multilanguage Packages

Could use, say, Mozilla, Emacs, &c.

Benchmark packages

Exploratory study

Systematic selection

Slide 14

The goal is to find five multilanguage packages to perform case studies on. Now, I could just take some well-known multilanguage packages like Mozilla or Emacs, things that I use and like. But while there's some value in using benchmark packages such as Mozilla for software engineering studies—specifically they allow people to compare results, and when you publish a paper on Mozilla, since everyone's done something with Mozilla, other people will understand the context well—this is an exploratory study. I didn't really want to look at stuff that a lot of people had looked

at before; however, I did want to be systematic about selecting the packages.

Selection Goal: Five Multilanguage Packages

It's possible to identify
multilanguage packages
semi-automatically

So I used a semi-automatic procedure to find multilanguage packages to perform cases studies on.

Procedure

- 1) Extract filenames
- 2) Classify by language
- 3) Discard single-language packages
- 4) Randomly select candidates
- 5) Review manually

The procedure used was to extract all the filenames for all the source code for all the packages in Ubuntu, and then classify them by language based on their filename. So, if it's a .c file, then I say, oh it's written in C, and if it's a .java file, then, oh, it's written in Java. Using that information, I discarded all the packages that couldn't be multilanguage because I only identified a single programming language. If there were only .c files, for example, I would say it can't be a multilanguage package. Then, from the remaining packages, I randomly selected candidates that had multiple

programming languages in their source code. Of those, like I said before, some of them could be databases that were written in C and had a Java client library available, which means they're not really multilanguage. So I manually inspected randomly-selected candidates until I had five multilanguage packages to look at.

Slide 17

Packages from Ubuntu 9.10
16 384 source

29GB compressed, 101GB source
30 minutes to extract
6.3M filenames
64MB cache file, 20s to load
3s to iterate over
On-the-fly class reloading

Here are some of the implementation details. Ubuntu has more than 16,000 source packages. All the source code is about 29 gigs compressed. When you recursively unarchive all the source code you end up with 101 gigs of source code. On my laptop here it takes about 30 minutes to crunch through all of that, get six million filenames, and squish them into a 64 meg cache file. It takes a bit to load into memory and it uses many gigs of ram, but the end result is that you can iterate over every filename in every source package in Ubuntu in about 3 seconds.

The tool I implemented for this uses both Java and C together. Java filename analysis classes, such as the one I used for finding multilanguage packages, can be reloaded on the fly. So, once the cache file is generated and loaded into memory, when you make a change to your filename analysis code in Eclipse, three seconds later you get your new results from all of Ubuntu.

These are just some implementation details that I thought were kind of cool.

Selection has variety along many dimensions, enabling useful comparisons

Package	Description	Languages	Industrial?	Age (years)	KSLOC	Committers
synopsis	Source code documentation	C++ Python	No	9	94	5
python3.0	Programming language	C Python	No	18.5	585	150
gnat-gps	IDE	Ada Python C	Yes	5	450	17 [†]
axiom	Computer algebra system	Lisp Scratchpad C	Yes and No	30+	360	41 [†]
ruby-prof	Profiler	Ruby C	No	2.5	4	4

[†] Undercounts—**gnat-gps** and **axiom** have only 1 and 5 years, respectively, of public commit history

Slide 18

The result is this selection of five different case study packages. I kind of got lucky with the randomization stuff here because they're diverse along a lot of different dimensions. Specifically, there are different application areas, using a variety of languages, and different code sizes and team sizes. These are all open-source packages, but **gnat-gps** is industrially-developed by a company called AdaCore, so that's the 'yes' in the "Industrial?" column, and also **axiom** was originally developed by IBM and is now open-source and the current build system was written by open-source volunteers, so

that's the 'yes and no' for whether it's industrial.

It's important to have a variety of different kinds of packages so that I could make useful comparisons. If I only looked at, say, C compilers, then the sorts of conclusions I could draw from looking at those wouldn't necessarily be very generalizable and I might not be able to find anything interesting because they'd all be so similar. Here there are a variety of different packages that use a variety of different languages and differ in a variety of other ways. They're all independently-developed and that should allow me to make useful comparisons later on from the case studies.

Contributions

- Filename-based selection procedure
- Five deep case studies of open-source multilanguage packages
- Build patterns and anti-patterns
- Error-proneness finding
- (Anti-)pattern uses, implications
- Abstraction “leakage” finding

Slide 19

Now I'm going to discuss the case studies of these five packages.

Case Study Questions

Purpose and abstractions

Architecture, languages,
and interactions

Build system structure

Build issues

Rebuild issues

Build features

Slide 20

For each case study package, I addressed questions under these major headings.

- The first is purpose and abstractions: What is the package? What does it do? What sort of things does it present to the user?
- What languages is it written in? What parts are written in different languages? How do those different parts interact?
- How is the build system structured?

- What are the issues that occur when you try to initially build it?
- And also when you try to rebuild the package after making a small change?
- Finally, what are the build features that prevent build problems?

synopsis
python3.0
gnat-gps
axiom
ruby-prof

Slide 21

These were the five case studies: synopsis, python, gnat-gps, axiom, and ruby-prof. I only have time to go into one of them and I'm going to go with python because I think most people here are familiar with that even though in some ways it's not the best system to go into detail because it doesn't have build problems. All the other ones do have build problems but I'm going to talk about python so that you get a feel for what the case studies are like.

python3.0

Purpose and abstractions

Interactive high-level object-oriented programming language
with functions, methods,
classes, and modules

Slide 22

The purpose and abstractions of python—it's an interactive high-level object-oriented programming language. This question seems kind of straightforward, and for python it is pretty straightforward, because we all understand this, but for some of the other packages this question was important in trying to understand what exactly the package does and how it works.

Case Study Questions

Purpose and abstractions

Architecture, languages,
and interactions

Build system structure

Build issues

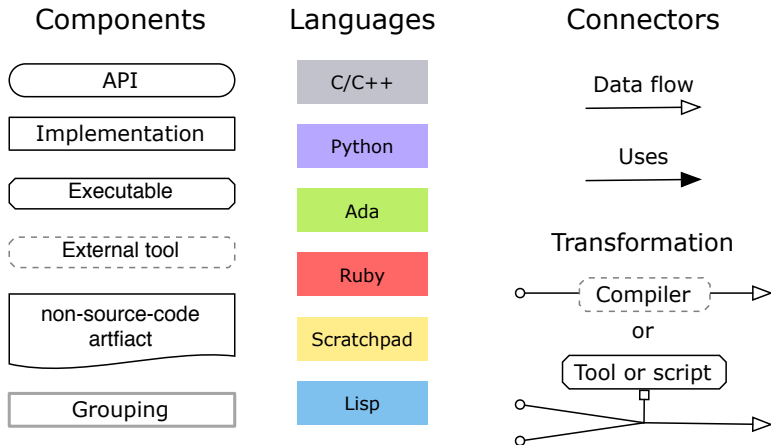
Rebuild issues

Build features

Slide 23

For the architecture, languages, interactions, and build system structure, I have textual descriptions, but I also use diagrams.

Multilanguage and build diagrams

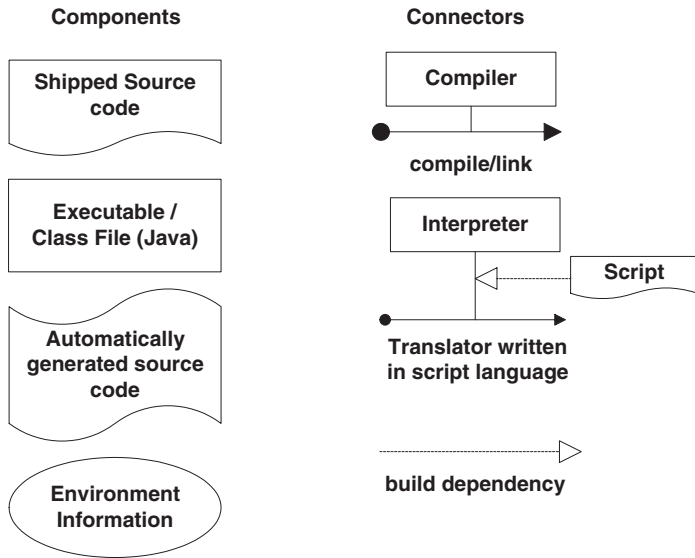


1 Build problem

Slide 24

This is the notation used for the diagrams to convey what the structure of the build system and the software itself is. There are components that are different shapes that show what different parts of the software we're looking at. The different implementation languages are all colour-coded and then there are different connectors that show the relationships between different components. For example, if data is flowing from one component to another, I use the white-headed arrow. If one component is calling into another or using it in some way, I use the dark-headed

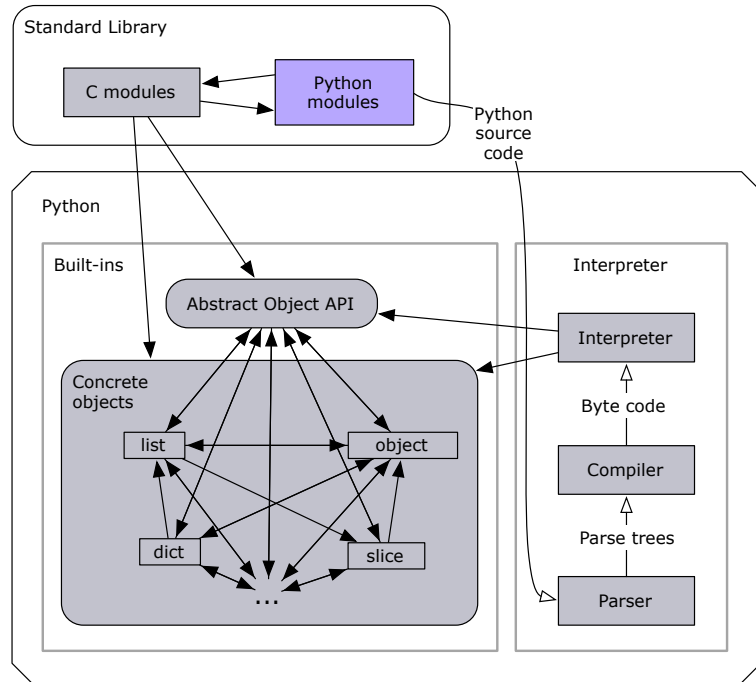
arrow. And then, for the build there will be a sort of dataflow arrow that has a white circular arrow tail. That shows that some artifact is being transformed into some other artifact via a tool. For some of the more complicated builds, something will be generated and then it's used to generate something else, and for that case I use a square arrowtail to show that artifacts are being transformed into other artifacts via a different artifact. An example would be that a configure script might generate a makefile that then drives the build process.



Slide 25

This notation is based on work by Tu and Godfrey called “The Build-Time Software Architecture View.” This is the notation they use. I changed it a bit. The other differences are that I’m using multilanguage stuff for colour coding, and that I’m doing it at a slightly more abstract level. I’m looking at higher-level architectural components instead of the individual artifacts shown in their diagrams.

Notation based on the paper by Tu and Godfrey,
“The Build-Time Software Architecture View”



Slide 26

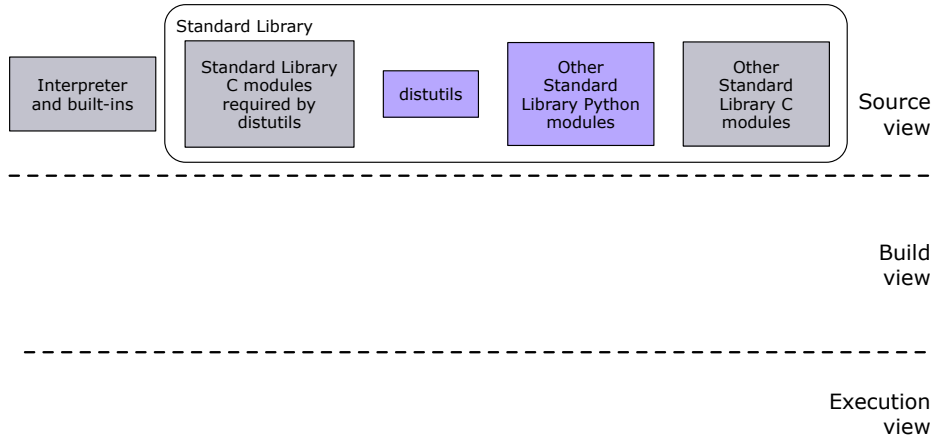
Here's one of these diagrams. This is the multilanguage architecture of python. The gray is C and the purple is python. Almost all of python is written in C and there are a bunch of python modules written in python.

The two major parts are that there is a standard library and then there's the python implementation itself. The python interpreter is written entirely in C and most of python's objects are implemented in C. So there's a file called object.c that implements what an object is. These are all tightly coupled together because they all use each other. For example an object's

namespace is a dict and a dict is also an object. All these entities are tightly coupled together, and C modules call into these objects via either the APIs directly or the abstract object API. Python code can't call this API directly. However it comes down as python source code through a parser, compiler, and interpreter, which ends up calling the exact same APIs as the C modules. The important point here is that modules written in C and modules written in python that talk to each other are calling completely equivalent APIs, it's just the C is calling the API directly, whereas the python is calling it indirectly through an interpreter.

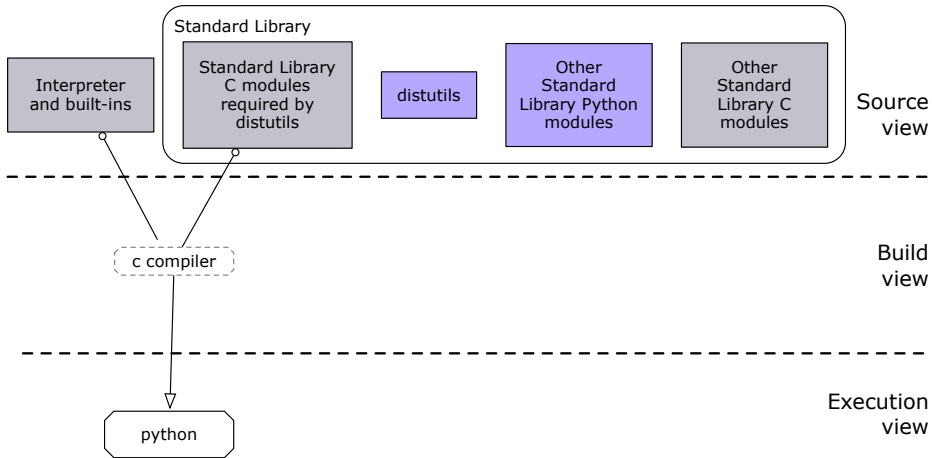
An example would be that to multiply two numbers using the C API, you call `PyNumber_Multiply`, whereas in python you write `a * b` to multiply two numbers. And eventually when you get into the interpreter, it just ends up calling `PyNumber_Multiply` too. So the code written in C and python have access to exactly the same API.

Slide 27a



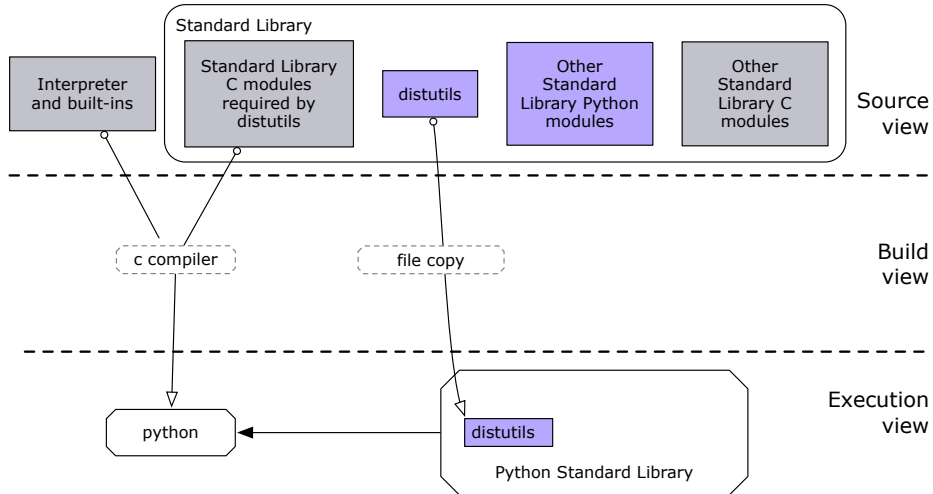
Now I'll show how this is actually built. This is a build-time view diagram. At the top level there's the source code. This is just what comes when you unpack the source code. The build view shows what happens at build time, and then the execution view shows which components exist at runtime and how they're related.

Slide 27b



To start, the interpreter is written in C so it just goes through the C compiler to turn into the python executable. Now the standard library is built by a standard library module called distutils. Well you kinda need the standard library to build itself—it's a complicated issue—so the way that it's resolved is that the C modules that are required by distutils are hardcoded into the build scripts. For example, distutils needs regular expression evaluation, which is a C module, so that is compiled directly into the python executable.

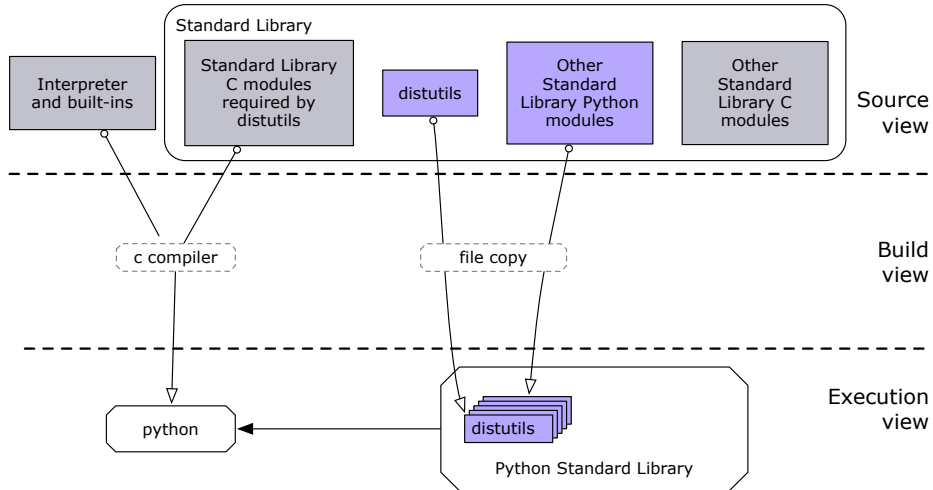
Slide 27c



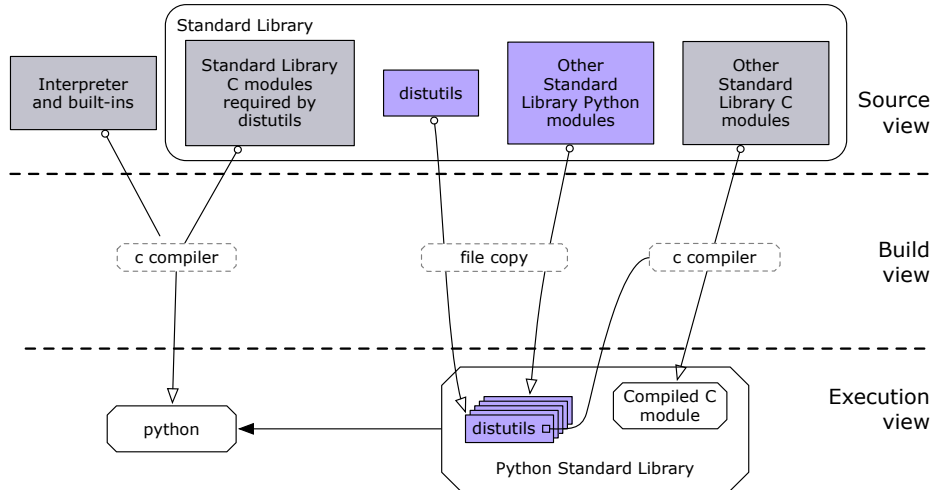
Once that's done, distutils is simply copied to the execution environment, because it's python code, and it uses the python interpreter directly to run.

Slide 27d

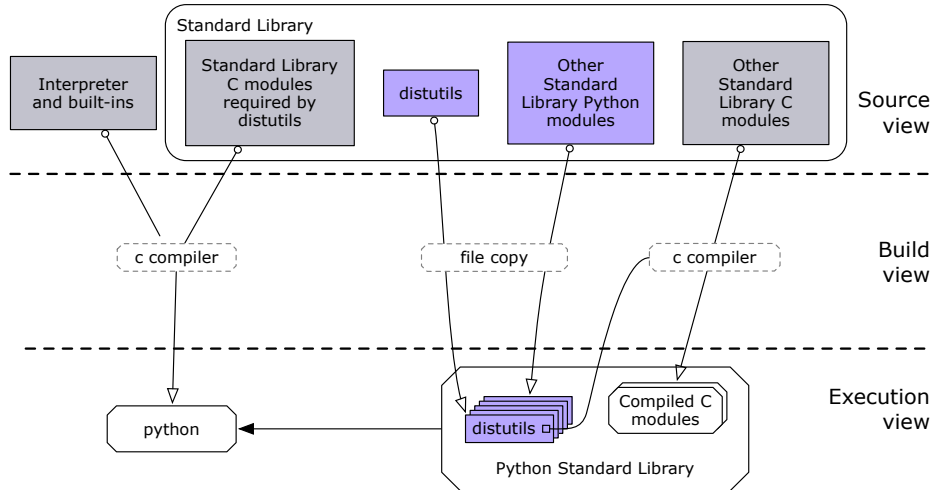
And the other standard library python modules are also all copied into the standard library.



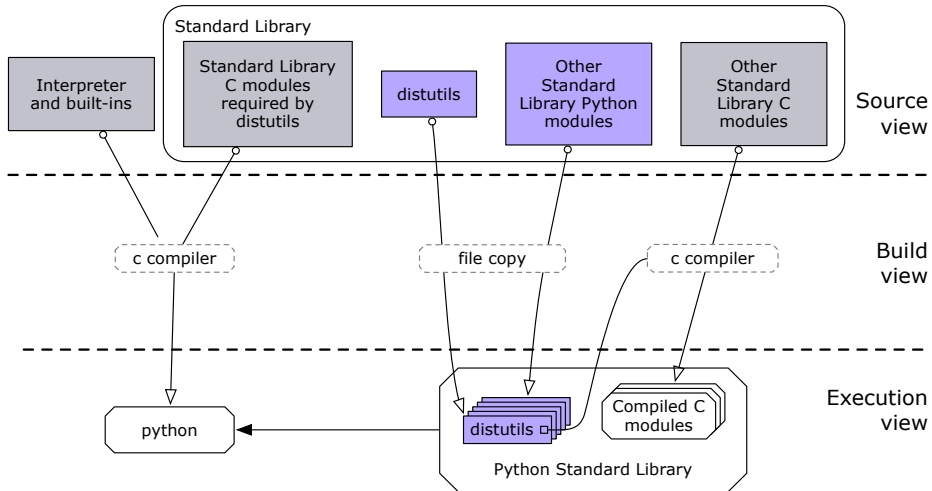
Slide 27e



Slide 27f

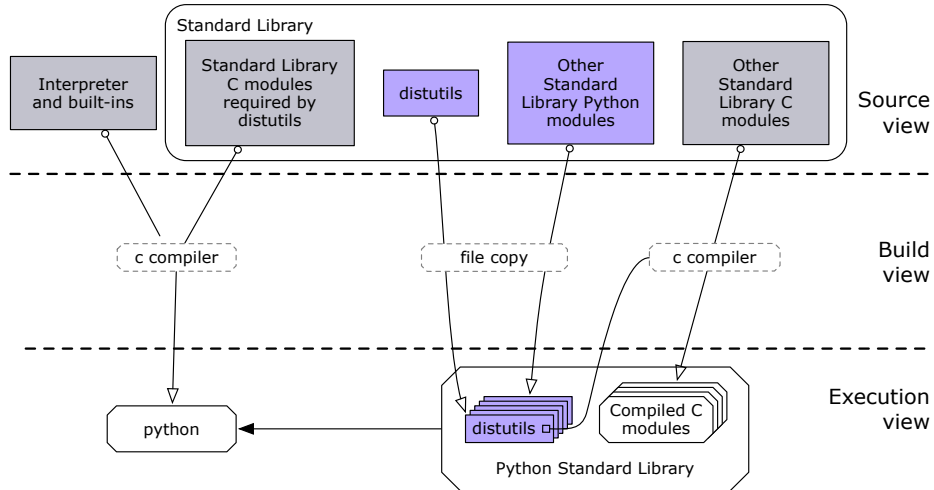


Slide 27g



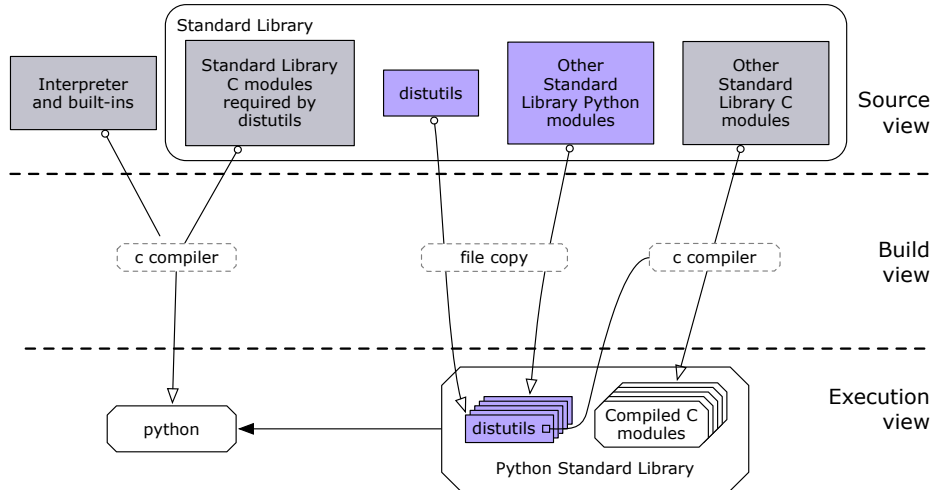
Once that occurs, distutils can now run because it has access to the C modules it depends on, and the rest of the python standard library is there. So it's able to build up the rest of the python standard library by invoking a C compiler to generate compiled C modules.

Slide 27h



Slide 27i

And the end result is the complete python standard library and the python executable at runtime.



Case Study Questions

Purpose and abstractions

Architecture, languages,
and interactions

Build system structure

Build issues

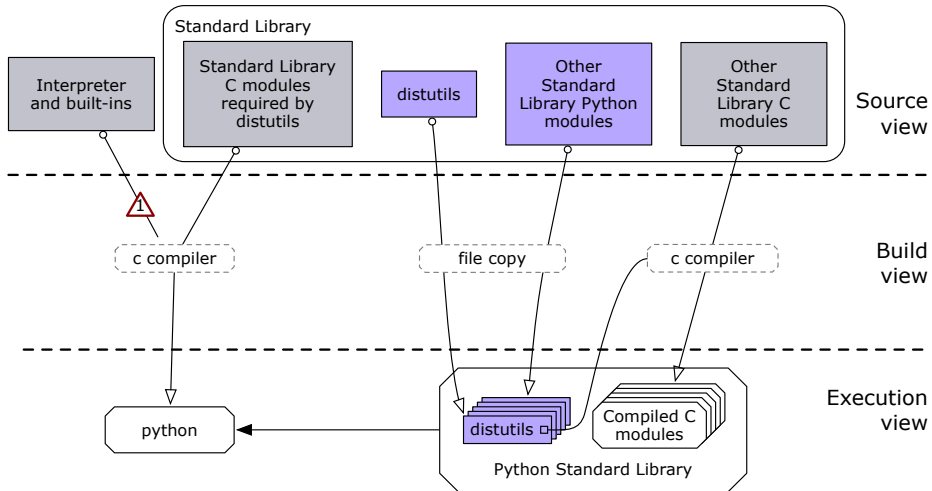
Rebuild issues

Build features

Slide 27j

The next question I addressed for each case study was what build issues and rebuild issues there were in trying to build the software.

Slide 27k



As I mentioned before, python isn't the greatest example for this, because there's only one issue. It's this red warning triangle right here that shows that when the interpreter is built the dependencies between the header files and the source files specified in the Makefile are manually specified as that every object file depends on every public python header file, but the list of all header files isn't quite right. The end result is that most of the time when you change any python header file and then try to rebuild, it will rebuild all of python. However there are some header files where

when you try to change them, those changes are just totally ignored. That's the one rebuild issue in python.

Case Study Questions

Purpose and abstractions

Architecture, languages,
and interactions

Build system structure

Build issues

Rebuild issues

Build features

Slide 28

The build features refer to the build problems of all the other case studies. It's not going to make much sense without context so I'm just going to skip that.

Purpose and abstractions

Architecture, languages,
and interactions

Build system structure

Build issues

Rebuild issues

Build features

synopsis

python3.0

gnat-gps

axiom

ruby-prof

Slide 29

































The contribution again was these case studies of five systems. I addressed all of these questions for each of them, including the diagrams of how they work, all the build problems I ran into, and the problems I encountered trying to rebuild these systems as well.

Contributions

- Filename-based selection procedure
- Five deep case studies of open-source multilanguage packages
- **Build patterns and anti-patterns**
- Error-proneness finding
- (Anti-)pattern uses, implications
- Abstraction “leakage” finding

Slide 30

Based on those build problems and features, I was able to produce a set of build patterns and anti-patterns.

	synopsis	python3.0	gnat-gps	axiom	ruby-prof
Anti-pattern: Filename Collision					
Anti-pattern: Installation Required					
Anti-pattern: Unverified Third-Party Software			 	 	
Anti-pattern: Incorrect Dependencies	 				 
Anti-pattern: Ignored Error					
Pattern: Build-Free Extensibility					
Pattern: Object-Oriented Builds					  
Pattern: Persistent Configuration			 		 

Slide 31

Here's a table of them. I'll let you look at that for a little.

The way that this table works is that this column has names of patterns or anti-patterns I produced from the research, and each of the other columns are individual case studies. A red triangle shows that there was a build problem for this system that contributed to the finding that this is a pattern or anti-pattern. And a green circle shows that there was a build feature that contributed to this pattern or anti-pattern.

Build (Anti-)Pattern Template

Description
Consequences
Evidence
Remedies
Applicability

Slide 32

What exactly is a pattern or anti-pattern?
I'm using the definition that it is a discussion of something about build systems that follows this template: that there's description, consequences, evidence, remedies, and applicability.

I'm going to give an example that shows what each of these headings mean right away. But first I'm going to talk about some of these patterns and anti-patterns.

	synopsis	python3.0	gnat-gps	axiom	ruby-prof
Anti-pattern: Filename Collision	②				
Anti-pattern: Installation Required	①	①	⑥		①
Anti-pattern: Unverified Third-Party Software	③		③ ④	① ②	
Anti-pattern: Incorrect Dependencies	① ②	①	②		③ ⑤
Anti-pattern: Ignored Error	④		③	③	
Pattern: Build-Free Extensibility		②	①	①	
Pattern: Object-Oriented Builds	③	④			① ① ⑤
Pattern: Persistent Configuration		③	② ⑦	⑤	② ④

Slide 33

The pattern Object-Oriented Builds was inferred by noticing that there were some interesting features used here and here but also some problems here and together they showed that this Object-Oriented Builds pattern is something that can address build problems and is a positive feature to have in a build system. That's the example pattern that I'm going to go over.

Example Pattern: Object-Oriented Builds

Description

Build system can be dynamically
customized and extended
using object-oriented APIs

Slide 34

The description of Object-Oriented Builds is that the build system can be dynamically customized using object-oriented APIs. An example would be the build tool called Rake that's implemented in Ruby. All the build entities are objects that you can manipulate at runtime instead of the build system being a bunch of shell commands inside a Makefile that aren't object-oriented at all where you can't do anything dynamically.

Example Pattern: Object-Oriented Builds

Consequences

Encapsulation and reuse
of build functionality

Object-oriented tools are less
mature and may be buggy

Slide 35

The consequences of the pattern Object-Oriented Builds are that you get encapsulation and reuse of build functionality. Build functionality can be packaged up in objects and can be reused in general-purpose ways. However, there's one slight downside that object-oriented tools are less mature and they may be buggy. So there aren't many bugs left in make because it's been around for so long but these brand-new tools have new features that haven't really been thought out or tested extensively sometimes.

Example Pattern: Object-Oriented Builds

Evidence

python3.0, synopsis use distutils
ruby-prof uses Rake
exports build-time profiling
some bugs

Slide 36

The evidence for this pattern is that python and synopsis use an object-oriented tool called distutils that works really well for helping it to build some fairly complicated software. And a package called ruby-prof uses an object-oriented build tool called Rake. It has a very small build system and has this really interesting feature—ruby-prof is a profiler for the Ruby language, and through the use of this build system, its ability to profile programs is exported as a build-time module. Other packages can import the build-time module to profile their own code at build time,

which is really interesting. However there are some bugs in Ruby libraries. It happens.

This is evidence of commonality among build systems in that these different packages are using object-oriented builds to good effect.

Example Pattern: Object-Oriented Builds

Remedies

Use tools like Rake, distutils, SCons

Package build functionality
for reuse in other projects

Slide 37

The remedies heading for a pattern describes what you could do to either implement this pattern or address an anti-pattern. The remedies for Object-Oriented Builds are to use object-oriented build tools like Rake or distutils or SCons, and to package build functionality for reuse in other projects.

Example Pattern: Object-Oriented Builds

Applicability

Single- and multilanguage builds

May help deal with complexity

Slide 38

And finally applicability asks, what exactly does this pattern or anti-pattern apply to? Does it just apply to the one system it occurred in, does it apply to build systems in general, or is it only for multilanguage? For Object-Oriented Builds, although I found evidence of object-oriented builds in multilanguage systems there's nothing really necessarily single-language about it. An object-oriented build could really be used for any sort of build system including single-language. However, it may help better deal with the complexity of a multilanguage build. Single-language builds

are, as I said before, kind of a solved problem, and you might not need this. But when you've got a more complicated build, this might help.

	synopsis	python3.0	gnat-gps	axiom	ruby-prof
Anti-pattern: Filename Collision	②				
Anti-pattern: Installation Required	①	①	⑥		①
Anti-pattern: Unverified Third-Party Software	③		③ ④	① ②	
Anti-pattern: Incorrect Dependencies	① ②	①	②		③ ⑤
Anti-pattern: Ignored Error	④		③	③	
Pattern: Build-Free Extensibility		②	①	①	
Pattern: Object-Oriented Builds	③	④			① ① ⑤
Pattern: Persistent Configuration		③	② ⑦	⑤	② ④

Slide 39

Here's the table of anti-patterns again. Now I'll talk about incorrect dependencies. Most of the previous work on build systems has all been about finding the right dependencies and pruning the software manufacture graph and having the right level of detail and doing smarter builds by only looking at what functions are called and stuff like that. But when I did these case studies I found that while the anti-pattern Incorrect Dependencies did show up a couple of times in some systems, and some other systems had good remedies for automatically finding the correct

dependencies, it wasn't actually that big a problem. It didn't stop anything from building. Remember, the main problem I had with these packages was getting them to build at all. And dependencies only matter for rebuilds. The dependencies don't matter much for getting the software to build in the first place. So although there's been a lot of work and most people would expect your build system problems are caused by incorrect dependencies, I found that it just wasn't actually that big a deal in terms of getting the software to build initially.

	synopsis	python3.0	gnat-gps	axiom	ruby-prof
Anti-pattern: Filename Collision	②				
Anti-pattern: Installation Required	①	①	⑥		①
Anti-pattern: Unverified Third-Party Software	③		③ ④	① ②	
Anti-pattern: Incorrect Dependencies	① ②	①	②		③ ⑤
Anti-pattern: Ignored Error	④		③	③	
Pattern: Build-Free Extensibility		②	①	①	
Pattern: Object-Oriented Builds	③	④			① ① ⑤
Pattern: Persistent Configuration		③	② ⑦	⑤	② ④

Slide 40

Now I'm going to talk about the anti-pattern of Filename Collision because I'm going to come back to it later. This was where one package failed to build because I was building it in Linux, in a virtual machine running on my mac, and it was using the Shared Folders feature of VMware. The Mac's filesystem is case-insensitive. The software running on Linux expected to access one file but it actually got a different file with a similar name after you ignored the case. And that caused the build to fail. Although that is the only pattern or anti-pattern where I only have evidence

from one case study, however, analyzing that specific anti-pattern shows it's not something specific to that package. It wasn't something that could only happen in synopsis; it could happen in any package that there are similarly-named files.

	synopsis	python3.0	gnat-gps	axiom	ruby-prof
Anti-pattern: Filename Collision	②				
Anti-pattern: Installation Required	①	①	⑥		①
Anti-pattern: Unverified Third-Party Software	③		③ ④	① ②	
Anti-pattern: Incorrect Dependencies	① ②	①	②		③ ⑤
Anti-pattern: Ignored Error	④		③	③	
Pattern: Build-Free Extensibility		②	①	①	
Pattern: Object-Oriented Builds	③	④			① ① ⑤
Pattern: Persistent Configuration		③	② ⑦	⑤	② ④

Slide 41

Another pattern to go over is Build-Free Extensibility, which was that these two packages, gnat-gps and axiom, were really hard to build. Axiom can't actually be built at all on the version of Ubuntu I looked at. And gnat-gps had such a horrible build system that the Ubuntu maintainer just rewrote it and he complained in the source code for the new system about all the "evil recursive Makefiles." These are systems that are very hard to build. However, Build-Free Extensibility is that, these packages have mechanisms for end-users to extend and customize the software that

make the build problems not that big a deal. Once you get an axiom binary from somewhere, say the project website, you can extend axiom all you want by using its built-in language called Scratchpad. You don't really ever need to rebuild the whole system from scratch unless you're an axiom developer. You can customize and you can extend and you can do all sorts of things without ever building, and the same holds for gnat-gps. And python has a similar feature, in that you extend python by writing python source code. We extend python all the time by writing useful python modules and posting them on github. And not once do we ever build python when

we're doing that. So even though there was that rebuild issue in python I mentioned, that's not really a problem for most people working with python, because there are these mechanisms to extend python that don't involve builds at all. This is a pattern that is an end-run around build problems.

Contributions

- Filename-based selection procedure
- Five deep case studies of open-source multilanguage packages
- Build patterns and anti-patterns
- **Error-proneness finding**
- (Anti-)pattern uses, implications
- Abstraction “leakage” finding

Slide 42

The next contribution that I’m going to look at is this finding that build systems for multilanguage software are error-prone.

Build Systems for Multilanguage Software are Error-Prone

4 of the 5 case studies
require manual intervention
to build successfully

(and python3.0, the 5th, has rebuild problems)

Slide 43

Four of the five case studies required manual intervention to build successfully. So if you download the source code, you unpack it, and you try to build it, you're gonna get a weird error message and you're gonna have to fiddle with it for a while before you get it to build. And that's not really what we'd expect—if build systems were a solved problem, these packages would just build. And they don't. Based on the fact that four of the five case studies are error-prone, I'm saying that build systems for multilanguage software are error-prone. Now, if this was a quantitative study and I only looked at five

and said, you know, well—that wouldn't work. However, since we're doing analytical inference and generalization here, we can look at the specific problems we encounter, and we can see whether they would only apply to one particular system or whether they could apply to many systems.

Build Systems for Multilanguage Software are Error-Prone

Commonalities among build
problems of independently-
developed packages

Not “one-off” problems

Slide 44

Since I was able to infer so many build patterns and anti-patterns from the problems I looked at, that's showing that there are commonalities among these problems. And, these are all independently-developed packages, which shows that there are systematic problems that could be systematically addressed. These aren't just one-off problems where each package is making its own individual mistake, there are commonalities among the problems.

	synopsis	python3.0	gnat-gps	axiom	ruby-prof
Anti-pattern: Filename Collision	②				
Anti-pattern: Installation Required	①	①	⑥		①
Anti-pattern: Unverified Third-Party Software	③		③ ④	① ②	
Anti-pattern: Incorrect Dependencies	① ②	①	②		③ ⑤
Anti-pattern: Ignored Error	④		③	③	
Pattern: Build-Free Extensibility		②	①	①	
Pattern: Object-Oriented Builds	③	④			① ① ⑤
Pattern: Persistent Configuration		③	② ⑦	⑤	② ④

Slide 45

That's shown by this chart where, apart from filename collision which I already addressed, each of these is taking evidence from multiple unrelated case studies in order to infer the pattern or anti-pattern.

Research question 1

Slide 46

Q) What are the major issues in building multilanguage software?

A) Getting the software to build at all is the major issue

This addresses research question number one, which is, what are the major issues in building multilanguage software? And, based on the case studies I've conducted, the answer is: Getting the software to build at all is the major issue.

Contributions

- Filename-based selection procedure
- Five deep case studies of open-source multilanguage packages
- Build patterns and anti-patterns
- Error-proneness finding
- (Anti-)pattern uses, implications
- Abstraction “leakage” finding

Slide 47

The next contribution I’m going to address is about uses and implications of patterns and anti-patterns.

Uses and implications

(Anti-)patterns not necessarily
multilanguage-specific

(Anti-)patterns could be
addressed by build frameworks

Key (anti-)patterns

Slide 48

So there are three things I want to talk about in terms of uses and implications of patterns and anti-patterns. The first is that they're not necessarily multi-language specific. The next is that they could best be addressed by build frameworks. And the third is that there are two key patterns that seem like they could be most useful for addressing the systematic problems in building multilanguage software.

Uses and implications

(Anti-)patterns not necessarily
multilanguage-specific
(but may be more likely—e.g.,
Incorrect Dependencies)

(Anti-)patterns could be
addressed by build frameworks

Key (anti-)patterns

Slide 49

The first thing is that none of these patterns or anti-patterns are necessarily multilanguage-specific. When you look at what the pattern actually is, although it was found specifically in multilanguage systems, there's nothing specifically multilanguage about it. So, these could all apply to single-language software as well, which is kind of interesting. However, I think that they are more likely in multilanguage software. For example, with incorrect dependencies—it's kind of a solved problem in the single-language case, there's been lots of research, and there are tools

that just automatically do everything. But when you start mixing the different languages together, you end up with all sorts of things. The problems that happen in single-language packages, happen in multilanguage software, and I think it's more likely to happen there.

Uses and implications

(Anti-)patterns not necessarily
multilanguage-specific

(Anti-)patterns could be
addressed by build frameworks

Key (anti-)patterns

Slide 50

The next thing I want to talk to you about is that patterns and anti-patterns could best be addressed by build frameworks.

Build Frameworks

(Anti-)patterns could be
used by practitioners

But generally better for build
tools and build frameworks

Slide 51

A developer could take this list of patterns and anti-patterns, and go through their source code, and say, “Oh, oh! We’ve got that anti-pattern, I’m going to fix it. Oh! That would be a nice pattern to have, let me add it.” But it would be a lot of work. It would probably be better if this was done in some sort of general-purpose way in terms of build tools and build frameworks.

Why build frameworks?

Patterns take effort to implement

Anti-patterns take effort to correct

For individual projects:

Tangential

Technical debt

Slide 52

These patterns take effort to implement, and the anti-patterns take effort to correct, and for individual projects, it's sort of tangential. You could be adding features or fixing bugs in this time that you're adding this nice new feature to the build system. That isn't addressing a problem that actually affects your users. Many projects will be quite comfortable with the technical debt of having anti-patterns in their build systems, or not having some nice patterns.

Slide 53

Build frameworks allow this problem to be addressed systematically.

Why build frameworks?

Address problems systematically

Why build frameworks?

Address problems systematically

Example: Filename Collision

General-purpose solutions could
be viewed as less tangential

Slide 54

An example. I talked about filename collision earlier. It's this problem where when you build this software on a certain filesystem, it won't build because of the way certain files are named. There's a case-insensitive name clash. The issue I looked at was very specific to python—there was an issue that involved the python byte-compiled cache files, so it was kind of complicated. But you could write a tool that would automatically detect this, and your tool could give a warning when you build this. It could say, “Hey, this won't build on a Mac, are you sure you want to do

that?” And, creating a general-purpose solution that could be used by many different projects, could be viewed as less tangential by individual projects. So, if I report this bug to the synopsis developers—“this won’t build on a mac”—they could fix it, but it would be a lot of work for them, and it would only fix it for synopsis. Whereas, if they were to do this in a general tool, a lot of people could use it.

Object-Oriented Builds

Could address (anti-)patterns
systematically

e.g., Filename Collision plug-in

General-purpose solutions could
be viewed as less tangential

Slide 55

In terms of Object-Oriented Builds, that could be a plugin. So, since synopsis is done in an object-oriented build tool, they could implement such a check as a plugin for that build tool, and then it could just be used by all sorts of people. It would be less tangential, because instead of just fixing one obscure problem for one system, you're fixing one slightly less-obscure problem for a large number of systems.

Uses and implications

(Anti-)patterns not necessarily
multilanguage-specific

(Anti-)patterns could be
addressed by build frameworks

Key (anti-)patterns

Slide 56

Object-Oriented Builds is one of two key
patterns that I want to talk about.

And the other one is Build-Free
Extensibility.

Key (Anti-)Patterns

Object-Oriented Builds

Build-Free Extensibility

Build-Free Extensibility

Provide extension mechanisms
that do not require building,
e.g., a scripting interface

End-run around build problems

Someone still needs to build

Slide 58

As I mentioned before, that's about providing extension mechanisms that don't require building. For example, a scripting interface. Even if the software's really hard to build, you have some sort of interface that still lets you do a lot of the things that a build system would let you do.

It's an end-run around build system problems. Someone still needs to build, someone still needs to address these build problems, but not everyone who uses the software has to.

Build-Free Extensibility

Scripting components and build systems have the same goal:
turn source code into
running programs

No longer tangential

Slide 59

One interesting thing related to Object-Oriented Builds—when you're addressing build problems in an object-oriented way, releasing them as general-purpose solutions, it's less tangential—scripting components have the exact same goal that build systems do, which is, they turn source code into running programs. So, whether that's a Makefile that is supposed to turn your C source code into a running program, or whether that's a component in your system that's supposed to load python code and run the scripts on your documents, it's got the same goal, so

it's no longer tangential. Getting that code built and running properly becomes one of the goals of the software project.

Contributions

- Filename-based selection procedure
- Five deep case studies of open-source multilanguage packages
- Build patterns and anti-patterns
- Error-proneness finding
- (Anti-)pattern uses, implications
- Abstraction “leakage” finding

Slide 60

And, that's mostly related to this finding I had ...

Leaking Abstractions

Abstractions and mental models from the application and implementation domains are manifested in the build system with positive or negative effects

that abstractions from application and implementation domains are leaking into the build system. I have two ways to explain it. The first one is that abstractions and mental models from the application and implementation domain are manifested in the build system with positive or negative effects. I have some detailed examples right away if this doesn't make sense. When I performed these case studies, I was noticing things in the build system that looked like they came from the way the software was implemented or what the software was intended for. There were properties—

Leaking Abstractions

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

There's another way I have for explaining this. The build system has properties whose presence may only be explainable by reference to the application and implementation domains. So, if there's something in the build system, and I'm looking at it, saying, "I've seen a lot of build systems before, but I can't understand how anyone would do that. Like, why would you do that in a build system?" And after thinking about it for a while, and looking at what the package actually did, it started to make sense.

Slide 63

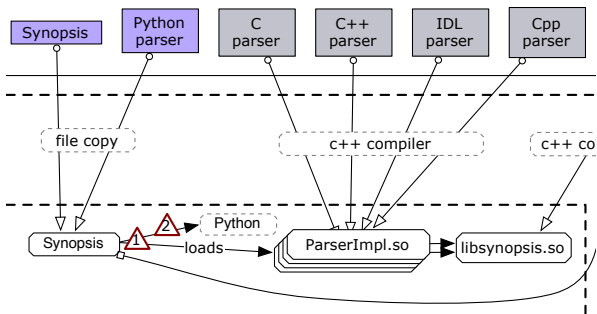
Let me give some detailed examples.

Detailed Examples

synopsis

independently-developed and -built
components with standard interfaces

1 only major problem:
search path configuration



CORBA

Slide 64

Synopsis is a source code documentation tool. The way it's built is there are parsers for different languages, and they're all taken from other open-source projects. Each of these—this is OmniIDL, this is openCpp—they took those and they wrapped a standardized interface around them. And each of these is independently-built to create a shared library and a python module. Synopsis loads each of these modules. It's using independently-developed and -built components with standardized interfaces, and the only problem, the only major

problem I encountered trying to build this was that it's kind of tricky to set up the paths so that synopsis can find all these different shared libraries and load them properly. Now, synopsis was developed as a documentation tool for an experimental CORBA-based UNIX windowing system. That's the CORBA there. For those not familiar with CORBA, CORBA uses independently-developed and -built components with standardized interfaces. And one of the problems you get trying to develop CORBA software is you gotta somehow configure all these components to find each other, and it gets kind of complicated. I thought it was really

interesting that the build system was designed in the same way as the application this tool was developed for, and that it had very similar problems, namely search path configuration.

python3.0

simple, flat, explicit build system

① incorrect non-‘magic’ dependencies

```
>>> 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.
```

Slide 65

The next case study was python. And if type ‘import this’ into python, you get a little poem about python’s philosophy. It says things like, “Implicit is better than implicit,” “Simple is better than complex.” These are sort of design decisions when you write in python. And even though the python build system for building the interpreter, the part that’s written in Make, is Make and shell scripts, it has many of the same features of idiomatic python. They pick explicit things, they pick simple things. One choice you have to make when designing a Make-based build system is, do

you want it flat, or do you want it hierarchical? And they went with flat. And part of the python philosophy is, “Flat is better than nested.” They’re using the same philosophy they use in developing python for developing the build system itself.

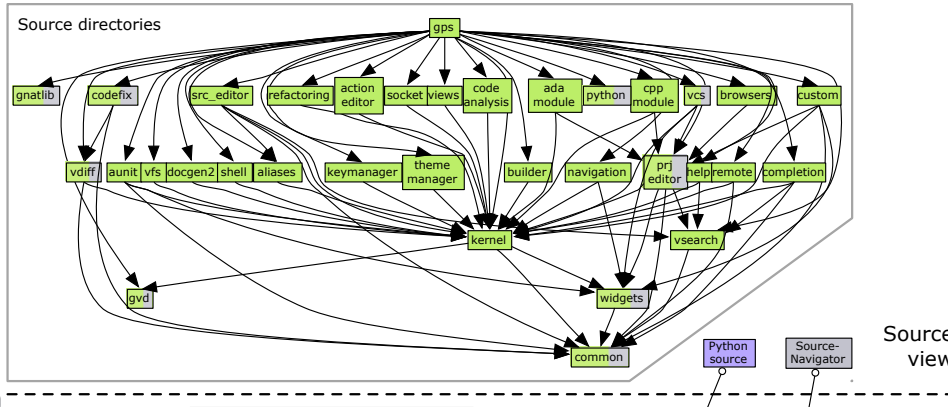
The only problem I encountered with python, as I said, was the rebuild problem, that some of these dependencies are manually-specified and they get them wrong. And, that, python in general tries to avoid ‘magic’ stuff like just automatically finding things. It’s right here—“Explicit’s better than implicit.” And in the python philosophy, you’re better off just explicitly stating these things instead of letting some

tool try to figure it out automatically, possibly getting it wrong.

Python the application and python's build system, share many of the same nice properties, but also the occasional bad property, such as avoidance of magic leading to extra work and bugs.

gnat-gps

hierarchical structure difficult to build
but allows splitting off libraries



Slide 66

gnat-gps has this really complicated hierarchical build structure. It's got this graph. Each of these is built independently and it calls into the build systems for everything where the arrow's going in. It gets messy. And this is the one where the Ubuntu maintainer rewrote this and they got rid of the hierarchical structure entirely, and cleaned out the evil recursive makefiles, and now it just builds all the code all at once, into one file, and it works. And it's great. And I could never get the original to build.

This seems like kind of a strange way to

make a build system. Why do it like this when you could have a nice simple flat one? And it turns out that this matches the business goals of the company that develops this software. This package is a development environment for Ada source code, but the company that makes it also uses it as a development environment for libraries. For example, they have an XmlAda library for XML processing in Ada. That started out as part of gnat-gps to deal with XML stuff, and once it got mature, they split it off into a separate library. It's on their website, you can download it. They like developing stuff as part of their IDE, and releasing it as separate open-source packages to make Ada

more attractive for development. When there are a lot of mature well-tested libraries for Ada, people are more likely to use it, and the company that makes this is more likely to sell commercial Ada compilers and commercial Ada support.

The reason for this hierarchical structure is that any particular library has to explicitly list all the other packages that it depends on. So that, when it's time to, say, split off this widgets module, into its own open-source library, it's not going to be calling into any of the other parts of the IDE. It won't get tangled up in the other parts of it and make it impossible to pull out as its own package. This structure,

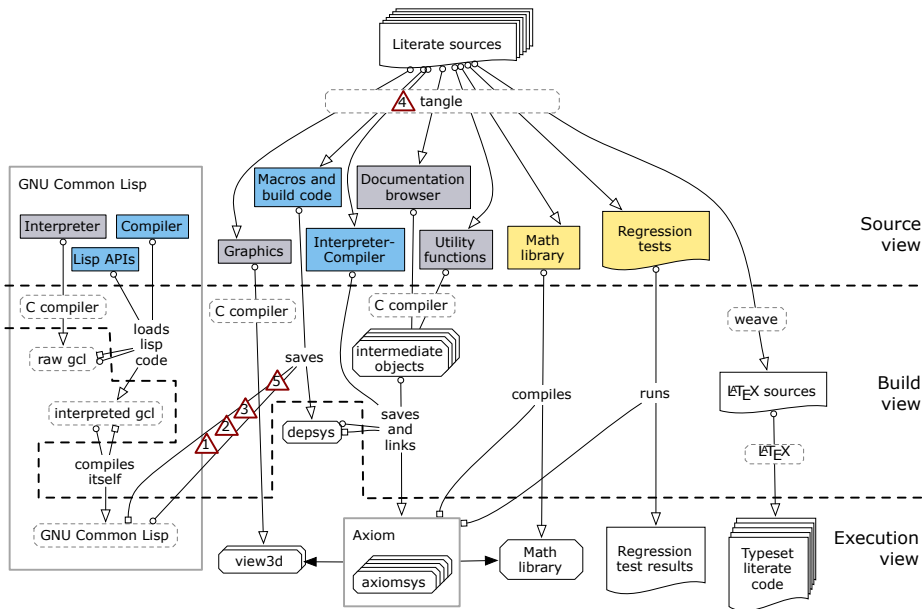
which makes it really hard to build, supports the business goals of the company that develops the software. From a purely technical standpoint, this isn't a very good way to design the build system, but it totally makes sense when you consider the application domain.

axiom

Slide 67

I found this in all the case studies and I'm going through them in order.

For the axiom case, it has a really complicated build, and it just will not and does not build on the Ubuntu version that I was working with.



Building axiom

Develop ideas in ScratchPad

Mark up code in \LaTeX

Rebuild from scratch

Run extensive regression tests

Slide 68

Here's the basic outline of how you work with axiom. It's got its own language called Scratchpad, that you can use for interactive development, and you can develop ideas, and you can write new mathematical code in it.

Building axiom

Develop ideas in ScratchPad

Mark up code in \LaTeX

Rebuild from scratch and run
extensive regression tests

Slide 69

Here's an example of the ScratchPad source code. It's not bad at all. It's nice and mathematical. And you can develop all sorts of algorithms, and you can add new types of number systems and stuff like that.

```
1  )abbrev package ADDTWO AddTwoNumbers
2  AddTwoNumbers(A) : Exports == Implementation where
3  A : IntegerNumberSystem
4  Exports == with
5      addTwo: (A, A) -> A
6  Implementation == add
7      addTwo(a, b) == a + b
```

Building axiom

Develop ideas in ScratchPad

Mark up code in \LaTeX

Rebuild from scratch

Run extensive regression tests

```
\defun(setCurrentLine){setCurrentLine}
Remember the current line. The cases are:
\begin{itemize}
\item If there is no  $\$currentLine$  set it to the input
\item Is the current line a string and the input a string?
    Make them into a list
\item Is  $\$currentLine$  not a cons cell? Make it one.
\item Is the input a string? Cons it on the end of the list.
\item Otherwise stick it on the end of the list
\end{itemize}
Note I suspect the last two cases do not occur in practice since
they result in a dotted pair if the input is not a cons. However,
this is what the current code does so I won't change it.
\usesdollar(setCurrentLine){currentLine}
<<defun setCurrentLine>>=
(defun |setCurrentLine| (s)
  (declare (special |$currentLine|))
  (cond
    ((null |$currentLine|) (setq |$currentLine| s))
    ((and (stringp |$currentLine|) (stringp s))
     (setq |$currentLine| (list |$currentLine| s)))
    ((not (cons p |$currentLine|)) (setq |$currentLine| (cons |$currentLine| s)))
    ((stringp s) (rplacd (last |$currentLine|) (cons s nil))))
    (t (rplacd (last |$currentLine|) s)))
  |$currentLine|)
```

Figure 4.20: noweb input example from axiom source code

5.3.23 defun setCurrentLine

Remember the current line. The cases are:

- If there is no $\$currentLine$ set it to the input
- Is the current line a string and the input a string? Make them into a list
- Is $\$currentLine$ not a cons cell? Make it one.
- Is the input a string? Cons it on the end of the list.
- Otherwise stick it on the end of the list

Note I suspect the last two cases do not occur in practice since they result in a dotted pair if the input is not a cons. However, this is what the current code does so I won't change it. [$\$currentLine$ p??]

```
(defun setCurrentLine)≡
  (defun |setCurrentLine| (s)
    (declare (special |$currentLine|))
    (cond
      ((null |$currentLine|) (setq |$currentLine| s))
      ((and (stringp |$currentLine|) (stringp s))
       (setq |$currentLine| (list |$currentLine| s)))
      ((not (cons p |$currentLine|)) (setq |$currentLine| (cons |$currentLine| s)))
      ((stringp s) (rplacd (last |$currentLine|) (cons s nil))))
      (t (rplacd (last |$currentLine|) s)))
    |$currentLine|)
```

Slide 70

Now, if you want to add this into axiom and have it become part of axiom itself, you gotta mark up the code in LaTeX. It's got this really complicated literate source code system, this didn't render properly, but it's probably better that you can't see it. And there's all this extra stuff added in. So if I read this, it says, "Remember the current line. The cases are ..." these, and then at then at the end, "Now I suspect that the last two cases do not occur in practice, however this is what the current code does so I won't change it." It's voluminous, and it's all done in this macro thing and the

input turns into that and it gets tangled,
and then, when you're building
axiom—there's no support for incremental
builds at all.

Building axiom

Develop ideas in ScratchPad

Mark up code in \LaTeX

Rebuild from scratch and run
extensive regression tests

Slide 71

You change one line in a seven-megabyte LaTeX file and then you've got to rebuild the whole thing from scratch and run all these regression tests. And I'm looking at this, and I'm trying to think, like, "Why would someone do it like this?" Like, what, how, what, how could someone create a build system that doesn't even support incremental rebuilds?

Building axiom

Develop ideas in ScratchPad

Mark up code in \LaTeX

Rebuild from scratch and run
extensive regression tests

One possible parallel for this
unusual build system is another
process that mirrors it:

Slide 72

And where's all this LaTeX stuff coming in from? Like, I understand that it's useful for math, and—this might be a kind of a tenuous connection, but the only thing that I could think of that kind of parallels that process is . . .

Building axiom

Develop ideas in ScratchPad

Mark up code in \LaTeX

Rebuild from scratch and run
extensive regression tests

Doing math

Develop ideas on scratch paper

Mark up results in \LaTeX

Submit for extensive peer review
and publication

Slide 73

Doing math itself. When you're a professional mathematician, you develop your ideas on scratch paper, you work out your proofs, you work out your algorithms, and when you think you have something, you mark it up in LaTeX, and then you submit it for extensive peer review and publication. That could take years sometimes. The only way I could think of, the only way I could conceive of someone coming up with a build system like that is if they were sort of using this system. This is what I talk about—the abstractions of doing professional mathematics show up in

the build system for this mathematical software.

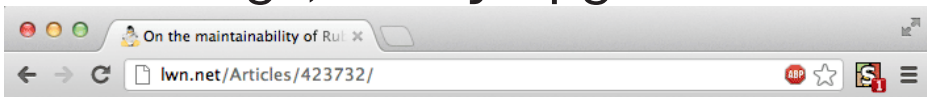
ruby-prof

cool new features

bugs, library upgrades

Slide 74

The last case study was the Ruby profiler. It has a very small build system that delegates almost everything to Ruby libraries. Which is the way Ruby does things. And the Ruby build system has some really cool features, like being able to export build time functionality to other packages. It does have some bugs though. But the bugs that I encountered in the version I looked at are now fixed, because Ruby believes in having lots of cool new features, and that it's ok if they're buggy, as long as there are lots of frequent library upgrades that fix those bugs.



Not logged in
[Log in now](#)
[Create an account](#)
[Subscribe to LWN](#)

Weekly Edition
[Return to the](#)
[Development page](#)

On the maintainability of Ruby

Lucas Nussbaum, longtime maintainer of Debian's [Ruby](#) packages, announced that he was stepping back from the role on January 2, making the future of Ruby packaging on the distribution uncertain. He [cited](#) a list of reasons leading to the decision on his blog — an assortment of project management, release process, and packaging concerns with Ruby itself that compounded his frustration to the point where he felt quitting was the best option. The public reaction to the announcement has been sympathetic to the hardships of Debian package maintainer-ship, but it has also sparked an interesting debate about the competing needs of developers, end users, and Linux distributors.

January 19, 2011

This article was contributed by
Nathan Willis

Here's an article in LWN where the people at Debian were complaining about Ruby packages in general. Debian likes making a stable release that people can use for years. And Ruby believes in releasing something cool every week, that has a few bugs, that get fixed next week. And that's really at odds with Debian trying to have a really stable version and Debian's just saying this is unsustainable. But the Ruby culture of doing this matches up exactly with the build system.

Leaking Abstractions and Build Ownership Styles

An Empirical Study of Build Maintenance Effort

Shane McIntosh, Bram Adams, Thanh H. D. Nguyen,

Yasutaka Kamei, and Ahmed E. Hassan

Software Analysis and Intelligence Lab (SAIL)

School of Computing, Queen's University, Canada

{mcintosh, bram, thanhnguyen, kamei, ahmed}@cs.queensu.ca

ABSTRACT

The build system of a software project is responsible for transforming source code and other development artifacts into executable programs and deliverables. Similar to source

1. INTRODUCTION

The build system of a software project is the infrastructure that translates source code, libraries, and data files into a set of deliverables (e.g., executables and documentation).

Slide 75

To show that I'm not off the wall on this abstractions stuff, I'm going to refer you to some related work called "An Empirical Study of Build Maintenance Effort" by Shane McIntosh and others. And this was an empirical study of a number of open-source projects. By looking at how often source code and build code were changed together or separately in the version control systems of about twenty different projects, they hypothesized that there seem to be two different kinds of 'build ownership styles.' In one, a project will have the build system be a shared

responsibility for everyone. Every time they make some changes, they're supposed to change the build system to keep up with those changes. Whereas the other style is, you have a team of people that have centralized build ownership, and they're responsible for the build systems. You make small changes to the build system if you add one file or something, but for more major changes it gets passed off to some people who are specifically responsible just for the build system. And they hypothesized that the centralized build ownership style reduces the overall maintenance effort for build systems. What I'm claiming for the leaking abstractions is that this agrees with

that. People who are dealing only with the build systems, and not also with the application or implementation domains, are less likely to include things from the application domains, the implementation domains, into the build system, that can have positive or negative effects but are usually negative. In the case studies, for the most part, having these abstractions did not help in the build system. However for python it definitely did. Because python's nice and clean, and it works well, and the build system was the same way.

Contributions

- Filename-based selection procedure
- Five deep case studies of open-source multilanguage packages
- Build patterns and anti-patterns
- Error-proneness finding
- (Anti-)pattern uses, implications
- Abstraction “leakage” finding

Slide 76

These are the contributions I've covered now. There's the filename-based selection procedure, the five deep case studies, the build patterns and anti-patterns produced from analyzing those case studies, the finding of error-proneness, and the uses and implications of (anti-)patterns, and this finding about abstraction “leakage.”

Research question 1

Q) What are the major issues in building multilanguage software?

Slide 77

To revisit the research questions:

What are the major issues in building multilanguage software?

Research question 1

Slide 78

Q) What are the major issues in building multilanguage software?

Getting it to build at all is the major issue.

A) Getting the software to build

Research question 2

Q) How can build problems be addressed?

Slide 79

How can build problems be addressed?

Research question 2

Q) How can build problems be addressed?

A) (Anti-)patterns, particularly when integrated into build tools and build frameworks

Slide 80

Potentially through patterns and anti-patterns, particularly when they're integrated into build tools and frameworks.

Research question 3

Slide 81

Q) Why do they occur?

Why do patterns and anti-patterns occur?
Or, why do build problems occur,
specifically?

Research question 3

Q) Why do they occur?

A) Tangential
Leaking abstractions

Potentially addressed by
object-orientation, build-free
extensibility

Slide 82

Working on the build system is kind of tangential to the software, so it doesn't get that much attention, and abstractions from the implementation and application domains leak in and sort of confuse the build system. These problems are potentially addressed by using object orientation, to make it less tangential, and Build-Free Extensibility, which would also make it less tangential.

Questions?

Slide 83

So, this is the end of my— why does it?
Oh, I'm sorry— It's really the last slide. It says 83 of 84 but I must have started at zero or something. Ok. [laughter]

Thanks. Thank you all for listening and I will now take questions.

[Long pause]

You can ask about the weather or something.

Abram: For filename collision, did you build all the software on the case-sensitive partition?

Andrew: No, after the first one, I just— I don't want to have to deal with this so I switched to Linux.

Abram: Do you think it would help you if you applied, if you tried to rebuild the other ones on that partition?

Andrew: Would it help ...?

Abram: Well, because you have this grid, right? And you say it only exists in one product, but really you only tested one product.

Andrew: Yeah that's true. I could potentially do that.

Abram: Ok.

...

Andrew: I know some people are saving their questions.

Ken: Yeah, the examiners have to, otherwise they'll run out. [Chuckles]

Andrew: Ok, well ...

Eha: Still there are two minutes to ten o'clock, so, ...

Ken: Or if you're afraid of having your question asked, you can ask it now.

Ehab: Ok. So. I guess if there are no

questions then we can ask the audience, not the examining committee, to leave the room, and we'll start the examination.

Andrew: Ok. Do I go away now, or—

Ehab: Just a second. You wait for a while.

Abram: Yeah we should thank the speaker.

Ehab: Oh. Let's thank the speaker.
[applause]