Building Component-Based Applications with PEAK

Release 0.5a4

Phillip J. Eby

June 11, 2003

Email: transwarp@eby-sarna.com

Abstract

PEAK, the “Python Enterprise Application Kit”, offers unique tools for creating “enterprise” applications from components. “Building Component-Based Applications with PEAK” introduces Python developers to the basic techniques for using the PEAK binding, naming, and config packages to build flexible, reusable, and configurable components, and to assemble them into large-scale applications.

CONTENTS

CHAPTER

ONE

Introduction: Component-Based Development for Enterprise Applications

1.1 What is PEAK?

PEAK is the "Python Enterprise Application Kit". If you develop "enterprise" applications with Python, or indeed almost any sort of application with Python, PEAK may help you do it faster, easier, on a larger scale, and with fewer defects than ever before. The key is component-based development, on a reliable infrastructure.

PEAK is based on several years of hard-won experience developing and managing mission-critical Python enterprise applications. Its authors have had to use and administer for some time, virtually all of the technologies present in PEAK. Although nearly all of PEAK is being written "from scratch", it was created to replace existing tools that we've enjoyed – or struggled with – over the years.

Our apps require 24x7 availability for thousands of employees worldwide. So anything we build has to be "stone-axe reliable." We want debugging to be a simple and painless process, so we've designed PEAK to make applications "easy to reason about". We use techniques like "lazy immutability" to prevent unintended side-effects, and "complain early and often" to ensure that broken code announces itself as broken *before* it causes bugs or data corruption to creep in. We've learned our reliability lessons from several previous generations of (mostly unpublished) application and component frameworks, and applied them to this new, best-of-breed framework.

1.1.1 What PEAK is for: Enterprise Applications

Enterprise applications are complex, reflecting the complexity of the environments they exist in and support. The "enterprise" is often a large corporation with many subsidiaries, each of which may have been acquired as a previously-independent company with its own product lines and business processes. With each acquisition, an enterprise inherits new "legacy" systems, which must be integrated or phased out. Databases and applications are legion, system administration is vital. There are multiple developers, some inside the company, some outside, some from different divisions. Repetition is everywhere, but also variation. For example: the New York database is on the same back-end as the one in London, but the data model is slightly different, and the time zones on stored dates are different. One thing doesn't vary, however: the users want new functionality, and they want it yesterday.

In such an environment, non-functional requirements – the "ilities" of development – reign supreme. Reusability, configurability, flexibility, extensibility, reliability, portability: these are the stuff "enterprise" applications must be made of. The authors of PEAK have built their professional reputations not only on their applications' reliabilty, but on their speed of development as well. So PEAK incorporates their best ideas for making application components highly reusable, extensible, and configurable, "right out of the box".

1.1.2 What PEAK is part of: Python Enterprise

The value proposition of J2EE is that you can have industry standards, whose implementations are supplied by competing vendors, to address the "ilities" across a variety of concerns, such as data storage, component assembly, messaging systems, and so on. But J2EE encompasses over a dozen technologies of varying complexity and immense APIs. Java is also not a compact language, nor is it especially suited to rapid development. Last, but not least, the sheer scope of J2EE ensures that implementations are resource-hungry, and complex to install, manage, and maintain, especially for the smaller enterprise, or small department within a larger enterprise. If you wish to do more with less, J2EE is probably not for you.

Python, on the other hand, is an easy-to-use language that "fits your brain", and comes with "batteries included". Often an idea that takes half a dozen classes and several hundred lines of convoluted code to express in Java, can be expressed in two classes and less than a hundred lines of crystal-clear Python. Indeed, Python is so expressive that it can take far less time for one to write an alternative implementation of a J2EE technology in Python, than it takes to understand that technology in the first place! Many J2EE technology concepts were actually developed independently in Python applications and libraries, long before they appeared in Java. But, compared to Java's marketing juggernaut, Python tools for the enterprise have had relatively little market exposure.

And so, there is an increasing movement in the Python world to develop and/or promote serious alternatives to J2EE technologies for use with Python, so that developers can leverage Python's lower total cost of ownership in application development and maintenance. The authors of PEAK see PEAK as part of this larger "Python Enterprise" movement, along with the Zope 3 application server, and other Python APIs and technologies. Zope 3 can be seen as an alternative to Java servlets, JSP, and EJB containers for "session beans" and "message-driven beans". PEAK supplies tools for creating the equivalents of "managed beans" and "entity beans", and for implementing parallels to other J2EE technologies such as JNDI and JMS. It is designed to integrate well with the Zope 3 object publishing architecture, and with other Python Enterprise technologies. (A detailed comparison of all the technologies in the J2EE and Python Enterprise worlds is outside the scope of this tutorial, but may be addressed later in a separate white paper.)

1.1.3 What PEAK is made of: Application Kit

So what are these PEAK technologies? PEAK is an application kit, and applications are made from components. PEAK provides you with a component architecture, component infrastructure, and various general-purpose components and component frameworks for building applications. As with J2EE, the idea is to let you stop reinventing architectural and infrastructure wheels, so you can put more time into your actual application.

The most basic part of PEAK is the component binding package: a set of tools for constructing components out of other components, and "binding" separate components together. Then, PEAK supplies a component naming package that provides a common API for associating names with components, looking them up by name, and handling "addresses" of objects like database connections, mail servers, log or lockfiles, and other infrastructure components. Third, PEAK supplies a component configuration package, which makes it easy to provide utilities and configuration data that "trickle down" through the components of an application.

PEAK also supplies, or will supply, several other packages dealing with application domain components, storage, transactions, internationalization, and system operations such as task scheduling, logging, and so on. But all of these other packages depend on the three core packages of binding, naming, and configuration, which are the subject of this tutorial. Understanding how to use these core packages is essential for building PEAK applications.

Luckily, PEAK is compact. At the time we started writing this, the binding, naming, and config packages consisted of only about 4,000 lines of Python (including docstrings, comments, and whitespace, but not counting unit test modules or shared utility modules). So it won't take you long to master their use, with the help of this tutorial.

1.2 Using this Document

Building Component-Based Applications with PEAK introduces the basic techniques for using the PEAK binding, naming, and config packages to build flexible, reusable, and configurable applications from components. To get the best results from this tutorial, you should:

- Have a solid grasp of the Python language, version 2.2 (we recommend checking out *What's New in Python 2.2* at http://www.amk.ca/python/2.2/ if you are familiar with earlier versions, but not 2.2).
- Have Python 2.2.2 or greater installed on your computer, along with PEAK, so that you can try out the sample code and exercises in this tutorial.

1.2.1 Formatting Conventions

This document uses the following formatting conventions:

- Code samples and the names of actual classes, functions, will be in a `fixed pitch font`.
- Text intended to be typed at a Python interpreter prompt is prefixed with the interpreter prompt, '`>>>`'.
- When a new, important term is first introduced, it will be marked in **boldface text**.

1.3 Getting Started with PEAK

1.3.1 Package Layout and API Conventions

PEAK is installed as a set of Python packages, such as `peak.binding`, `peak.naming`, and so on, within the top-level `peak` package space. To help distinguish between the "public" and "private" portions of the code, each subpackage includes an '`api`' module that exports its API classes, functions, constants, and interfaces.

The top-level package also includes an API subpackage, `peak.api`, which contains each of the other subpackages' API modules, named for the subpackage. In other words, '`from peak.api import binding`' produces essentially the same result as '`import peak.binding.api as binding`'.

For convenience, you can also use '`from peak.api import *`' to import all the API subpackages. Then, you can access any `peak.binding` API class such as `Once` by simply referring to `binding.Once`. In this tutorial and all PEAK documentation and code examples, we'll refer to APIs following this convention: major subpackage followed by a dot and the class or function name.

In general, it's most useful to use '`from peak.api import *`' to access the PEAK API. Not only does this give you immediate access to all the subpackage API modules, this will also import several other useful variables, like the special '`NOT_FOUND`' and '`NOT_GIVEN`' objects, and various logging functions like `LOG_CRITICAL` and `LOG_DEBUG`.

Experienced Python programmers may wonder whether using this approach will cause all of PEAK to be imported. In fact, it won't, because the `peak.api` subpackage uses a "lazy import" mechanism. Individual API subpackages like `peak.binding` won't actually be loaded until one of their attributes are accessed. This prevents wasteful up-front loading of all the modules and classes. See the source for `peak.api.__init__` and the `lazyModule` function in `peak.binding.imports`, if you'd like to know more about how it works.

Interfaces and Exceptions

In addition to an '`api`' module, each major subpackage also contains an '`interfaces`' module, which defines all of the interfaces used or provided by classes in that package. The '`interfaces`' modules contain almost no executable code: they are there to document the interfaces only. This makes them a good place to start learning about PEAK packages, especially ones that are frameworks, like `peak.naming` or `peak.storage`.

Another useful module to know about is `peak.exceptions`, which defines the exception classes used by PEAK's major subpackages. You probably won't use it often, though, since there aren't many PEAK exceptions that you'll want to explicitly catch. Most PEAK exception types indicate issues with your code or design, not runtime problems.

NOT_GIVEN and NOT_FOUND

Some PEAK API's use the special objects '`NOT_GIVEN`' and '`NOT_FOUND`' to represent special kinds of "null" values. '`NOT_FOUND`' is often used as a return value that signifies a desired item was "not found" in a cache, registry, or other mapping. This is often done in preference to raising a `KeyError` for both performance and semantic clarity. A `KeyError` could have been caused by some failure internal to an object, while returning '`NOT_FOUND`' indicates the method was successfully executed, but the item was simply not found.

'`NOT_GIVEN`' is rarely used as a return value; its primary purpose is to be a default value for argument(s) to a function or method. It's used in place of '`None`' as a default value, when the argument could legitimately have a value of '`None`' or even '`NOT_FOUND`'. A function can tell whether a value was supplied, by whether the argument is the '`NOT_GIVEN`' object. You'll probably never need to supply this value to a PEAK API call, or return it from a method, but you may find it useful in defining your own APIs.

Note that you should only compare '`NOT_GIVEN`' and '`NOT_FOUND`' using the Python '`is`' operator, e.g. '`if someParam is NOT_GIVEN: doSomething()`'.

PEAK's Major Subpackages

api
: The `peak.api` package supplies commonly used classes, functions, modules, and constants needed by PEAK applications and the PEAK framework itself.

binding
: The `peak.binding` package defines base classes for application components, and for the attribute bindings used to connect components together. These base classes are used throughout PEAK to implement all sorts of components.

config
: The `peak.config` package provides a framework for "placeful" and "lazily immutable" lookups of configuration data. Configuration data can be arbitrary objects or values, and can be looked up by either a property name, or by an interface that the desired object supports (similar to the Zope 3 concept of **utilities**).

model
: The `peak.model` package provides a framework for creating persistent, application domain class families. It provides abstractions such as "Structural Features" that can define fields and associations based on metadata, and automatically generate methods for those features based on code templates. The resulting persistent objects can be saved or loaded by subclassing `DataManager` classes from the `peak.storage` package.

naming
: The `peak.naming` package provides services comparable to those of Java's JNDI system, only easier to use and extend. Naming services are a standardized interface for looking up or recording the location of objects, whether they're database connections, printers, web services, or even something as simple as log files.

running
: The `peak.running` package provides runtime environment tools for logging, locking, process control, event loops, command line apps, periodic tasks, CGI/FastCGI web publishing, etc.

storage
: The `peak.storage` package provides APIs and components for handling transactions, database connections, and persistence.

PEAK's Minor Subpackages and Modules

exceptions
: The `peak.exceptions` module contains the definitions of all exception classes used by PEAK's major subpackages. It doesn't include exceptions from minor subpackages such as `peak.util`.

metamodels
: The `peak.metamodels` package contains metamodels; that is to say, class families representing metadata for object models. For example, `peak.metamodels.UML13` contains classes implementing the UML 1.3 specification, `peak.metamodels.MOF131` implements the MOF 1.3.1 specification, and so on. The packages here are mainly useful for creating tools that work with XMI files from CASE tools, or creating CASE tools of your own. It usually isn't needed for applications that aren't themselves CASE tools.

tests
: The `peak.tests` package controls the execution of PEAK's unit tests. In addition, many major and minor subpackages contain subpackages named `tests`, which contain the unit tests for that section of PEAK. The `test_suite` function in each `tests` module returns a `unittest.TestSuite` for its parent package.

util
: The `peak.util` package is a collection of generally useful modules that don't rely on anything else in PEAK, and thus could potentially be used independently of PEAK. Includes modules for dealing with Python bytecode, XML handling, simple table-like data structures, UUID/GUID handling, and more.

CHAPTER

TWO

Defining and Assembling Components with `peak.binding`

2.1 Component-Based Applications

What's in a component, anyway? Why use them to develop software? Software developers have dreamed for decades of a future where applications could be built by simply plugging together off-the-shelf components. In some development environments, this is at least partly reality today. Many GUI programming tools let you construct at least the visual parts of an application by assembling components.

The promised benefits of component-based development architectures include reusability (and therefore less repetitive work), reliability (if each part works separately, and they are assembled correctly, the whole assembly should work), and ease of understanding/maintenance (because parts can be understood separately).

To be useful, a component architecture must include ways of:

- connecting components to form an application,
- packaging and distributing the components, and
- separating the work of an application into components.

Let's look at how PEAK addresses these issues.

2.1.1 Composing vs. Connecting

Imagine a car. It's composed from a variety of parts: the wheels, engine, battery, frame, and so on. Some of these parts are also composed of parts: the engine has a block, cylinders, pistons, spark plugs, and so on.

Each part in this "component assembly" can be a part of only one larger part: its **parent component**. The hubcaps are part of the wheels, and so they can't also be part of the engine. (They wouldn't fit there in any case, but that's beside the point.) Consider that screws or bolts may be used in many parts of the car: each is part of only one other part of the car, although more than one of the same *kind* of part may be used in other places. In the UML (Unified Modelling Language) and in PEAK, this kind of parent-child "assembly" relationship is called **composition**: a component is being "composed" by assembling other components.

But in the UML and in real life, this isn't the only way of building things with components. It would be very inefficient if every light and accessory in your car had to have its own, independent electrical system. Ways of *sharing* components are needed. In the car, wires, pipes, hoses, and shafts serve to *connect* the services provided by shared components to the places where they are needed. Note that such connections may be between components at any level: wires carry electricity to every electrical part, no matter how big or small. In some cases, wires go to a major

subsystem, which then has internal wires to carry electricity inward to its parts, or to carry signals between its parts. In the UML, these kind of "shared" or "peer-to-peer" connections are called **associations**.

Implementing Components in Python and PEAK

In the Python language, components are Python objects, and composition and association relationships are represented using objects' attributes. Using the `peak.binding` package, you'll create **attribute bindings** that define what sub-objects will be created (via composition) or external objects will be referenced (via association) by each attribute of a component.

Of course, there are some important differences between software and the real world. In the real world, we have to actually build every part of the car "ahead of time", and we must have one screw for every place a screw is needed. In software, classes let us define the concept of a "screw" once and then use it as many times as we want, anywhere that we want.

Also, with PEAK, bindings are "lazy". What this means is that we can define an "engine" class whose parts aren't actually created until they're needed. We just list the parts that are needed and what attributes they'll be bound to, and when the attribute is used, the part is automatically created or connected, according to our definition.

Since each part "magically" appears the first time we want to use it, it's as though it was always there. It's as if your car was an empty shell until you opened the door or looked in the window, at which point all the contents magically appeared. And then when you got into the car, the radio was just an empty shell until you tried to turn it on, at which point all of its internal components sprang into being and wired themselves together.

This "lazy" construction technique can speed startup times for applications which are built from large numbers of components, by not creating all the objects right away, and by never creating objects that don't get used during that application run.

Component Interfaces

You can't just hook wires between random parts of your car and expect good results. In the same way, software components can only be "wired" together if they have compatible interfaces. A software component that expects to send data to a "file" component must be connected to a component that provides the same services a file would provide, even if the component is not actually a "real" disk file.

A specific set of services that a component provides is called an **interface**. Interfaces can denote a component's requirements, as well as the guarantees that it provides when those requirements are met. In PEAK, interfaces are defined and declared using the `protocols` package, which also supports the use of Zope X3 and Twisted interfaces. This means that components you create with PEAK's component architecture can also potentially work in Zope X3 and Twisted's component architectures. (Which may be useful if you plan to run PEAK applications and web services under Zope X3 or Twisted.)

Interfaces in PEAK are used primarily as documentation and as a way of finding compatible components and services. They can also be used to register adapters (which convert a component from one interface to another), declare web views of application components, and even define security restrictions based on interfaces.

See Also:

Component Adaptation + Open Protocols = PyProtocols
(http://peak.telecommunity.com/protocol_ref/module-protocols.html)
: The PyProtocols reference manual has a lot of background information on interfaces, protocols, and adapting components to work with each other. It's also the reference manual for the interface library used by PEAK.

Programming with the Zope 3 Component Architecture
(http://dev.zope.org/Wikis/DevSite/Projects/ComponentArchitecture/Zope3PythonProgrammerTutorialChapter1/)
: If you're interested in developing Zope X3 applications and want to learn more about what you can do with interfaces in the Zope X3 component architecture, check out this tutorial.

Twisted Components: Interfaces and Adapters
(http://www.twistedmatrix.com/documents/howto/components)
: A brief and sometimes amusing look at the basic ideas of interfaces and adapters in Twisted.

2.1.2 Applications = Components + Bindings

For now, we're going to skip over the issue of packaging and distributing components. Since they're implemented as Python objects, we can use virtually any of the standard techniques for packaging and distributing Python code, such as the `distutils` package, or perhaps more elaborate systems such as Gordon MacMillan's cross-platform *Installer* package.

So let's move on to the third major aspect of a component architecture: separating the work of an application or system into components.

Why compose and connect?

Is it realistic to expect to be able to define an application entirely in terms of components? And why would you want to do it in the first place? Can't we just write an application the "old-fashioned way"? That is, import the exact components we want, and use them wherever and whenever we want, instead of creating bindings to link them to a master "application" component?

Well, you could, but one of PEAK's goals is to improve reusability. Consider this: all cars have engines, but different models of car have different goals or requirements for their engines. If we are creating a "Car" application, wouldn't it be nice if we could switch out the "engine" component when we get a new project, without having to maintain multiple versions of the code? Perhaps for this project we need the "car" to be compact, or perhaps we need a bigger engine for a station wagon this time.

How can we achieve such reusability? It actually requires only a simple coding convention: never write code in a function or method that directly references another class. Instead, define *all* collaborations with other classes by way of instance attributes. This means that collaborating component classes can be easily substituted in a derived class, substituting a new "engine" class, for example.

While you can apply this technique of "hiding collaborators" in any object-oriented language, PEAK takes the approach to a new level. Because PEAK attribute bindings can be used to programmatically define connections in context, components can actually seek out their collaborators dynamically at runtime, via configuration files or other sources. This can be as simple and ubiquitous as looking up what database to connect to, or as complex as specifying strategy components to select algorithms that are optimal for a specific deployment environment.

We're back to the "ilities" again: reusability, composability, extensibility, and flexibility, in this case. (Maybe portability, too.) We even get a bit of understandability and maintainability: when we isolate collaborations in attribute bindings, it becomes a lot easier to implement the *Law of Demeter* and write "adaptive programs".

See Also:

http://www.ccs.neu.edu/home/lieber/LoD.html has lots of links about the *Law of Demeter*, and you can also see http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/object-formulation.html for its "object-oriented" version, if you'd like to know more about this software quality technique. PEAK takes this "Law" very seriously, insisting that code which references even a collaborator class must do so via an instance attribute or other "neighbor" as defined by the Law.

You don't have to know or follow the *Law of Demeter* to use PEAK. (Your programs just won't be as flexible.) But all of our example programs will obey the Law, so you'll have a chance to see how – and how easy it is – to follow it.

Services, Elements, and Features

So what's an application made of? The PEAK approach describes application components in terms of their lifecycles and roles, as follows:

Services
: Similar in concept to "singletons", Services are "well-known" instance objects which exist for the lifetime of the application. For example, a database connection object in an application could be a service component, and so could a top-level window in a GUI application. J2EE "session beans" and "message-driven beans" are also good examples of service components. (Note: don't confuse this general concept of services with the Zope 3 concept of a "Service"; the latter is an example of the former, but not necessarily the other way around.)

Elements
: Elements are typically "problem-domain" objects or their proxies in the user interface. They are created and destroyed by other Elements or by the application's Services. Typical Elements might be "business objects" such as customers and sales, but of course any object that is the actual subject of the application's purpose would be considered an Element. In a mail filtering program, for example, e-mail messages, mailboxes, and "sender whitelist" objects would be considered Elements.

Features
: Features are "solution-domain" objects used to compose Elements, and less often, to compose Services. Features are typically used to represent the properties, methods, associations, user interface views, database fields, and other "features" of a problem-domain Element. Feature objects are often used as a class attribute in an Element's class, and shared between instances. PEAK makes extensive use of feature objects to implement labor-saving techniques such as generative programming.

This breakdown of application components is called the **Service-Element-Feature (SEF) Pattern**. PEAK makes it easy to create components of each kind, but this pattern certainly isn't limited to PEAK. You'll find Services, Elements, and Features in almost any object-based application, regardless of language or platform. But, the pattern is often implemented in a rather haphazard fashion, and without the benefit of explicit "wiring" between components.

When you design an application using the SEF Pattern, your top-level application object is itself a Service. You construct that service from lower-level services, such as database connections, object managers, and maybe even a top-level window object for a GUI application. If your application is to be web-based, or you're constructing a middle-tier subsystem, perhaps it will be composed of web services or the equivalent of J2EE session beans.

The top-level application object may provide various methods or "value-added" services on top of its subcomponents' services, or it may just serve to start them up in an environment that defines their mutual collaborators. You may find later that what you thought was an "application" component, is really just another service that you want to use as part of a larger application. Fortunately, it's easy to change a PEAK component's bindings and incorporate it into a larger system.

Defining the Elements of your application design is also fairly straightforward. Elements are the subject of what your application *does*. A business application might have Customer elements, a web server might handle Page elements, and a hard drive utility might manipulate Partition elements.

Features, on the other hand, are usually provided by frameworks, and incorporated into your application's Elements to create a specific kind of application. For example, a GUI framework might provide visual features that allow mapping Element properties to input fields in a window.

While PEAK does or will provide many Element and Feature base classes you can use to build your applications, these facilities are outside the scope of this tutorial, which focuses primarily on assembling an application's Service components. However, some of the techniques you'll learn here will be as applicable to Element and Feature objects as they are to Service components.

So, let's get started on actually *using* PEAK to build some components, shall we?

2.2 Specifying Attributes Using Bindings

2.2.1 Binding Fundamentals

It's time to write some code! Our objective: create a "car" class that keeps track of its passengers.

```
>>> from peak.api import binding
>>> class Car:
        passengers = binding.Make(dict, attrName='passengers')

>>> aCar=Car()
>>> print aCar.passengers
{}
```

Let's go through this sample line by line. In the first line we import the `peak.binding` API. Then, we create a simple class, with one attribute, `passengers`. We define the attribute using the `binding.Make` function, which creates an attribute binding from a type or function, and an optional attribute name.

Instances of class `Car` will each get their own `passengers` attribute: a dictionary. Now, if you're new to Python, you might wonder why we don't place a dictionary directly in the class, like this:

```
class Car:
    passengers = {}
```

The problem is that attributes defined in a Python class body are shared, so every car will end up with the same passenger dictionary. Experienced Python programmers solve this problem by placing code in their `__init__` method to initialize the structure, like so:

```
class Car:
    def __init__(self):
        self.passengers = {}
```

This works alright for simple situations, but gets more complex when you use inheritance to create subclasses of `Car`. It becomes necessary to call the superclass `__init__` methods, and the order of initialization for different attributes can get tricky.

If you develop in Java or C++ or some other language that has instance variable initializers, you'll be happy to know that PEAK lets you have them in Python too – only better. In most static languages, variable initializers run at object creation time. So, either you create *all* of an object's collaborators at creation time, or you write accessor functions to hide whether the fields or attributes are initialized. This can be quite tedious in complex programs.

But PEAK's attribute bindings are **lazy**. They do not compute their value until they are used. If we take a new instance of our `Car` class, and print its dictionary before and after referencing the `passengers` attribute:

```
>>> anotherCar=Car()
>>> print anotherCar.__dict__
{}
>>> print anotherCar.passengers
{}
>>> print anotherCar.__dict__
{'passengers': {}}
```

We find that the object doesn't really *have* the attribute until we try to access it, but once we do access it, it springs into being as though it had always been there, and it acts like a normal attribute thereafter.

If you're familiar with the Eiffel programming language, you'll notice that PEAK attribute bindings are quite similar to Eiffel's **once functions**, except that they're for instances rather than for classes. A "once function" is computed at most once, the first time its value is referenced. In PEAK, attribute bindings compute their value once, and then cache the result, until or unless the attribute is deleted.

Deriving from `binding.Component`

As we've already seen, attribute bindings can be made to work with "old-style" or "classic" Python classes. There are some drawbacks to that, however. Most visibly, it's necessary to specify an attribute binding's name in its definition, as we saw in our example, i.e. '`passengers = binding.Make(dict, attrName='passengers')`'.

However, if we derive our class from `binding.Component`, we no longer have to do this:

```
>>> class Car(binding.Component):
        passengers = binding.Make(dict)

>>> aCar=Car()
>>> print aCar.passengers
{}
```

Now, it's sufficient to use '`binding.Make(dict)`' to define the attribute. What would happen if we did this *without* `binding.Component`?

```
>>> class Car(object):     # new-style class error messages are more helpful
        passengers = binding.Make(dict)

>>> aCar=Car()
>>> print aCar.passengers
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in ?
    print aCar.passengers
  File "C:\cygwin\home\pje\PEAK\src\peak\binding\once.py", line 420, in __get__
    return self._installedDescr(ob.__class__).__get__(ob,typ)
  File "C:\cygwin\home\pje\PEAK\src\peak\binding\once.py", line 449, in _installedDescr
    self.usageError()
  File "C:\cygwin\home\pje\PEAK\src/peak/binding/_once.pyx", line 125, in _once.BaseDescriptor.
    raise TypeError(
TypeError: Binding was used in a type which does not support active
bindings, but a valid attribute name was not supplied
```

Ouch! It doesn't work, because PEAK can't tell what name the attribute has without help from either you or the base class. (Actually, it's the metaclass, not the base class, but that's not important right now). There are some circumstances where PEAK will try to guess the attribute name for you, such as when you use a class or a function to define an attribute binding, but for the most part you must either have a base class (such as `binding.Component`) whose metaclass supports activating bindings, or else you must supply the attribute name yourself when defining the binding.

So, if you plan to use attribute bindings in your program, it's probably best to subclass `binding.Component`; it'll save you a lot of typing! Most PEAK framework classes for "service" components derive from `binding.Component` already.

`binding.Attribute` - The Basis for all Bindings

All PEAK bindings are created using the class `binding.Attribute`, or a subclass thereof. It provides the basic machinery needed to lazily compute an attribute ”on-the-fly”. Here’s a simple example of its use:

```
class Car(binding.Component):

    def height(self):
        baseHeight = self.roof.top - self.chassis.bottom
        return self.wheels.radius + baseHeight

    height = binding.Make(height)
```

In this example, we want to compute the car’s height based on various other attributes, but we only want to compute it once, and save the value thereafter. To do this, we define a function that takes three arguments: the object, its instance dictionary, and the name of the attribute being computed. Most of the time, you’ll only care about the first argument, which is the object for which the attribute is being computed. It’s rare that you’ll need the instance dictionary or the attribute name, but if you need them, they’re there. Normally, you’ll just return the value you want the attribute to end up with.

Note, by the way, that there’s no need for this function passed to `binding.Once` to be a method, or for the parameters to have specific names. If we continued the class above as follows:

```
    verticalCenter = binding.Once(lambda self: self.height/2)
```

This would be a perfectly valid way to express the idea that the car’s vertical center is half of its height. It’s also true that we could say:

```
    passengers = binding.Once(lambda: {})
```

as another way of saying ‘`binding.Make(dict)`’. However, using ‘`binding.Make(dict)`’ is more compact, and clearer as to intention.

By the way, although we’ve been only showing uses of `binding.Make` with one argument (the function to be called), `binding.Make` does actually take other arguments, such as a default attribute name, an “offer as” specification (more on this later) and a docstring. Check out the PEAK API Reference or the source code for more details.

Re-binding, Pre-binding, Un-binding, and Persistence

We’ve mentioned over and over that attribute bindings are computed only once, but that’s not precisely true. It’s actually possible for them to be computed zero times, or many times, if we set or delete the attribute the binding defines.

You may have wondered, “how does a binding know whether it’s been computed or not?” It knows because there is an entry in the object’s dictionary for that attribute name. (Notice, by the way, that this means objects without dictionaries can’t get much use out of bindings.)

But what if there’s something already in the dictionary? Or what if you remove the value from the dictionary? Well, it’s pretty much as you’d expect. If you set a value for an attribute, then the binding will not compute a value. If you delete the attribute, and try to access it, the binding will recompute the value. This behavior can actually be quite useful in the context of transactions: many PEAK transactional components delete certain bindings at the conclusion of a transaction, allowing them to be recomputed in the next transaction.

PEAK also makes use of the ability to override a binding by manually setting a value for an attribute. For example, most PEAK component classes’ `__init__` methods accept keyword arguments which can override at-

tributes which would otherwise be determined by bindings. This is especially useful in conjunction with the `binding.requireBinding` class, which simply raises an error when you try to access the attribute. It's a handy way of specifying that a component must have the attribute, and that a subclass or instance must supply the value.

By the way, it's important to note that bindings do *not* participate in persistence, as they directly alter the associated object's dictionary, bypassing the normal "set attribute" machinery. You should carefully consider the use of attribute bindings in persistent objects, as to how they will interact with your persistence machinery (e.g. whether they should be saved or not) and whether you will want to code them in such a way as to consider the object "changed" when an attribute is computed.

2.2.2 Creating Attribute Bindings

The `peak.binding` package offers many kinds of attribute bindings for constructing your components. In this section, we'll look at the conventions for how these bindings are created, and at the details of several of the basic binding types.

API Conventions

There are four standard parameters that almost all attribute binding constructors accept. They always use the same parameter names, so you can supply them as keyword arguments.

attrName
: The *attrName* parameter specifies the attribute name that the binding will have. As mentioned previously, this is only needed when the binding is being used in a class that doesn't support active descriptors. If you're using a class that inherits from `binding.Component`, you don't need to supply this parameter.

offerAs
: The *offerAs* parameter is a sequence of **configuration keys**, such as interfaces or `PropertyName` objects. This is used for integration with the configuration system, so we'll explore it in more detail later on. For now, you can just ignore it.

doc
: The *doc* parameter is an optional docstring which will be used for the attribute. This is useful for documenting your class so that tools like *pydoc* and the Python `help` function will display the docstring for the attribute as part of the class' documentation.

uponAssembly
: *uponAssembly* is a flag that indicates whether the attribute should be computed when the object's `uponAssembly` method is called. We haven't covered "assembly events" yet, so don't worry about this one for now.

adaptTo
: *adaptTo* is an interface that the attribute's computed value should be adapted to, before it is cached or set. Since the adapt operation is done whenever the attribute is set, this effectively gives you built-in type checking and conversion for attributes.

suggestParent
: *suggestParent* is a flag that indicates whether the referenced object should be informed that it is being attached to the referencing object, whenever the attribute is cached or set. This allows components to automatically discover their parent component and component name, if they do not already know their parent. By default, this flag is a true value, since most of the time you will want it to apply. However, if you know that the value of the attribute will never be a component, or that it will be a component that should not treat your object as its parent, you can set this flag to a false value.

noCache

noCache is a flag that indicates the attribute's computed value should not be cached. Note that this only stops the computed value from being stored: if you set the attribute, the value you set will be cached and reused until/unless the attribute is deleted. Set this flag to a true value if you do not want the computed value cached. (The default value is false.) (Note that if this flag is set, the *suggestParent* and *adaptTo* keywords will only affect setting the attribute, not computing it, since the value is not cached.)

permissionNeeded

permissionNeeded is either a `peak.security.IAbstractPermission`, or '`None`'. This is a convenience feature for use with the `peak.security` package, so we won't deal further with it here.

Basic Binding Classes/Functions

The following functions and classes can be used as-is to create attribute bindings, or you can subclass the classes to create custom binding types of your own:

`Make`(*recipe, **keywords*)

The most basic of all binding types, the `binding.Once` class requires only a callable object such as a function or lambda. If you don't supply an *attrName*, the '`__name__`' attribute of the *func* parameter is used, if available. Similarly, if you don't supply a *doc*, the '`__doc__`' attribute of the *func* parameter is used, if available.

As we've already seen, the `binding.Make` function just needs a class or type object as its *obtype* parameter, and it creates a new instance of that type.

`binding.Make` will also accept a string for its *recipe* parameter, in which case it will interpret the string as an **import specification**, using the `peak.util.imports.importString` function to load it. We'll talk more about import specifications in the chapter on the `peak.config` package.

`Require`(*description, **keywords*)

The `binding.Require` class is the odd one out in this group. It doesn't compute a value at all! Instead, it raises an error when the attribute is accessed. This is so that you can easily define an attribute that you expect a subclass or instance to provide, and thus get a clearer error message if the expected attribute is never defined.

XXX Need Obtain and Delegate here as well.

2.3 Composing Hierarchies with Bindable Components

So far, we've only looked at binding various kinds of "helper" attributes into components, but not assembling components themselves into larger components or applications.

One issue that arises quite a bit in the assembly of components is the notion of **context**. A component often needs to know "where" it is located, in the sense of its placement in a larger component. While components will typically have many connections to other components, the connection between a component and its container or **parent component** is particularly important. Let's look at an example:

```
>>> class Wheel(binding.Component):
        def spin(self):
            print "I'm moving at %d mph" % self.getParentComponent().speed

>>> class Car(binding.Component):
        wheel = binding.Make(Wheel)
        speed = 50


>>> c = Car()
>>> c.wheel.spin()
I'm moving at 50 mph
```

The `binding.Component` class defines two methods for dealing with parent components: `getParentComponent` and `setParentComponent`. `setParentComponent` is automatically called for you when you create an instance via `binding.Make`, which is why our `Wheel` instance above was able to access its parent component with `getParentComponent`. Let's look at a slightly different version of that example:

```
>>> class Wheel(binding.Component):
        def spin(self):
            print "I'm moving at %d mph" % self.speed
        speed = binding.Obtain('../speed')

>>> class Car(binding.Component):
        wheel = binding.Make(Wheel)
        speed = 50

>>> c = Car()
>>> c.wheel.spin()
I'm moving at 50 mph
```

By now, perhaps the wheels in your brain will be spinning as well, thinking about the possibilities here.

2.3.1 Inspecting Component Hierarchies

The `peak.binding` package supplies several useful functions for examining relationships between components. Let's run through a few examples, continuing the previous example above:

```
>>> print c
<__main__.Car object at 0x00F7BCB0>

>>> print binding.getParentComponent(c.wheel)
<__main__.Car object at 0x00F7BCB0>

>>> class Transport(Car):
        cargo = binding.Make(Car)

>>> t=Transport()
>>> print t
<__main__.Transport object at 0x00FA1670>

>>> print binding.getRootComponent(t.cargo.wheel)
<__main__.Transport object at 0x00FA1670>

>>> print binding.getComponentName(t.cargo.wheel)
wheel

>>> print binding.getComponentPath(t.cargo.wheel)
/cargo/wheel

>>> print binding.getComponentPath(t.wheel)
/wheel

>>> print t.cargo
<__main__.Car object at 0x00FABB70>

>>> print binding.getParentComponent(t.cargo.wheel)
<__main__.Car object at 0x00FABB70>

>>> binding.lookupComponent(t,'/cargo/wheel')
<__main__.Wheel object at 0x00FAB900>
```

As you can see, `binding.getParentComponent`, `binding.getRootComponent`, `binding.lookupComponent`, `binding.getComponentName`, and `binding.getComponentPath` do pretty much as you would expect from their names. (One thing we didn't show, however, was that `binding.getComponentPath` doesn't actually return a string, but a `binding.ComponentName` instance that just looks like a string when printed. More about this later.)

How Hierarchies Are Assembled

Perhaps you've been wondering how the objects know what objects are their parents. In the process of answering that question, we'll pass through lots of interesting or important points along the way.

The short answer to how an object knows what its parent is, is that you tell it. The constructor for `binding.Component` takes as its very first argument, the object which should be its parent.

In our examples above, however, we never passed a parent into a constructor. We created the `Car` instance and `Transport` instance without supplying a parent component. So those instances were created without a parent, which is why when we inspect them, they show up as parentless or **root components**.

In PEAK, a root component is any object without a parent. Only objects whose classes provide a `getParentComponent` method can have parents, and even then they are only considered to have a parent if that method returns something other than '`None`'. Classes without such a method, such as Python built-in types, are almost always root components. For example:

```
>>> print binding.getParentComponent(123456)
None
>>> print binding.getRootComponent([1,2,3])
[1, 2, 3]
>>> print binding.getComponentPath("a string")
/
>>> print binding.lookupComponent("some string", "upper")
<built-in method upper of str object at 0x00F26100>
```

As you can see, the inspection functions provided by `peak.binding` will work with just about any kind of object, and will provide results consistent with the interpretation that objects without a `getParentComponent` method are root components.

Notice also that an object doesn't have to be based on `binding.Component` to work in a component hierarchy; all it needs is a `getParentComponent` method defined in its class. For example:

```
>>> class myGUIControl:
        def getParentComponent(self):
            return self.parentWindow

>>> aControl = myGUIControl()
>>> aControl.parentWindow = "just a demo"
>>> print binding.getParentComponent(aControl)
just a demo
```

This, by the way, is one of the reasons why it's better to use `peak.binding` functions to inspect components instead of calling their `getParentComponent` or other methods directly. `binding.getParentComponent` has code to handle the case where an object has no `getParentComponent` method. Similarly, other inspection functions gracefully handle the absence of appropriate support by the object:

```
>>> print binding.getComponentName(aControl)
None
>>> print binding.getComponentPath(aControl)
/*
```

Notice that since we didn't provide any special support in our example GUI control class for component names, the `peak.binding` system considers it not to have a name. And unknown names show up as '`''*''`' in component path segments.

Of course, if we wanted to support our hypothetical GUI controls having component names, we would need only to add a `getComponentName` method to its class.

But how do we know what methods to add, and what values they should accept or return? The `peak.binding` framework defines an **interface**, `binding.IBindingNode`, that defines what methods the binding framework calls on objects that it expects to be nodes in a component tree. The `binding.IBindingNode` interface looks basically like this:

```
class IBindingNode(config.IConfigSource):

    """Minimum requirements to join a component hierarchy"""

    def getParentComponent():
        """Return the parent component of this object, or 'None'"""

    def getComponentName():
        """Return this component's name relative to its parent, or 'None'"""

    def notifyUponAssembly(child):
        """Call 'child.uponAssembly()' when component knows its root"""
```

By convention, interface names in Python are usually begun with a captial "I", to help distinguish them from regular classes. Also by convention, methods are described from the *caller's* perspective, so '`self`' arguments are not included in method definitions.

So, the `binding.IBindingNode` interface tells us that to implement the Binding Node interface, we need a `getParentComponent` method and a `getComponentName` method, neither of which takes any parameters, and whose return values are as documented.

This interface also inherits from `config.IConfigSource`, which defines some additional methods that a binding component should implement. We'll look at those in the coming chapter on the configuration system. Notice that this does *not* mean a component has to inherit from `IConfigSource`! It simply means that a class which promises to implement the `binding.IBindingNode` interface is also promising to implement the `config.IConfigSource` interface as well.

Did our example `myGUIControl` class promise to implement `binding.IBindingNode`? No. We didn't declare that it does, nor did it inherit from a class that declares support for the interface. So, it's not promising to implement the full `IBindingNode` interface. This is acceptable because `peak.binding` verifies the presence of needed methods automatically, and it isn't required that a component implement all of the `IBindingNode` methods. Most other interfaces however, *must* be explicitly promised by a class in order for the functionality to work. (See the `protocols` package documentation for more information about declaring support for an interface.)

The `binding.IComponentFactory` interface is a good example of this. `IComponentFactory` is the interface which defines how component constructors work - at least if they're to be supported by `peak.binding`!

Let's take a look at the `binding.IComponentFactory` interface in more detail.

```
class IComponentFactory(Interface):

    """Class interface for creating bindable components"""

    def __call__(parentComponent, componentName=None, **attrVals):
        """Create a new component

        The default constructor signature of a binding component is
        to receive an parent component to be bound to, an optional name
        relative to the parent, and keyword arguments which will be
        placed in the new object's dictionary, to override the specified
        bindings.

        Note that some component factories (such as 'binding.Component')
        may be more lenient than this interface requires, by allowing you to
        omit the 'parentComponent' argument.  But if you do not know this is
        true for the object you are calling, you should assume the parent
        component is required."""
```

Notice that this interface describes a `__call__` method. This doesn't mean that you'll need a `__call__` method on your object instances or in your class. It simply means that an object that provides the interface should be callable. Because `IComponentFactory` is an interface for functions or classes, the `__call__` method simply documents what behavior the function (or class constructor) should have.

XXX show how to implement in a class, __init__, __class_implements__, etc.

XXX IComponent interface; setParentComponent

Names and Paths

XXX lookupComponent

2.3.2 Hierarchy-Building Tools

XXX Component, ...

2.4 Connecting Components by Name or Interface

XXX bindTo, Constant, Acquire, ComponentName, ...

XXX Stuff that needs to go to config chapter: iterParents, lookup, iterValues