Current Shard Approach

30th June 2007

1 Problem

To find some way of splitting and recombining Components by functionality.

Components are written as classes. The main focus has therefore been on adding groups of methods to classes. Modelling these groups as classes themselves gives the possibility of adding any class to any other class, i.e. any level of functionality is potentially a shard to be reused.

Any approach aims to be intuitive for users and developers, and to interfere as little as possible with the normal writing of code. This applies to both the amount of additional code required to fit into a shard framework and modifications that would have to be made to existing code.

2 Alternatives

2.1 Partial

Partial is a module for splitting a class definition across several modules. The segments of class definition would be used as shards in this case. Each segment is defined as a new class that must inherit from partial and its 'home' class. The class partial has a custom metaclass that (through inheritance) overrides the constructor that is called for each segment: instead of instantiating the object, it copies the methods it contains into the segment's home class and returns a reference to it — no reference to the segment class is retained.

Although this limits additional code writing to import statements (of the segment modules) and an extra parent class, it has several problems. Its use of metaclasses makes the initialisation process complex and opaque: modifications have unexpected results, and bugs are hard to track down and explain. It also makes the shard object unavailable in its own right. Amongst other things, subclassing these shards is not possible for this reason. The approach is also intended to add methods back into a single named class only and would require significant modification to increase flexibility.

2.2 Decorator

This is an external function that modifies a class or function after it has been defined. It is much easier to understand and develop than using metaprogramming. Although an arbitrary function is much more flexible in the level at which it can be pitched, I chose to model partial's class-based approach as a starting point. The current implementation is therefore a class decorator that adds a class's (shard's) methods to a target container class after it has been defined.

This approach requires a little additional code to use: as well as import statements, an explicit call to a shard merging function is needed, however the style of supplying

a single-argument function has been used here to take advantage of the neater class decorator syntax when it becomes available.

In contrast to partial, any class can be made into a shard after it has been defined, and any class can import shards. Subclassing shards (where import adds all inherited methods to target class) is also possible, modulo a slight modification to superclass calls. It is also simple to extend: method dependency calculation has already been implemented.

2.3 Rationale for approach

Of the two options, I prefer the decorator approach for its simplicity and greater functionality. It is still similar to partial in its design, i.e. in merging classes. However this does result in a two-level structure: that of the class, and the functions inside it. This grouping can be useful: the usual inheritance mechanisms can be exploited to extend and reuse existing shards, and requirements can be set at a per-class level so that each individual function does not have to declare its dependencies. Given that they are grouped by functionality, I have found this higher level of granularity to be useful.

On the flip side, it does mean that a whole class of methods must currently be imported at a time, and that the pattern of nested functions is interrupted at this level.

In any case, the shard decorator can easily be modified to work on a purely functional level. Its initial form was to accept dictionaries of names -> methods, this was changed to allow easier access to inherited methods. The alternative grouping should this approach be taken would probably be by module.

All of these approaches also modify the classes at runtime, rather than generating code. I believe this is more in line with the 'shard library' idea — dynamic creation allows components to benefit immediately from updates and bugfixes to shards, whereas statically generated code will probably have to be edited after creation each time the shards it contains are changed.

3 Currently Proposed Shard Definition

3.1 Shard class requirements

To take advantage of dependency calculation, or to be compatible with any set of shards and components that currently does, a shard must declare the additional methods it requires to function. Another decorator (requires()) has been written to simplify this; it creates and sets an attribute in the class that the addShards methods can check. Other than this, any class can be used as a shard, and the requires() method can be applied to the class at any time after its definition.

Requirements need not be methods: although the system is primarily designed for them, no type check is enforced on the attributes that are copied.

Currently, a shard cannot define a method or attribute whose name clashes with attributes in its target class. Partial offered a way around this, by supplying an @replace decorator to flag shard methods that will override those in the target class. However, the shard system is designed from a user perspective, i.e. the target class is more likely to be modified than the shard, and as the shard is in a library, it doesn't know which target class attributes it may clash with.

3.2 Target/Container class requirements

To add shards into a component, the relevant shard modules must be imported, and the addShards function called with the classes to add. This call will check that all methods required by the shards are provided; if a component chooses to declare its requirements

(in the same way as shards) then these will be checked as well. This dependency checking does mean that any application of addShards must completely fulfil dependencies between the shards and components involved.

4 Conclusion

The current shard approach seems to be mostly satisfactory. There is some motivation for moving to a function-level approach instead of class-level. The most significant of these that I can see at the moment involves the current implementation of dependency checking: a component must satisfy the dependencies for all methods in a shard, even if it only uses one.

Details aside though, the essential style seems appropriate for the task.