DØ Note XXXX

A Proposal for a Framework using Multiple Inheritance

H. Greenlee

November 11, 1997

Contents

Figure 1: Framework class diagram.

This document contains a proposed framework design based on multiple inheritance of package interfaces. The ideas contained here are derived from ideas presented in the Design Patterns course of Oct. 23,24, as well as discussions in the frameworks group meeting of Oct. 29 and afterward.

1 Overview

The basic idea that is presented here is a way to implement a run 1 style framework in an object oriented way. This design incorporates from the run 1 framework the idea of factorizing programs into hooks and packages. The main design goals are as follows.

1. A program builder should not be needed for generating code.
2. There should not be any such thing as a "master hook list." Adding new hooks should not break existing code.
3. Defining a new package class should be simple.

The design presented here satisfies items 1 and 2 (item 3 is a judgement call). It includes the following elements.

1. A framework program.
2. Abstract hook classes.
3. Package classes.
4. Registration and factory classes.

The relationship between these elements is shown in Fig. 1. Each of these elements is discussed in more detail in the following sections.

2 The Framework Program

The design presented here in principle has little to say about the framework program. Although represented as a single class in Fig. 1, the framework program itself can be completely general. It can be simple or complicated, procedural or event-driven, single- or multi-threaded, Python or C++, etc. There may be more than one framework program.

The main feature of the framework program and its associated classes is that there is no compile time or link time physical coupling between them and any package class. The framework program itself always interacts with packages through abstract hook classes. The lack of physical coupling achieves the first design goal given above, namely, that there is no need for a program builder.

3 Hook Classes

There is an abstract class associated with each hook, and also a common virtual base class that all hook classes derive from. There is no *a priori* requirement on how simple or complicated a package interface can be, although DØ may wish to adopt a standard simple interface, as was done for run 1.

4 Registration and Factory Classes

The following classes participate in the registration process:

1. Package registration class.
2. Package factory.
3. Hook registry.

4.1 Package Registration

The package registration classes are templatized with a template parameter that corresponds to a package class. There is also a non-templatized package registration base class. The purpose of the package registration class is to encapsulate compile time knowledge of the package type via its template argument and to initiate the registration process prior to the beginning of the main program by being instantiated as dummy global variables. Each package defines a single dummy package registration global variable in its implementation file. The constructor of the package registration variable registers itself with the package factory.

4.2 Package Factory

The package factory is a non-templatized singleton class that cooperates with the framework program to instantiate package classes. Packages register themselves (actually their registration variables) with the package factory via global constructors. The package factory lacks compile time knowledge of package classes, and is therefore not physically coupled to any package class. The package registration variable gives the package factory the ability to access standard static methods of the package class, and to instantiate the package class.

4.3 Hook Registry

The hook registry is a set of templatized singleton classes that maintain an association between packages (identified by pointer) and hooks (identified by the template argument of the hook registry class). The association is created when the hook constructors call the register method of the hook registry class at package at instantiation time. The framework calls methods of the hook registry classes to obtain lists

Figure 2: Package class diagram.

of packages to call for each hook. Since the hook registry only knows about the hook interface, it is not physically coupled to package classes.

5 Packages

Each concrete package multiply inherits the interface for each hook for which it wants to be called (see Fig. 2). Packages cause themselves to be registered with the package factory by including a static dummy global registration variable. Registration with the hook registry happens automatically via calls located in hook base class constructors.

5.1 Nested Packages

Nothing in the preceding design prevents the nesting of packages. Packages are free to instantiate and/or call other packages by calling the hook registry and package factory, the same as the framework program.

6 Overview of the Package Initialization Process

In this section we summarize the steps involved in the package registration and instantiation process. The chronological sequence is a follows.

1. At global initialization time, prior to the start of the main program, dummy package registration variables are instantiated. Their constructors run.

2. The package registration variables constructors register themselves with the package factory.

3. The main framework program starts. No packages have been instantiated yet.

4. The framework program controls the instantiation of packages. It tells the package factory which packages to instantiate by text string name, and which RCP parameters to use. Initialization of packages happens when the are instantiated. There is no package initialization hook.

5. As each package is instantiated, the constructors of the hook interface base classes register themselves with the hook registry classes.

6. After the instantiation of packages is complete, the framework program gets lists of packages to be called for each hook.

7. The framework program calls the hooks.

7 How to Create a New Package

This section explains what is required of a user who wants to create a new package. Basically, the user is required to define a new class for his package. The user must include the following elements.

1. The package class must derive from one or more hook classes.
2. The implementation file for the class should define a static dummy global variable of the appropriate package registration type.
3. The package class must include a static method to that returns the string name of the class.
4. The package must implement methods required by the hook interfaces.

8 Sample Code

8.1 A User Package

```
static PackageRegistration<MyClass> dummy;
class MyClass : public ProcessEvent,
                public DumpEvent,
                public JobSummary
{
public:
  // Constructor
  MyClass(RCP* param);

  // Package name.
  static string package_name { return string("MyClass");}

  // Required hook overrides
  virtual bool process_event(Event& event);
  virtual bool dump_event(Event& event);
  virtual bool job_summary();
};
```

8.2 Package Registration Class

```
template <class T>
class PackageRegistration<T> : public PackageRegistrationBase
{
public:

  // Default constructor
```

```
  PackageRegistration()
  {
    PackageFactory* p = PackageFactory::instance();
    p->register(this);
  }

  // Get package name
  virtual static string package_name { return T::package_name(); }

  // Instantiate package.
  virtual HookBase* create_package(RCP* param) { return new T(param); }
};
```

8.3 Package Factory

```
class PackageFactory
{
public:

  // Register package
  void register(PackageRegistrationBase* package)
  {
    packages.push_back(package);
  }

  // Find the specified package.
  PackageRegistrationBase* find(string name);

  // Create package
  HookBase* create_package(string name, RCP* param)
  {
    return find(name)->create_package(param);
  }
private:
  list<PackageRegistrationBase*> packages;
};
```

8.4 Hook Class

```
class ProcessEvent : public HookBase
{
public:
  // Default Constructor
  ProcessEvent()
  {
```

```
    HookRegistry<ProcessEvent>* p =
      HookRegistry<ProcessEvent>::instance();
    p->register(this);
  }

  // Pure virtual interface method.
  virtual bool process_event(Event& event) = 0;
}
```

8.5 Hook Registry

```
template <class T>
class HookRegistry<T>
{
public:
  // Register method
  void register(T*) { packages.push_back(T*); }

  // Return packages for this hook.
  list<T*> get_packages() { return packages; }

private:
  list<T*> packages;
};
```