HOOPS Developers Guide

Version 0.9

HEASARC
Code 662
Goddard Space Flight Center
Greenbelt, MD 20771
USA

April 2003

Contents

1 Overview of HOOPS Capabilities **3**

1.1 Introduction 3

1.1.1 HOOPS's relationship to ISDC's PIL 3

1.2 Summary of Class Families 4

1.2.1 The ParPrompt Family 4

1.2.2 The ParFile Family 4

1.2.3 The ParGroup Family 5

1.2.4 The Par Family 5

1.2.5 The BiDirItor Family 5

1.2.6 The Prim Family 6

1.3 The ParPrompt Family 6

1.4 The ParFile Family 7

Chapter 1

Overview of HOOPS Capabilities

1.1 Introduction

HOOPS is an object-oriented library for the development of user interfaces and high-level software control structures. To the user it behaves similarly to most IRAF-inspired parameter systems in use today in astronomical software, such as HEASARC's Xanadu Parameter Interface (XPI) and ISDC's Parameter Interface Library (PIL). To the developer, HOOPS is intended to provide convenient, flexible, extensible and powerful access to IRAF-style parameter concepts, both in the context of traditional parameter files and in a more general way. In particular, it is hoped that new missions will use HOOPS to put the tried and true IRAF parameter paradigm to new and novel uses.

1.1.1 HOOPS's relationship to ISDC's PIL

HOOPS was originally conceived as an object-oriented extension to PIL, to be used by GLAST mission-specific software. Its current implementation uses PIL internally for file access, command-line parsing, and user interaction. However, it is intended eventually to remove the dependency on PIL, for three reasons. First, it is always easier to maintain a single software component than to maintain two fairly tightly coupled components. Second, PIL has (at the time of this writing) some dependencies on Unix system libraries, and third, the usage of PIL forces some unneccessary limitations as well as some complexity and inefficiency into the HOOPS implementation.

In anticipation of the eventual replacement of PIL by native HOOPS methods, PIL usage is restricted to only two classes, PILParFile and PILParPrompt. New classes which use PIL a bit differently could be derived from these classes. New interfaces for file access or prompting which do not use PIL at all should instead be derived directly from the base classes IParFile and IParPrompt, respectively.

1.2 Summary of Class Families

HOOPS is divided into five main class hierarchies. To convey the overall design concept, the classes will be presented in descending order from the highest level to the lowest level of abstraction. The detailed descriptions of each class may be read in any order, however.

- ParPrompt: mechanisms for obtaining input from the user
- ParFile: a group of parameters associated with a file
- ParGroup: a collection of parameters
- Par: encapsulation of a single parameter
- GenBiDirItor: a universal bidirectional iterator
- Prim: support classes to handle type conversions

All of these classes have virtual methods as necessary to support polymorphic behavior, to facilitate extension of class behavior through derivation. Only the declarations for pure virtual classes and concrete factory classes are considered public.

1.2.1 The ParPrompt Family

The abstract base class (ABC) of the ParPrompt family is IParPrompt, which defines a standard interface for user prompts. Presently included in the hierarchy are two concrete derived classes, ParPrompt and PILParPrompt. The first of these, ParPrompt, is only partially implemented at present, but it will eventually provide a complete implementation of the IParPrompt interface. The second class, PILParPrompt, is a complete implementation of the IParPrompt interface which uses PIL. Due to some limitations in PIL, the PILParPrompt interface can only function in the context of a set of parameters associated with a parameter file. When it is complete, the ParPrompt interface will be more general, and allow prompting for any group of parameters.

Also defined is an abstract factory, or "virtual constructor" class, IParPromptFactory. At present its only concrete subclass is PILParPromptFactory, which may be used to create instances of PILParPrompt.

1.2.2 The ParFile Family

The ABC of the ParFile family is IParFile, which defines a standard interface for parameter file access. Presently included in the hierarchy are two concrete derived classes, ParFile and PILParFile. The former is only partially implemented at present, but it will eventually provide a complete native implementation of the IParFile interface which will not require PIL. The second class, PILParFile, is a complete implementation of the IParFile interface which uses PIL.

Also defined is an abstract factory, or "virtual constructor" class, IParFileFactory. At present its only concrete subclass is PILParFileFactory, which may be used to create instances of PILParFile.

1.2.3 The ParGroup Family

The ABC of the ParGroup family is IParGroup, which encapsulates the concept of a group of parameters, and defines a standard interface for accessing parameters contained within the group. Presently only one derived class exists, ParGroup, which provides a simple but complete implementation.

Both the IParPrompt and IParFile interfaces can produce references to an IParGroup-derived object containing the parameters for which they are prompting and providing file access, respectively. This is so that software which uses HOOPS can be easily separated into three parts: drivers which prompt the user and determine what course the software should take, file I/O sections, and components which simply use the parameters. Most code (i.e. general library code) will fall in the latter category, and it can and should use IParGroup-derived objects only, and leave the other tasks to higher-level, more specialized components.

Also defined is an abstract factory, or "virtual constructor" class, IParGroupFactory. At present no concrete factories exist, so it is not possible to instantiate explicitly a ParGroup object except internally to HOOPS.

1.2.4 The Par Family

The ABC for the Par family is IPar, which encapsulates the concept of a single parameter, and defines a standard interface for accessing that parameter. Presently only one derived class exists, Par, which provides a complete implementation using the Prim class family to perform type conversions.

The IPar interface contains methods to access the eight kinds of information found in an IRAF-compliant parameter: Name, Type, Mode, Value, Minimum value, Maximum value, Prompt string and Comment. In addition there are overloaded methods to perform explicit conversions to and from C++ primitive types and strings, as well as assignment and conversion (cast) operators to perform such conversions implicitly.

Also defined is an abstract factory, or "virtual constructor" class, IParFactory. One concrete factory, ParFactory, exists for the creation of Par objects.

1.2.5 The BiDirItor Family

The purpose of this hierarchy is to provide an iterator type which may be used to iterate over collections of parameters. In general, templated iterator classes (from STL and elsewhere) have common features, but do not have common base classes. Moreover they are not generally polymorphic. The BiDirItor family provides a means of wrapping any class which behaves like an iterator inside a polymorphic class deriving from a common base. This iterator may then be used to iterate over a collection of any object type.

The BiDirItor family is a templated hierarchy, whose ABC is IBiDirItor. This provides a standard interface for a bidirectional iterator, with methods to provide the standard operations -¿, unary *, ++, –, ==, !=. This is subclassed in the templated BiDirItor class, which wraps a specific iterator type, as indicated in one of its template arguments. Finally, GenBiDirItor, which also derives from

IBiDirItor, contains a pointer to a IBiDirItor. Thus GenBiDirItor provides a single iterator which may be used to iterate over any underlying iterator type.

1.2.6 The Prim Family

The Prim family currently consists of the classes IPrim and Prim. The former is an ABC defining an interface of overloaded abstract To() and From() methods to convert the (unspecified) type represented by IPrim to and from C++ primitive types and strings.

The templated class Prim, derives from IPrim. It contains a data member of the type given as its template argument, and provides an implementation of the Prim interface for that type. For example, Prim<int> is a class which knows how to convert its int member to and from other types, throwing exceptions when such a conversion results or may result in a loss of precision or accuracy of the converted value.

Because Prim defines conversions to and from ALL C++ primitive types and strings, a fair number of these conversions may result in a loss of precision or accuracy of a converted value. For example, if a Prim<int> object is converted to a floating point value, a loss of precision will occur. In this situation, Prim<int> will perform the conversion, but then throw an exception. Similarly, conversions which result in overflows, underflows, conversions to a (potentially) smaller type, and conversions between signed and unsigned integral types will all throw exceptions. Whenever possible a sensible conversion will be performed before the exception is thrown, allowing calling code easy recovery if it wishes to ignore the error.

Although developed to meet the needs of the Par family, the Prim family was developed separately in the hopes that it may be reused completely outside the context of parameter objects. Because it is a completely general templated class, it may be used any time one wishes to convert between different types with a measure of control over what types of conversions are considered valid.

An abstract factory class IPrimFactory exists for IPrim objects. At present there is one concrete factory, PrimFactory, which creates Prim objects.

1.3 The ParPrompt Family

The abstract interface defined by the IParPrompt class declares a number of methods to handle prompting. In the standard IRAF modality, parameters may be set on the command line, in which case the prompts are suppressed. Hence this interface also contains methods for handling command line argument arrays. Of course, the arguments passed need not necessarily come literally from something a user typed, but could be explicitly constructed strings. Its public methods are:

- IParPrompt(); deletes all resources associated with the prompt object, including the underlying parameter group.

- IParPrompt & operator =(const IParPrompt & p); deletes all resources associated with the destination prompt object before copying the source object.

- IParPrompt & Prompt(); prompt for all parameters associated with this IParPrompt object. Returns the object.
- IParPrompt & Prompt(const std::string & pname); prompt only for the named parameter. Returns the object.
- IParPrompt & Prompt(const std::vector¡std::string¿ & pnames); prompts for the collection of parameters named in the argument. Returns the object.
- int Argc() const; returns the number of command line arguments known to this object.
- char ** const Argv() const; returns a copy of the command line arguments known to this object.
- IParGroup & Group(); returns the group of parameter objects which is present in this prompt object. Note that the reference which is returned is modifiable.
- const IParGroup & Group() const; returns the group of parameter objects which is present in this prompt object. Note that the reference which is returned is not modifiable.
- IParPrompt & SetArgc(int argc); set the number of command line arguments contained in this prompt object.
- IParPrompt & SetArgv(char ** argv); set the command line arguments contained in this prompt object. The values of the array of strings passed in are copied.
- IParGroup * SetGroup(IParGroup * group); set the underlying group of parameters to the group given in the argument. Returns the previous group. Note that when the prompt object is deleted, it automatically deletes its internal group. In other words, if SetGroup is used to set the group used by the prompt object, that group is henceforth "owned" by the prompt object.

In addition, there is an IParPromptFactory abstract class with one concrete subclass, PILParPrompt-Factory.

1.4 The ParFile Family

The abstract interface defined by the IParFile class declares a number of methods to write and read parameters to and from parameter files. It is anticipated new file types can be supported by subclassing IParFile. Its public methods are:

- void Load(); clear the current group of parameters and load the parameters from the currently selected parameter file.
- void Save() const; save the current group of parameters to the currently selected parameter file.

- const std::string & Component() const; returns the name of the current component, which is used to find the parameter file.
- IParGroup & Group(); returns the underlying group of parameters. Note that the reference returned is modifiable.
- const IParGroup & Group() const; returns the underlying group of parameters. Note that the reference returned is not modifiable.
- IParFile & SetComponent(const std::string & comp); set the name of the current component. This will be used to determine the parameter file name.
- IParGroup * SetGroup(IParGroup * group); set the underlying group of parameters to the group given in the argument. Returns the previous group. Note that when the prompt object is deleted, it automatically deletes its internal group. In other words, if SetGroup is used to set the group used by the prompt object, that group is henceforth "owned" by the prompt object.
- GenParItor begin(); returns an iterator pointing to the first (modifiable) parameter object in the group.
- ConstGenParItor begin() const; returns an iterator pointing to the first (non-modifiable) parameter object in the group.
- GenParItor end(); returns an iterator pointing to one item past the last (modifiable) item in the group.
- ConstGenParItor end() const; returns an iterator pointing to one item past the last (non-modifiable) item in the group.

In addition, there is an IParFileFactory abstract class with one concrete subclass, PILParFileFactory.