HDF5 Performance Testing Library

1. Introduction

The HDF5 performance testing library provides HDF5 users the required APIs to

- Measure the performance of their routines
- Store, retrieve, update, and remove the performance information into external storage system such as file and database
- Utility functions to interact with command line input

The library is implemented using C++, but C API is also provided. In the following sections, we'll explain how the library works.

2. Performance Testing Information

The Figure 1 at the end of this section depicts the class diagram for performance testing information. In library, performance testing information has three levels. The first level is the program level containing information about a program that uses HDF5 library. This level is represented by **TestRoutine.** At the second level, there are functions of a program that uses HDF5 library. This level is represented by **TestAction.** Obviously**,** each **TestRoutine** must have one or more **TestAction.** The next level belongs to each execution of a function. Clearly, each function in different execution due to different settings could have different results. This level is implemented by **TestInstance**. The same relationship between **TestRoutine** and **TestAction** goes for **TestAction** and **TestInstance.**

2.1 Base Class PO

Since all the performance testing classes are persistent and could be stored in database, they all subclass **PO** which provides the basic methods for persistent classes. These methods set and get the unique Id of an object:

Class PO

int getID();
void setID(int id);

2.2 Derived Class Env and EnvOwner

The other common property among all the testing information classes is that they all could have arbitrary number of name, value pairs or settings. Usually these are information about a program, function, or even each execution of a function that are in the form of name, value pair. They are very flexible in terms of type and number. For instance, the operating system program is running under the hardware platform, and so on and so forth. These *input and environmental settings* for the programSetting is implemented by **Env** class:

Class Env

**void set(char* name, char* value);
char* get(char* name);
bool remove(char* name);
bool empty();**

The class **Env** implements the required method to add, update, and remove a setting and could have any number of settings.

Due to the common property mentioned above, all testing information classes subclass from class **EnvOwner** which has a 1-to-1 association relationship with **Env** (aggregates one **Env**). The **EnvOwner** provides the methods to access its aggregated **Env** object:

Class EnvOwner

**void setSetting (char* name, char* value);
char* getSetting(char* name);
bool removeSetting(char* name);**

2.3 Derived Class TestRoutine

The root object for performance testing information is **TestRoutine. TestRoutine** represents a user's program that uses HDF5 library. It includes the information about the testing program including *name, description,* and *version* of the program. Developer could changes these values using, for example, *setName()* and *getName()* for changing the name field. Each **TestRoutine** object must have unique name.

TestRoutine object has a collection of different **TestAction**s. A **TestAction** could be observed as a function or a meaningful code fragment in a test program or routine. Therefore, **TestRoutine** has the required methods to add, remove, or get its TestAction objects. Each **TestAction** object of a **TestRoutie** object must have a unique name.

Class TestRoutine: public EnvOwner, public PO

void setName(char* name);
char* getName();
void setDescription(char* desc);
char* getDescription();
void setVersion();
char* getVerson();
 	
//Methods to access and/or manipulate TestAction members
void addAction(TestAction& action);
void removeAction(vector<TestAction>::iterator pos);
TestAction* getAction(char* actionName);
TestAction* getAction(vector<TestAction>::iterator pos);
vector<TestAction>::iterator beginActions();
vector<TestAction>::iterator endActions();

2.4 Derived Class TestAction

TestAction class has the same general functions as **TestRoutine** to set and/or get its name and description. As **TestRoutine, TestAction** contains one or more **TestInstance** objects. **TestInstance** class represents each execution of a **TestAction. TestAction** provides necessary methods to access or manipulate its **TestInstance** objects.

Class TestAction: public EnvOwner, public PO

```
void setName(char* name);
char* getName();
void setDescription(char* desc);
char* getDescription();

//Methods to access and manipulate TestInstance members
Void addInstance(TestInstance& instance);
Void removeInstance(vector<TestInstance>::iterator pos);
TestInstance* getInstance(char* instanceName);
TestInstance *getInstance(vector<TestInstance>::iterator pos);
vector<TestInstance>::iterator beginInstance();
vector<TestInstance>::iterator endInstanes();
```

2.5 Derived Class TestInstance

Since **TestInstance** depicts an execution of a **TestAction,** it has related properties: name of the dataset used, the dataset optional description, date, host or machine **TestAction** was executed on, the version of the HDF5 library used in the run, and the result. The date of execution makes a **TestInstance** object among other objects in the same **TestAction** unique**.**

```
Class TestInstance: public EnvOwner, public PO
```

```
void setDatasetName(char* name);
char* getDatasetName();
void setDate(Date date);
Date getDate();
void setLibVersion(char* libVersion);
char* getLibVersion();
void setHost(char* host);
char* getHost();
void setDatasetDescription(const char* datasetDescription);
char* getDatasetDescription();
void setResult(double result);
double getResult();
```

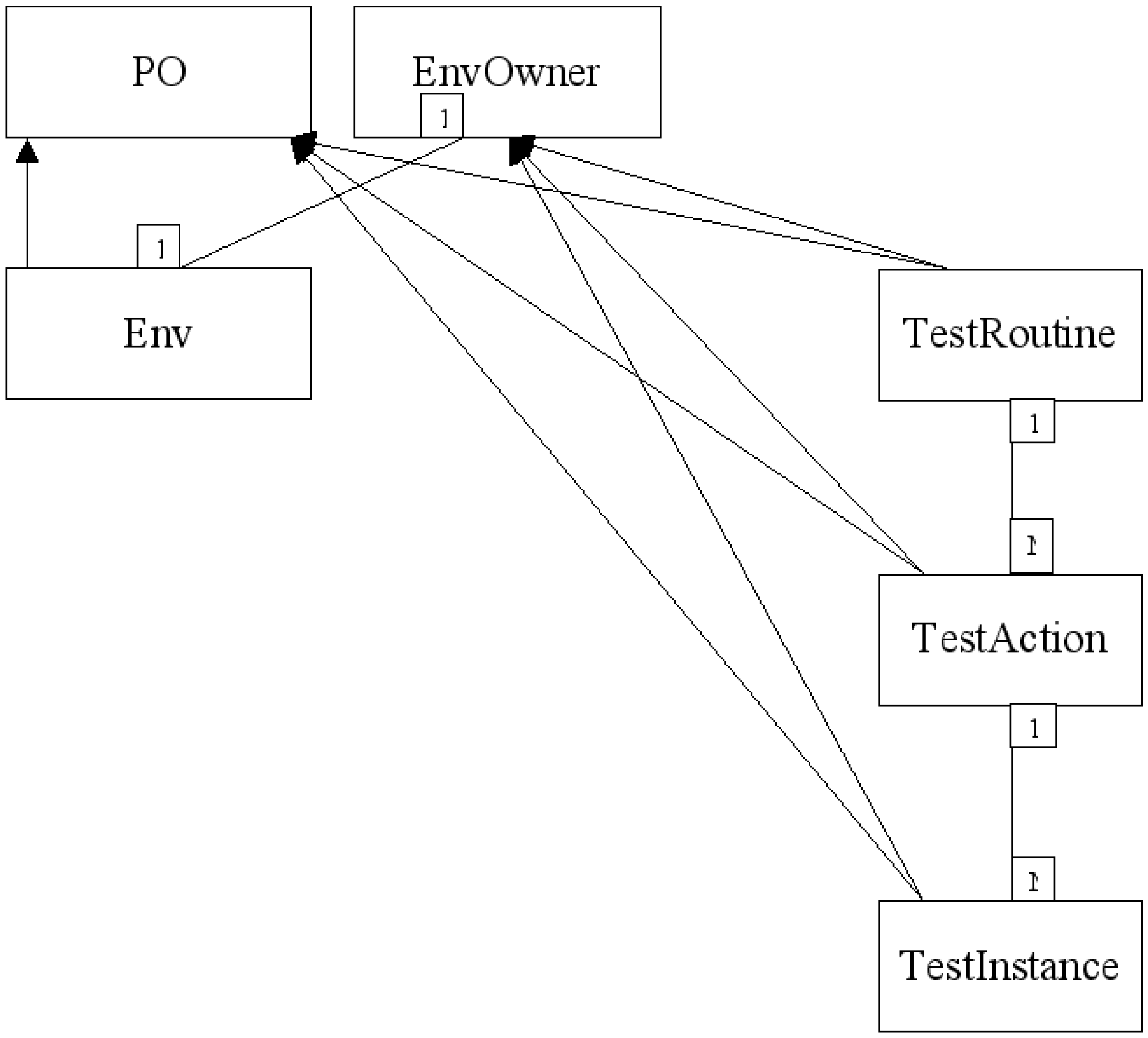


Figure1. The class relationship diagram for test information classes

3. Performance Information Storage

In order to store the above objects to the external storage and getting different reports afterward, **Storage** class has been implemented. This abstract class encapsulates the notion of storage by its basic methods.

Class Storage
virtual void open(char* server, char* name, bool append=false, char* user=NULL, char* passwd, unsigned int port=0);
virtual bool isOpen();
virtual void write(TestRoutine* routine);
virtual void close();

The write method implementation in its concrete subclasses must write the **TestRoutine** object as well as its aggregated objects like **TestAction** to the external storage. The concrete implementation of **Storage** is **FileStorage** which stores the information to a single file on a local file system. Its open method ignores the user, password, and port parameters.

Class FileStorage: public Storage
virtual void open(char* path, char* name, bool append=false, char* user=NULL, char* passwd, unsigned int port=0);
virtual bool isOpen();
virtual void write(TestRoutine* routine);
virtual void close();

Random access storage systems like relational database systems have more functionality. The abstract class **RandomAccessStorage** which is in turn a subclass of **Storage** class, adds more methods to the **Storage** class for finding, updating, and removing the objects from random access storages.

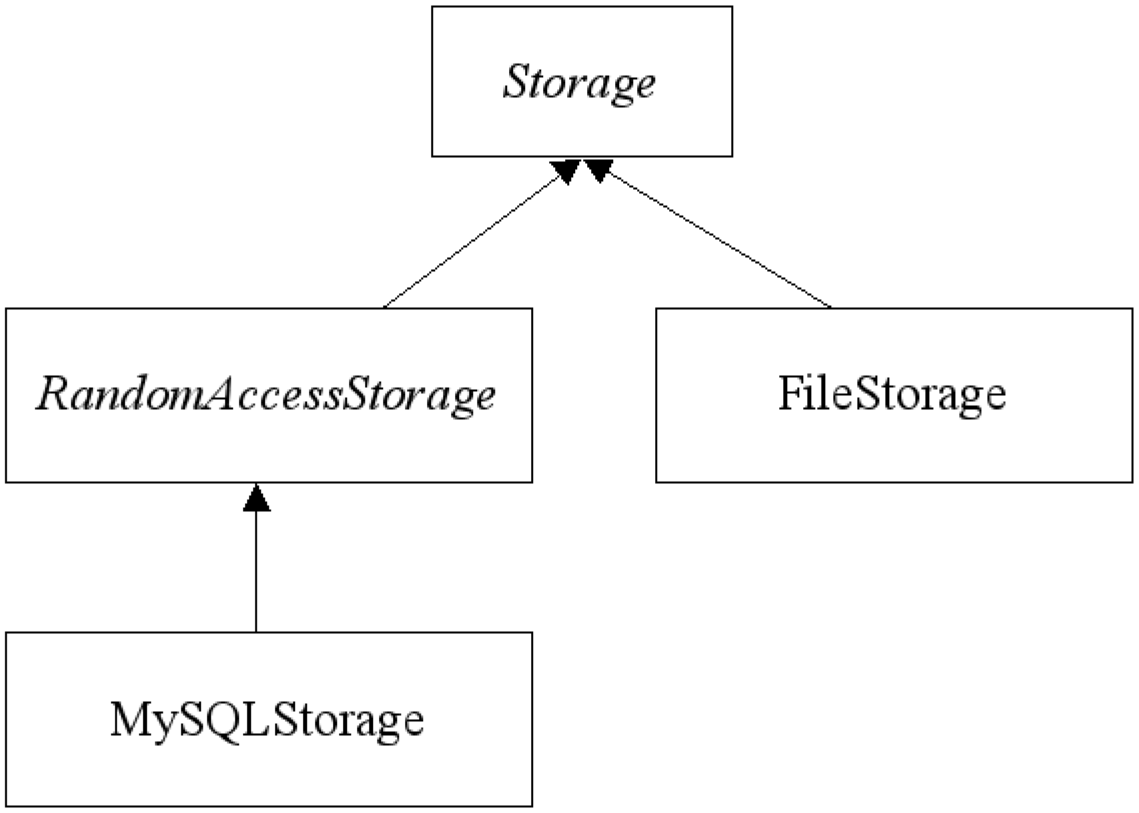
Class RandomAccessStorage: public Storage
virtual TestRoutine* findByName(char* routineName);
virtual bool update(TestRoutine* routine);
virtual void remove(char* routineName);

A concrete subclass of this class has been implemented for MySQL database system, named **MySQLStorage.**

Class MySQLStorage: public RandomAccessStorage
virtual TestRoutine* findByName(char* routineName);
virtual bool update(TestRoutine* routine);
virtual void remove(char* routineName);

Choosing the right storage class would be hard for the API user due to the expected increase of the number of storage system types that the library would support. Therefore, **StorageManager** class encapsulates all the classes above and exposes unified methods to the user. In this way, user does not need to know anything about different storage handling classes.

Class StorageManager
Void open(char* server, char* name, bool append=false, char* user=NULL, char* passwd, unsigned int port=0);
bool isOpen();
void write(TestRoutine* routine);
void close();
TestRoutine* findByName(char* routineName);
bool update(TestRoutine* routine);
void remove(char* routineName);


Storage
RandomAccessStorage
FileStorage
MySQLStorage

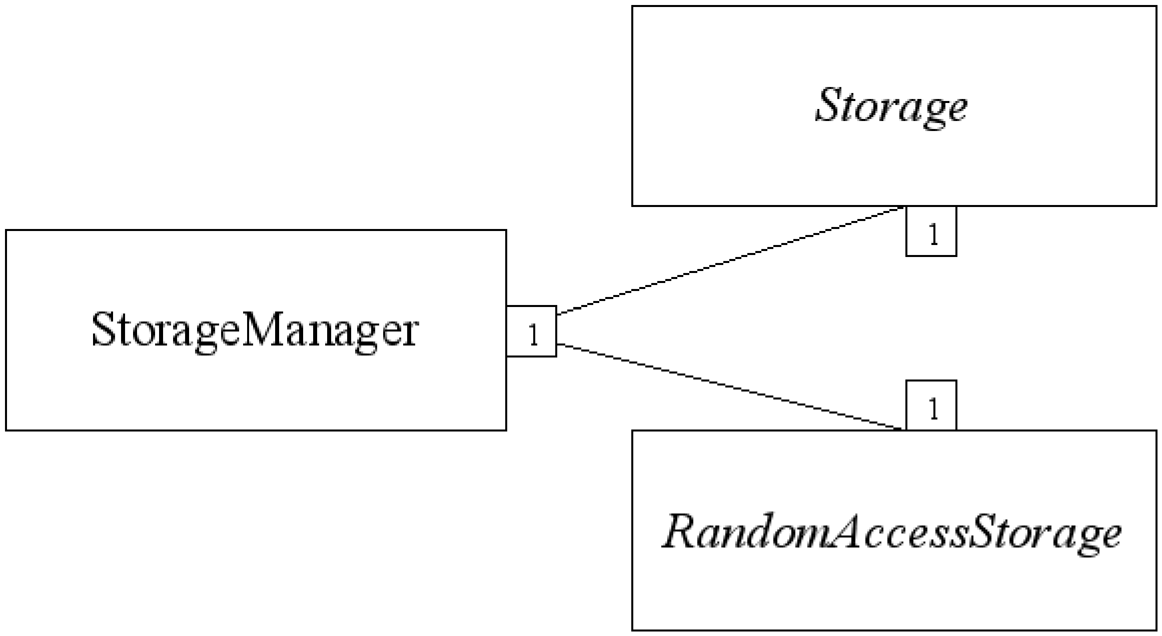

Storage
1
StorageManager
1
1
RandomAccessStorage

Figure2. The class relationship diagram for the storage classes

4. Performance Measurement

The library also provides some utility functions to measure time spent on an operation. The **TestUtil** class implements the following methods.

Class TestUtil

static void startTimer(timeval* start);
static double endTimer(timeval start);
static double random(long int limit);

In order to measure the spent time for an operation, user could call **startTimer** method by passing a standard *timval* structure described in **sys/time.h** header file. After finishing the operation, a call to the **endTimer** method with the same argument, returns the time spent on the operation. The method **random** returns a random number within the [0,limit).