Problems in Software Composition

Stephen Kell

Stephen.Kell@cl.cam.ac.uk

About this talk

This talk is

- an introduction to some problem areas
- about questions, not answers
- supposed to be interesting to all
- supposed to generate feedback
- not quite finished

Developing software is hard

Developing a big project from scratch is hard. But so are:

- extending existing software, without source
- extending existing software, with source
- re-using existing software (in a new project)
- re-using existing software (together with another existing piece)
- extracting a piece from a bigger piece
- removing a piece from a bigger piece
- replacing an existing piece with another piece

Why? Many reasons, mostly rooted in *unanticipated* nature.

Some example scenarios

Consider Firefox. It

- consists of many functionally-separable components
- has several dependencies (kernel, libc, helper apps...)
- contains code useful in realising novel functionalities

Suppose we want to do the following with Firefox:

- **extend** to support storing bookmarks in a database
- **compose** with libhttpd to make peer-to-peer web client
- **replace** local history log with calls into system-wide log
- **extract** tabbing widget logic

What makes them hard? Easier than they might be?

Summary of problems

- no suitable extension interface
- (part-) reimplementation unavoidable
- constrained choice of language
- complex APIs are hard to understand
- (semantically) mismatched interfaces
- need to write much ad-hoc non-reusable (glue) code
- tendency for *tight coupling* of new code to existing
  - artifacts of *packaging* permeate source
  - two “interfaces”: ideal A, and actual C
    - $C \rightarrow A$ adaptation is a source of coupling

Example 1: `imap-status`

My first Python code: a script which outputs the status of each folder in an IMAP folder tree. Some quick and dirty measurements:

- 100 lines total, of which
  - 15 lines core functionality
  - 5 lines language boilerplate
  - 5 lines documentation (“usage” printout)
  - 35 lines input retrieval and unpackaging
  - 20 lines adaptation and workarounds
  - 10 lines output packaging
  - 10 lines blank

Anatomy of a solution

We need at least two things:

- a way of specifying components such that they can easily be composed
  - a problem of languages and tools
- a way of composing such components
  - in the static case, a tools problem
  - in the dynamic case, a systems problem

The following would also be nice:

- a refactoring approach, to improve composability of legacy components
- useful software measurements, for evaluation

Why involve the operating system? (1)

What is an operating system anyway?

- secure multiplexing of hardware
- *support for* abstraction
  - avoid compulsory abstractions in OS (Exokernel)
  - also avoid them in applications!
- diagram goes here

Why involve the operating system? (2)

What characterises the things that *should* go in the operating system?

- elevated privilege
- generality: no undue compromise to any application

A service for the *composition* of software could satisfy these. This means a linker capable of (programmed) semantic adaptation. Other arguments:

- precedent, inclusivity (“end-to-end razor”)
- efficiency
- OS manages many application concerns (communication, storage, security)
  - want customisation, guarantees and re-use

Some relevant existing research

Some relevant areas:

- module systems
  - OO-family: classes, features, aspects, mixins; higher-order....
- rich interface specifications
  - contracts; protocol; QoS; concurrent behaviour; ...
- automatic adaptation
  - language-to-language (often generative e.g. SWIG)
  - syntactic adaptation: argument name / position
  - fully semantic
- separation of compositional concerns
  - linkage / composition languages; flexible packaging; abstract imports

Abstract imports

Targeting complex concrete APIs causes coupling and makes development harder. So why not:

- declare the API you'd *like* to have
- write your code as if it existed
- specify adaptations in a domain-specific language
- apply this to *all* code on which your module depends

Notable benefits (aside from the obvious):

- DSL interpreter can check correctness of adaptation
  - up to precision of available interface specs
- adoptable: can use familiar languages and paradigms
  - only requires separate compilation

Uncertainties and next steps

As-yet-unanswered drawbacks:

- adds lots of complexity to linker
  - becomes interpreter of DSL
- writing adaptation code might still be hard
- erodes programmer discipline?

Next steps:

- identify some more case studies
- proof-of-concept hacking on GNU `ld`
- work out whether/how to do semi-automatic refactoring
- see if anyone knows how to extract rich interface specs