featuremonkey Documentation

Release 0.2.1

Hendrik Speidel

January 28, 2013

CONTENTS

> `featuremonkey` is a tool to support *feature oriented programming (FOP)* in python.

`featuremonkey` is a tiny library to enable feature oriented programming (FOP) in Python. Feature oriented software development(FOSD) is a methodology to build and maintain software product lines. Products are composed automatically from a set of feature modules and may share a set of features and differ in others.

There are multiple definitions of what a feature really is. Here, we use the definition of Apel et al.:

> A feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option [ALMK] .

When trying to modularize software-systems to acheive reusability, components come to mind. However, there is a problem with that: large components are very specific which limits reuse; many small components often make it necessary to write larger amounts of glue code to integrate them.

So components are nice — but it feels like there is something missing.

Features provide an additional dimension of modularity by allowing the developer to encapsulate code related to a specific concern that is scattered across multiple locations of the codebase into feature modules. Products can then be composed automatically by selecting a set of feature modules.

Common approaches to FOSD are the use of generative techniques i.e. composing a product‘s code and other artefacts as part of the build process, the use of specialized programming languages with feature support, or making features explicit using IDE support.

`featuremonkey` implements feature composition by using monkeypatching i.e. structures are dynamically redefined at runtime.

CHAPTER

ONE

FUN FACTS ON FEATUREMONKEY FOR FOSD PEOPLE

- features are bound at startup time or later (dynamic feature binding)
- however, feature binding is not fully dynamic as there is no way to unbind a feature once it has been bound.
- featuremonkey uses delegation layers that are injected at runtime when composing the features.
- featuremonkey composes objects(instances that is)
- packages, modules, classes, functions, methods and so on are all objects in python ... therefore, featuremonkey can compose those as well.
- featuremonkey uses monkeypatching to bind features: it adapts the interpreter state.

The basic operation offered by `featuremonkey` is `compose`.

CHAPTER

TWO

FEATURE ORIENTED SOFTWARE DEVELOPMENT

CHAPTER

THREE

GETTING STARTED

3.1 Installation

3.1.1 Installing the latest stable version

Make sure you have `pip` installed. `featuremonkey` can then be installed using the following command:

```
pip install featuremonkey
```

3.1.2 Installing the development version

To get the development version of `featuremonkey` directly from github, use:

```
pip install git+https://github.com/henzk/featuremonkey.git#egg=featuremonkey
```

You can check by importing `featuremonkey` from a python prompt. If you don't see an error, everything should be ok.

CHAPTER

FOUR

FEATUREMONKEY REFERENCE

4.1 featuremonkey Reference

4.1.1 Feature Structure Trees

featuremonkey recognizes python packages, modules, classes, functions and methods as being part of the FST.

4.1.2 FST Declaration

FSTs are declared as modules or classes depending on the preference of the user. modules and classes can be mixed arbitrarily.

Note: when using classes, please make sure to use new style classes. Old style classes are completely unsupported by featuremonkey - because they are old and are removed from the python language with 3.0. To create a new style class, simply inherit from `object` or another new style class explicitely.

FSTs specify introductions and refinements of structures contained in the global interpreter state. This is done by defining specially crafted names inside the FST module/class.

FST Introduction

Introductions are useful to add new attributes to existing packages/modules/classes/instances.

An introduction is specified by creating a name starting with `introduce_` followed by the name to introduce directly inside the FST module/class. The attribute value will be used like so to derive the value to introduce:

- If the FST attribute value is not callable, it is used as the value to introduce without further processing.
- If it is a callable, it is called to obtain the value to introduce. The callable will be called without arguments and must return this value.

Example:

```
class TestFST1(object):
    #introduce name ``a`` with value ``7``
    introduce_a = 7

    #introduce name ``b`` with value ``6``
    def introduce_b(self):
        return 6
    #introduce method ''foo'' that returns ''42'' when called
    def introduce_foo(self):
        def foo(self):
            return 42

        return foo
```

Warning: Names can only be introduced if they do not already exist in the current interpreter state. Otherwise `compose` will raise a `CompositionError`. If that happens, the product may be in an inconsistent state. Consider restarting the whole product!

FST Refinement

Refinements are used to refine existing attributes of packages/modules/classes/instances.

An introduction is specified much like an introduction. It is done by creating a name starting with `refine_` followed by the name to refine directly inside the FST module/class. The attribute value will be used like so to derive the value to introduce:

- If the FST attribute value is not callable, it is used as the refined value without further processing. **This is a replacement**
- If it is a callable e.g. a method, it is called to obtain the refined value. The callable will be called with the single argument `original` and must return this value. `original` is a reference to the current implementation of the name that is to be refined. It is analogous to `super` in OOP.

Example:

```
class TestFST1(object):
    #refine name ''a'' with value ''7''
    refine_a = 7

    #refine name ''b'' with value ''6''
    def introduce_b(self, original):
        return 6

    #refine method ''foo'' to make it return double the value of before.
    def refine_foo(self, original):
        def foo(self):
            return orginal(self) * 2

        return foo
```

Note: when calling `original` in a method refinement(for both classes and instances), you need to explicitely pass `self` as first parameter to `original`.

Warning: Names can only be refined if they exist in the current interpreter state. Otherwise `compose` will raise a `CompositionError`. If that happens, the product may be in an inconsistent state. Consider restarting the whole product!

4.1.3 FST nesting

FSTs can be nested to refine nested structures of the interpreter state. To create a child FST node, create a name starting with `child_` followed by the nested name. The value must be either a FST class or instance or a FST module. As an example, consider a refinement to the `os` module. We want to introduce `os.foo` and also refine `os.path.join`. We could do this by composing a FST on `os` to introduce `foo` and then composing another FST on `os.path` that refines `join`. Alternatively, we can use FST nesting and specify it as follows:

```
class os(object):
    introduce_foo = 123
    class child_path(object):
        def refine_join(self, original):
            def join(*elems):
                return original(elems)
            return join
```

Got it?

4.1.4 FST Composition

`featuremonkey.`**`compose`**(*self*, **things*)

compose applies multiple fsts onto a base implementation. Pass the base implementation as last parameter. fsts are merged from RIGHT TO LEFT (like function application) e.g.:

class MyFST(object): #place introductions and refinements here introduce_foo = 'bar'

compose(MyFST(), MyClass)

`featuremonkey.`**`compose_later`**(*self*, **things*)

register list of things for composition using compose()

compose_later takes a list of fsts. The last element specifies the base module as string things are composed directly after the base module is imported by application code

4.1.5 Feature Layout

4.1.6 Product Selection

`featuremonkey.`**`select`**(*self*, **features*)

selects the features given as string e.g passing 'hello' and 'world' will result in imports of 'hello' and 'world'. Then, if possible 'hello.feature' and 'world.feature' are imported and select is called in each feature module.

`featuremonkey.`**`select_equation`**(*self*, *filename*)

CHAPTER

FIVE

CHANGELOG

0.2

- first release on PYPI
- composer is now class based
- fixes compose_later composition order
- initial docs

0.1

- initial version

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[ALMK] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST), volume 5140 of Lecture Notes in Computer Science, pages 36–50. Springer-Verlag, 2008.