launchpad

Foundations

Gary Poster
gary.poster@canonical.com

CANONICAL

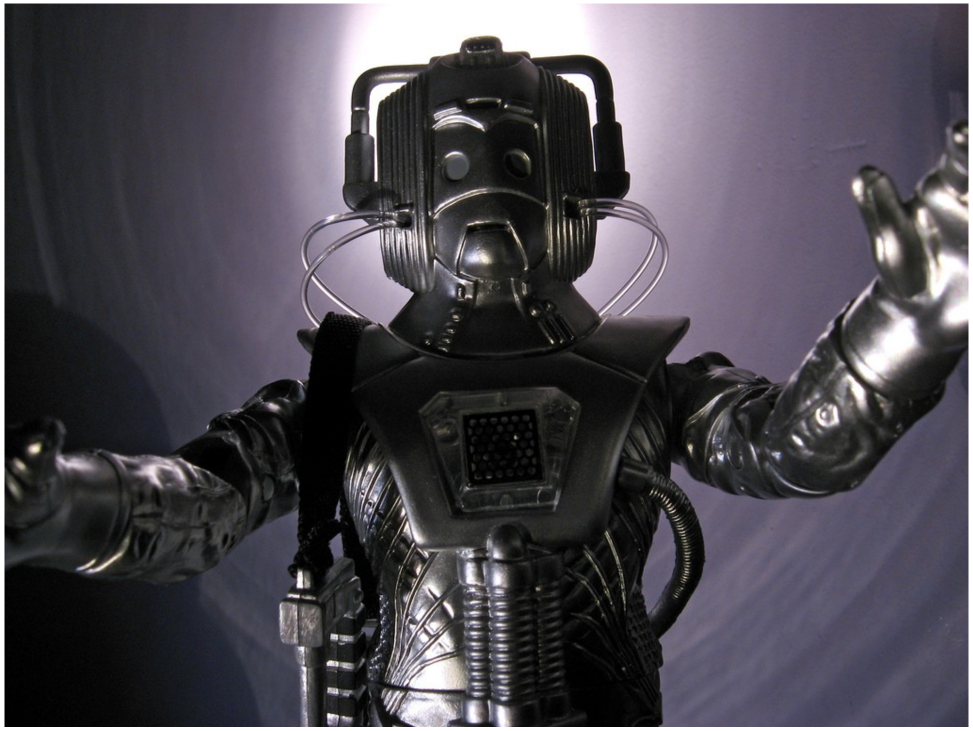
Worst Talk Title Ever

Launchpad Foundations: Learning to Leverage a Component Architecture

CANONICAL

New Attempt

Launchpad Foundations: If the Abstractions Don't Kill Us...



CANONICAL

What is Launchpad?

- A code hosting and software collaboration platform
- Code hosting, bug tracking, translations, mailing lists, Ubuntu package building and hosting, specifications, and community support
- Free for open-source projects
- Open-sourced this week
- Used by ubuntu
- Communicate to upstream and downstream projects
- Integrate with external code hosting and bug tracking

CANONICAL

See the tour!

https://launchpad.net/+tour

Thanks for the interviews!

- Francis Lacoste
- Steve Alexander
- Paul Hummer
- Barry Warsaw
- Michael Nelson

“Component Architecture”?

- A component: an object that provides an interface.
- Interfaces are usually first-class objects.
- Software components agree on one or more expected interfaces, and can interoperate and introduce new functionality and tools because of the agreed-upon contract.
- UNIX pipes, CORBA, CASE, XPCOM

Our Abstractions

- Interfaces from the zope.interface library
- Adapters and utilities from zope.component
- Docs: http://www.muthukadan.net/docs/zca.html

Today’s Plan

- Some of our Abstractions: Concepts and Examples
- Launchpad’s Usage
- How It Worked Out For Us
- How We Might Improve
- How You Might Use the Abstractions

Component-Based Abstraction 1: Dependency Injection via Utilities

- Import a module, or...
- ...look up a utility via an interface.

For example,

- import a library to send email, or...
- ...look up a utility to send email.

UTILITIES

Component-Based Abstraction 2: Adapters, or Wrapping

- Cast an object to a new type, or...
- ...adapt an object to match an interface.

For example,

- cast an integer to a float, or...
- ...adapt an object to an interface for being a message target.

ADAPTERS

CANONICAL

Component-Based Abstraction 3: Multiple Dispatch or Multi-methods for Factories

- Instantiate a class with your arguments, or...
- ...request a new instance providing an interface, given your arguments.

For example,

- instantiate a specific view class with a model object and a request, or...
- ...request a new instance providing a view interface, given a model object and a request.

But wait, there's more! (events, named adapters...)

Launchpad's Usage

- Agile formality
- Pervasive interfaces
- Hidden components
- Exposed components

Launchpad's Usage: Agile Formality

- We use interface-based abstractions in a language without native interfaces: Python
- XML configuration
- Tries to combine the agility of a dynamic language like Python, and the built-in structured formalism of more systematic languages like Java.

Launchpad's Usage: Pervasive Interfaces

- zope.interface, with zope.schema for more specific data descriptions
- Interfaces for library and application code

Launchpad's Usage: Hidden Components

- Infrastructure set up years ago
- Customizes core libraries via utilities and adapters
- zope.security, web publisher and url dispatch
- lazr.restful...

(lazr.restful)

- Automatically generates a RESTful web service from annotated interfaces
- Generates WADL, consumed by lazr.restfulclient
- Rocks
- Leonard Richardson, Francis Lacoste
- Open-sourced, but still hard to get started
- Improving: we are using it internally, polishing
- Soon start with annotated interfaces + WSGI + URL dispatch instructions
- Keep an eye on this!

Launchpad’s Usage: Exposed Components

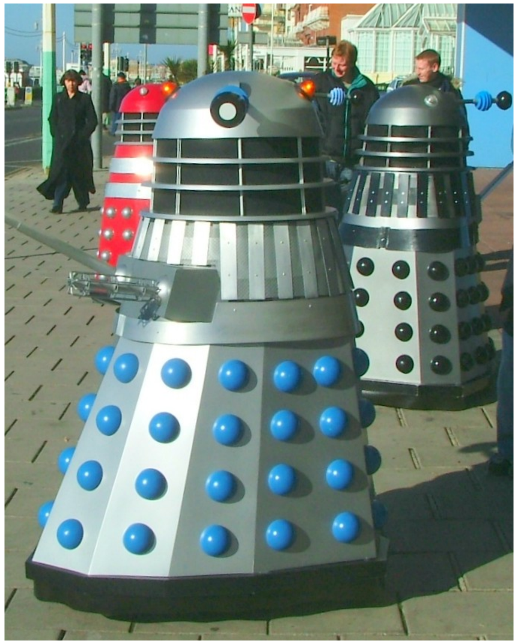
- Great for testing: model external services with utilities (external bug trackers, for instance)
- Great for pretty code: adapters for interactions across subsystems (question targets, for instance)

Launchpad’s Usage: Summary

- In interviews, the more infrastructure responsibility an engineer had, the more they were aware of how we use components
- Interfaces the most obvious result
- Some tension of “agile formality” was often evident

CANONICAL

How did it work out, living with these abstractions?



Expected advantages to component-based abstractions

- Replaceability and Reuse? (yes, but historical, infrequent, foundational)
- Refactoring? (somewhat, but less than might be expected because we consciously limited our use of the tools)

CANONICAL

If the abstractions don’t kill us…
…they’ll make us stronger.

The Good: Interfaces

- Most engineers liked them inherently
- Documentation
- Separation of contract from implementation

The Good: Tools Using the Component Architecture

- Tools are unique and powerful
- zope.security, configured via interfaces and utilities, gives an advantage difficult to duplicate without reusing or duplicating similar machinery
- lazr.restful made the pervasive interfaces more palatable
- Form generation was a more mixed win, but still a win

The Good: Configuring Infrastructure

- Generally, configuring infrastructure with components, adapters and utilities was perceived to be a win
- repoze.bfg is a new framework that uses the same libraries in a similar way

The Good: A Nice Plug-in API

- Engineers generally praised the API we used as a general tool for making pluggable behavior
- Some criticized the cost in developer comprehension and in the overhead of writing and maintaining interfaces

The Good: Test Stubs

- The abstractions are a very nice way to support test stubs
- Much nicer than monkeypatching!

If the abstractions don't kill us...
...nothing will.

The Bad: Interfaces

- In application code, interfaces felt like DRY violations
- If an interface has only a single implementation, felt like writing C
- Victim of automation success: supposed to be developer docs (good for lazr.restful) but usage in forms and security sometimes muddies the water
- Foreign to Python...

The Bad: Barriers to Design and Understanding

- …"Agile formality" brings a foreign philosophy to Python
- Too many new ideas to learn
- Raises the bar for the abstraction tools' APIs
- Factoring your interfaces is difficult to get right, as seen in both Launchpad and some of the zope libraries (publishing and URL dispatch)

CANONICAL

(Zope APIs good, but...)

- “self” is missing from interface call signature. A good reason, but a barrier to learn
- zope.schema is fairly heavyweight
- “multiadapter” frequently cited as confusing. (name? Algorithm?)
- Extensions to the basic ideas break analogies, and sometimes break initial understanding, making it feel that you never really know all of the tool (named adapters and utilities, local registries, adapting to the null interface)

The Bad: Barriers to Opportunistic Coding

- Systematic tools fight writing opportunistic, goal-oriented code
- “copy-paste-modify” coding: is it important to support?
- Cost of an abstraction increases if everyone on your team needs to understand it entirely
- Increases further if everyone who contributes to your code needs to understand it
- (Increases further if everyone who uses your code needs to understand it entirely)
- Many people want systematic coding anyway (TDD, for instance)

The Bad: Barriers to Starting

- Not only hard to understand abstractions, but hard to understand available components and configure them
- Default configurations (repoze.bfg, Grok) help a lot
- Sometimes finding the right place to change configuration is a needle-in-a-haystack search
- Some libraries and frameworks help this by using the abstractions very selectively and putting defaults inline

CANONICAL

The Bad: Registry Configuration and Debugging

- The registry is the central place that adapters and utilities are registered. It is the mechanism by which the indirections are resolved.
- We use Zope Configuration Markup Language to configure (ZCML)
- ZCML: XML (disliked)
- ZCML: external file (disliked, especially for application code)
- Debugging the indirection can be painful

How We Might Improve

IN CASE OF ZOMBIES
FEDERAL
Ammunition
24 JACKETED HOLLOW POINT ROUNDS
PROFESSIONAL GRADE STOPPING POWER
keep out of the reach of children
BREAK GLASS

Launchpad Change: Educate Developers

- Many still don’t understand our abstractions completely
- For better or worse, they often don’t have to

Launchpad Change: Simplify Model Code with Adapters

- Large model classes: basic API plus API for subsystem interactions
- Put basic API in model, API for subsystem interactions in adapters?

Launchpad Change: Set a High Bar for Creating Indirection with the Component Architecture

- Make sure developer and code reviewer understand and agree on new patterns for using the abstractions
- Be especially careful of nested indirection: can become particularly painful spaghetti code

Launchpad Change: Discard XML-based Configuration Language in Application Code

- Grok project provides tools to put registrations inline with Python
- Simplifying configuration machinery further would be nice, but there is some inherent complexity

Launchpad Change: Reduce or Simplify Usage of Interfaces in Application Code

- While some engineers would like to reduce our use of interfaces in the application code, it would be difficult because we get so much automation from them (security, forms, webservice)
- Maybe to ease the application usage and reduce the feeling of DRY violations, we can make a tool to generate interface objects from introspecting our model classes, maybe parsing epydoc docstrings?

Launchpad Change: Use and Build Tools to Debug the Registry

- We should use zope.app.apidoc. (See http://apidoc.zope.org/ for example output.)
- We should investigate building additional debugging tools for the registry. One idea: a flag that makes the registry verbosely log its decisions, like the python -v flag outputs imports.

Launchpad Change: Build Schema Support on Top of an Existing Form Library

- zope.formlib now
- z3c.form is next generation for many developers. Investigate.
- Using a schema for form generation often causes pollution between developer documentation and user interface, despite some tools to help.
- Maybe discard form automation from schema? Maybe try to build schema support on top of a more generic form library?

Component Architecture Change: Retool API to Stay Closer to Familiar Patterns

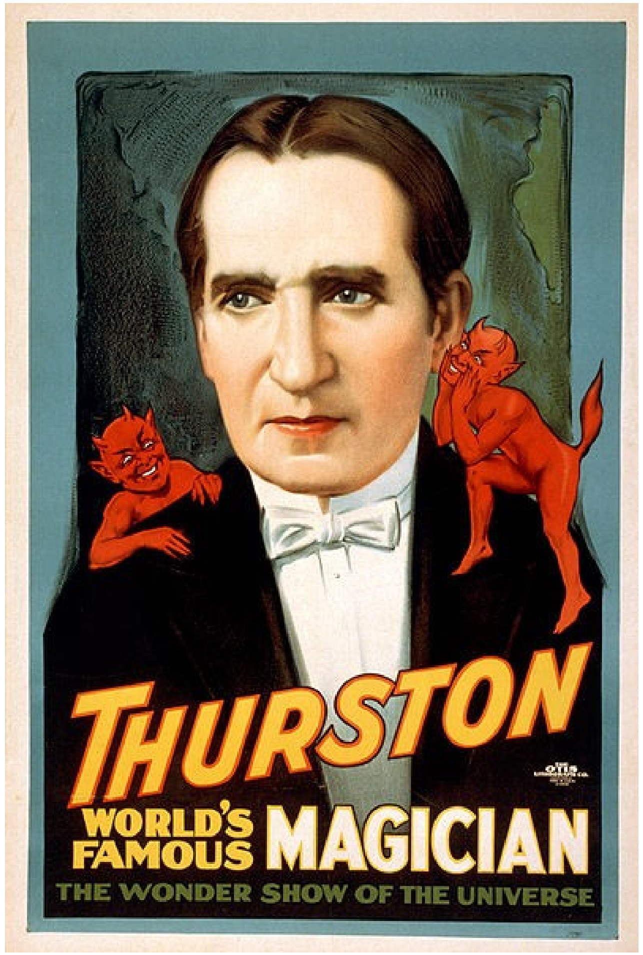
- Adapting looks like mostly like casting in our toolkit. This is a win for understandability.
- Replace multiadapters with more familiar parallel? Mimic class instantiation?
- Generally, simplify and reduce the API, try to build on Python programmers' existing knowledge be drawing parallels.

Component Architecture Change: Use Abstract Base Classes

- New in Python 2.6/3.0
- Part of the language, so reduces new ideas
- Spelling is identical to normal class (e.g., includes “self”)
- Semantic problem: “is instance” is a different question that “provides.” Hiding the difference entirely does not let a user discover the difference, reading the code. The difference can be important.
- Technical problem: ABC implementation of “register” is one-way, so you cannot ask a class what ABCs it provides.
- Usage problem: only for classes. (Small problem?)

CANONICAL

How You Might Use the Abstractions



Lessons

- Use the indirections sparingly, ideally only driven by real need. Be especially wary of nested indirections.
- Consider hiding away the abstractions behind higher-level APIs.
- When the abstractions are exposed, use the simplest possible versions that you can.
- Weigh the value of interfaces for different parts of your code. As a guideline, the closer code gets to “glue code,” the more likely an interface won’t be valuable.
- If you have a component registry like the one we use from zope.component, get familiar with its debugging story, and make sure it will be sufficient for your needs.

CANONICAL

Example Use Case: Allow Replacing Django's ORM

- (Forgive the naivety)
- Possible to describe framework's interactions in a constrained interface?
- If so, maybe provide one, look up a utility for the ORM, defaulting the usual one.
- Other ORMs maybe just need to provide a small wrapper?

CANONICAL

Example Use Case: Django's Security

- Security in the view works well for many applications
- May not scale for some projects, or provide the right security profile
- A configurable, model-based, white-list security system that exposes attributes based on interfaces, and has a pluggable policy via dependency injection, may be a win sometimes
- Might not be that hard to integrate, with a view subclass and some usage patterns.

CANONICAL

Example Use Case: Customizing Behavior of Django Applications

- Django applications are functionality libraries, like event calendar
- Customization points typically strings and view classes?
- What if you want to send email differently, or change the policy for when a calendar event is included in an RSS feed, or change some other behavior?
- Model behavior and other underlying behavior is usually only customizable by monkey-patching or forking.
- Instead, there could be an easy way to specify an object with a specific behavioral responsibility that could be plugged in, but that could operate with a default if no customization has been requested.

CANONICAL

The abstractions make us stronger.

We think.

Now let's make them better, and Launchpad too!

Thank you!

gary.poster@canonical.com

CANONICAL

Image Credits

- *Attack of the Cyberman* http://www.flickr.com/photos/54459164@N00/ / CC BY-NC-SA 2.0
- *Zombie Walk in Edmonton*, Mark Marek Photography ©2007 from http://commons.wikimedia.org/wiki/File:Zombie-walk-kids.JPG
- *Rocky Horror Monster Show* http://www.flickr.com/photos/kevingoebel/ / CC BY-ND 2.0
- *Magic* http://www.flickr.com/photos/cayusa/ / CC BY-NC 2.0
- *Say Cheeeeeeeese* http://www.flickr.com/photos/elsie/ / CC BY 2.0
- *"Silence is the virtue of fools." Sir Francis Bacon* http://www.flickr.com/photos/geekgirly/ / CC BY-NC-ND 2.0
- *In case of zombies...* http://www.flickr.com/photos/samsmith/ / CC BY-NC 2.0
- *Thurston magician poster* Public domain http://commons.wikimedia.org/wiki/File:Thurston_magician_poster.jpg

CANONICAL