A Java application framework for scientific software development

D. Glez-Peña, H. López-Fernández, M. Reboiro-Jato, J.R. Méndez, F. Fdez-Riverola*

Dep. Computer Science, University of Vigo, ESEI, Campus As Lagoas, 32004 Ourense, Spain

Summary. This paper presents AIbench, a Java desktop application framework mainly focused on scientific software development, with the goal of improving the productivity of research groups. Following the MVC design pattern, the programmer is able to develop applications using only three types of concepts: operations, data-types and views. The framework provides the rest of the functionality present in typical scientific applications, including user parameter requests, logging facilities, multi-threading execution, experiment repeatability and graphic user interface generation, among others. The proposed framework is implemented following a plugin-based architecture which also allows assembling new applications by the re-use of modules from past development projects.

Key Words. Java application framework; Reusable components; Component-based Software development

* Corresponding author. Tel.: +34 988 387015; Fax: +34 988 387001
E-mail address: riverola@uvigo.es
URL: http://sing.ei.uvigo.es/
1. Introduction

Home-made software applications are often valuable resources for many research groups in the scientific community. Although the most popular applications are designed for end-users, some with commercial purposes, there is a large number of software projects that are developed only for internal department usage. These applications (typically console-based) are intended to be used by the research group members or by a restricted set of researchers, being mainly focused on collecting the user parameters, executing a given computational process and analysing the obtained results.

In the scientific community, the most active research areas offer also a growing number of freely available software tools, packages, or programming libraries implementing new advances in the distinct fields. Examples of these scenarios are Bioinformatics, with hundreds of programs and web applications, popular machine learning environments [1, 2], natural language processing packages [3], among a plethora of other examples. These utilities are intended to be used, embedded and improved by other programmers inside their own projects, easing the creation of custom applications. In this context, Java has been widely adopted in the scientific community due to the huge amount of freely available APIs and open source scientific developments, regardless of its other benefits such as object-orientation, language inter-operability, cross-platform applicability, built-in support for multi-threading, networking, etc.

Developing applications in a scientific context presents a large number of specific requisites, like the extensive use of logging messages to monitor the progress of long processes, setting values for a high and variable number of parameters before running the experiments, the ability to repeat the same workflow but changing a few parameters or input data, taking the maximum advantage of multi-threading, or the need to integrate third party (or previously developed) software. These tasks represent only some examples of the broad and particular nature of scientific research software applications.

Nevertheless, writing software is not normally the main focus of a research group, where the software design skills or production capacity are typically not comparable to those of a development team in a dedicated software company. The development of applications meeting the previously mentioned requisites is not easy. Furthermore, writing more elaborate software including sophisticated graphic user interfaces (GUIs) brings additional advantages, like the possibility of a friendly application being publicly available and useable by other researchers. This fact surely increases the group prestige, but it requires too much experience and development efforts and thus it is usually not taken into consideration. However, and regardless of whether a given application is more or less elaborate, the real benefits come when a development team is able to reuse its own code, which is not a trivial issue.

Every application has repetitive (or shared) functionalities dealing with user interaction, application context handling, logging and results visualization, which are not problem specific functionalities. Moreover, if a GUI is required, the time invested and the difficulty of developing these aspects increase significantly. It may happen that, when the core algorithms are already implemented or integrated from external packages, it takes too much additional time to start using the application due to the fact that these necessary elements are not yet available. The reuse of this code is, therefore, essential. Furthermore, data structures, algorithms, input/output routines, small custom visualization components and other problem-specific pieces of code should also be easy to reuse when needed. However, frequent bad design decisions like merging these two kinds of code (shared and specific functionalities), produce applications which are hard to subsequently maintain and/or reuse.
Therefore, it should be possible (but not trivial) to increase the code reusability and consequently the overall productivity. A typical approach to cope with this kind of problem is to make use of an application framework, which can be seen as a semi-finished application and a reusable architecture design [4, 5]. By the use of such a solution, the common functionality between scientific projects is delivered by the framework and is consequently isolated from the application specific code, allowing the programmers to spend more of their time in developing the real application requirements, rather than in low level details. Such a separation also enhances the code reuse, not only of the proper framework services, but also of a tested, independently evolving design, and of a more coherent problem-related behaviour. In other words, a framework can be seen as the skeleton of an application that can be customized by a programmer [6].

In recent years, frameworks have become more popular, especially in Web development [7-11], where Apache Struts is one of the most successful examples. Nowadays, there are frameworks for almost every kind of software including object-oriented desktop applications [12-14], software testing [15], compiler generation [16, 17], multimedia [18-20] and middleware [21, 22]. More focused domain specific frameworks can be also found for different scientific fields such as bioinformatics [23, 24], simulation and visualization [25], content-based image retrieval [26], computational optimization algorithms [27] or fuzzy systems [28].

However, to the best of our knowledge there is no object-oriented framework that, while including the characteristics of modern general application frameworks (GUI generation facilities, pluggable architecture, non-intrusive design, update management, scripting facilities, etc.) is specifically focused on the development of scientific applications by providing an experiment-oriented architecture, a clear steps definition, intuitive data management, experiment repeatability, multi-threading execution and logging.

In the past years, we have developed successful applications which integrate and apply several Artificial Intelligence techniques to different kinds of problems [29]. In this context, our geneCBR system [30], a previous Java desktop application in the Bioinformatics field, can be considered as the seed of the present work. After several cycles of expansion-consolidation [31], the user interaction based on input-process-output cycles, the generated objects tree, the main window layout, and some other ideas were identified as abstract reusable functionalities to be included in an emerging framework for our research group.

Built upon our previous experience [32], this paper goes into AIBench, an open-source Java desktop application framework, specifically intended to improve both the quality and the productivity in the development of home-made applications inside research groups. Such a framework aims at the rapid production of high quality application prototypes with minimum effort on problem-unrelated functionalities and with the maximum level of reusability between previously coded algorithms. We believe that when the core algorithms for solving a problem are available, it should be possible to deliver, almost immediately, an acceptable application prototype (including a graphical user interface) without affecting the code of the domain specific routines. This prototype may subsequently evolve to a real final application by the use of more advanced capabilities of AIBench.

In order to accomplish the final goal of covering the gap between general desktop application frameworks and domain frameworks by taking into account the specific requirements of scientific software development, AIBench implements the IPO (input-process-output) model by means of the MVC (model-view-controller) design pattern and makes use of several state of the art technologies. In this sense, AIBench incorporates a
plugin-engine, a Java scripting platform and takes advantage of Java 1.5 annotations, reflection, XML and various design principles in order to make the framework easy to use, lightweight and non-intrusive.

The rest of the paper is structured as follows. Section 2 introduces the basic design concepts needed to understand and use the proposed framework. Section 3 discusses the main capabilities and services provided by AIBench. Section 4 looks into the usage details and presents a case study. Finally, Section 5 summarizes the main conclusions and identifies future work.

2. Essential design concepts and principles
AIBench was specially designed for applications based on the input-process-output model, in which the output of one task can be the input of another process. For example, one application could first load some sample data from a file taking the path as input, then execute a loading procedure and finally, generate an in-memory data representation as output. After that procedure, its output may be forwarded to the input of another process, say an analysis or simulation algorithm, producing a new in-memory representation of its results as output, and so on. In addition to this main cycle, all outputs could be rendered and displayed to the final user at any time after being produced. Figure 1 shows this application concept.

![Figure 1. IPO model representation of a hypothetical research application.](image)

The IPO model is not new but is particularly well suited to the scientific domain. In this sense, this model is often the basis of Workflow Management Systems (WMS), which are frameworks focused at the design and execution of custom workflows, composed with a set of previously developed basic operations. Examples of successful WMS are RapidMiner [1] in the Machine Learning area, which is able to interconnect data input/output operations, as well as model induction, application and validation, or Taverna [33] a WMS focused on Bioinformatics web services orchestration. Generally speaking, WMS follow the IPO model by establishing a contract between the system and the end-user, who is responsible for designing his custom workflow, being aware of IPO philosophy. Along the same lines, AIBench follows the IPO model, but it is more focused on the contract between the applications programmer and the framework itself. AIBench applications are designed by the programmer following IPO,
but the end-user is provided with a product where this paradigm is not as explicit as in WMS. AIBench applications execute the developed processes as user requests to them (there is no workflow design phase). With AIBench acting as the glue between each process invocation, the programmer needs to focus only on the internals of the processes, data and viewers.

The AIBench IPO model implementation is based on the MVC design pattern, which promotes a separation between the user interface and the domain logic, being successfully applied to desktop applications. In this context, we have mapped the IPO model with the MVC concepts and implemented them as three different elements in AIBench: operations, data-types and views. In addition, the full MVC architecture is completed by making the framework responsible for retrieving the input parameters, invoking the requested process, collecting the results and instantiating suitable views for them. In this sense, AIBench acts as the MVC controller and, partially, as a view. Table I shows the mapping between the IPO, MVC and AIBench concepts.

<table>
<thead>
<tr>
<th>IPO</th>
<th>MVC</th>
<th>AIBench</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>View (forms)</td>
<td>AIBench workbench</td>
</tr>
<tr>
<td></td>
<td>Model (parameters)</td>
<td>data-types</td>
</tr>
<tr>
<td>Process</td>
<td>Controller</td>
<td>AIBench core</td>
</tr>
<tr>
<td></td>
<td>Model (behaviour)</td>
<td>Operations</td>
</tr>
<tr>
<td>Output</td>
<td>View (results display)</td>
<td>Views</td>
</tr>
<tr>
<td></td>
<td>Model (results)</td>
<td>Data-types</td>
</tr>
</tbody>
</table>

Moreover, there are additional properties driving different design and implementation decisions in the proposed framework:

- **Repeatability.** All processes executed inside the framework during a session should be repeatable in the future, in the same order and without the interaction of the user. This is a very valuable requisite for scientific applications allowing the user to create execution macros, which could be applied with other input data in the future.

- **Problem independence.** The framework should not contain any concept related to a specific research field, say Physics, Maths, Data-mining, Vision, etc. since AIBench is a more general, domain-independent framework.

- **Non intrusive.** Although AIBench can be classified as a white-box framework [34] (which require the programmer to inherit or implement specific classes or interfaces), the application specific code (algorithms, data structures and viewers) should contain the minimum amount of framework-related code. This allows the programmer to keep clean his own application core functionality and even reuse it in other environments or applications (web, a more detailed GUI, etc.). In order to accomplish this goal, the inheritance is discarded and a more declarative method is preferred. In this sense, AIBench makes extensive use of XML and Java 1.5 Annotations.

- **Smart defaults.** In order to allow the programmer to run his software as soon as possible, the framework should require a minimum configuration to execute the application. Whenever possible, the framework makes a decision in the absence of a custom value for any parameter.

- **No necessity to reinvent the wheel.** If possible, standard Java techniques are preferred rather than introducing more framework concepts. A built-in output renderer based on the standard toString, a flexible default object saver based on Java Serialization and an output console displaying the standard log4j [35] messages coming
from the programming code, are some examples demonstrating this principle. These characteristics also improve the non-intrusive and the smart-defaults aspects of the framework.

Next, the three AIBench key concepts are introduced and briefly explained (a more advanced discussion is described in Section 4).

2.1. Operations
Operations define high-level problem-oriented processes. Each operation is implemented through only one Java class (which can delegate its internal behaviour to other classes). This idea is based on the Command design pattern [31], but also adding meta-data information to make the framework aware of its complete interface. Generally speaking, one operation is a unit of logic with a well-defined input and output specified via a set of ports. A port is a point where some data can enter into the operation or can be generated as output. For each port, a method of the class is associated with it. There are three types of ports: IN (for input data, where the associated method must have one parameter of the type of the incoming data), OUT (for output data, where the associated method must not have any parameter and its return type must be of the type of the output data) and IN-OUT (for both input and output data, where the method must have one parameter for the input data and a return type for the output data). The data type of the port can be any Java class, as conceptualized in Figure 2.

Every time an operation is executed, one instance of its class is allocated and all the methods associated with the ports are invoked in a predefined order, with the parameters already retrieved from the user. The framework is responsible for providing the input values, in the case of IN (or IN-OUT) ports, and to gather the output values, in the case of OUT (or IN-OUT) ports. To specify which methods of a given class are ports, how they are ordered and some other details, the class should be annotated with a set of predefined annotations. Figure 2b shows the source code of two basic AIBench operations (i.e., a simple addition procedure and a factorial calculation) containing the three available port types. To describe the whole

```java
@Operation(description="this operation adds two numbers")
public class Sum {
    private int x, y;
    @Port(direction=Direction.INPUT, name="x param")
    public void setX(int x) {
        this.x = x;
    }
    @Port(direction=Direction.INPUT, name="y param")
    public void setY(int y) {
        this.y = y;
    }
    @Port(direction=Direction.OUTPUT)
    public int sum() {
        return this.x + this.y;
    }
}

@Operation(description="this operation calculates the factorial of a given number")
public class Factorial {
    @Port(direction=Direction.BOTH, name="value")
    public double factorial(double value) {
        if (value < 50)
            return 1;
        else
            return value * factorial(value-1);
    }
}
```

Figure 2. (a) AIBench operation model using ports and (b) sample code of two example operations.
operation, both the @Operation and the @Port annotations are included near to their associated entities. These annotations are covered more in detail in Section 4.

2.2. Data-types
AIBench data-types are intended to support problem specific data structures (in-memory representation of some stored data, models, results, etc.). Although they are as important as operations, AIBench data-types are very simple to implement. Any Java class can be a data-type without additional code, only the fact of being the input or output type of some operation is required to be considered as an AIBench data-type. However, in order to support repeatability of experiments, non-primitive data-type instances must be previously created by operations, which took primitive values (or complex instances whose origin is recursively known) as input.

In addition, in order to exert more control over data-types, the programmer can create explicit data-types. These data-types are also classes, but with additional information about their internal structure in order to make AIBench able to use them for both showing their parts to the final user in the GUI or be selected as input in operations. Moreover, AIBench is able to update its GUI components if the state of the data-type instances change. This is achieved by making use of the Observer design pattern [31]. Any complex data-type extending java.util.Observable will be observed by AIBench. More details can be found in [36].

In addition, related to data-types AIBench also defines the concept of transformer. These abstractions are very useful when an AIBench application invokes other software (a typical task is the need to adapt data structures, for example, using the adapter design pattern [31]). A transformer is a method of some class which takes an object of one type as input and returns an object of another type. If such a method is declared, AIBench can perform automatic data type conversions.

2.3. Views
Views are intended to visualize the results of the executed operations in a friendly way. In this sense, one view is associated to a given data-type and is implemented through a Java class ensuring two requisites: (i) it must extend the standard class JComponent and (ii) it must have a constructor with only one parameter of the same type of the class it visualizes.

When the framework needs to visualize a data-type, it will create an instance of the view class and pass the data through its constructor. Once the component is rendered, it is displayed to the user. Following the design principle of smart defaults, AIBench views are not mandatory: a default view is provided based on the standard toString method and also including a bean property inspector.

2.4. Framework overview
Every AIBench application can be seen as a collection of operations, data-types and views. The programmer only needs to decide how to divide the problem specific code into objects of these three entities. The framework will carry out the rest of the work to complete a real runnable application. These tasks include:

- Create a user interface skeleton where the user is allowed to select the operations he/she wants to execute.
- Retrieve the user parameters of a given operation whenever it is needed. The parameters could be both primitive values (numbers, strings, booleans) or any data-type if it was previously created by an operation.
- Run operations, gathering the results and keeping them available for further use.
- Display the results through custom (or default) views.
- Keep track of the executed operations containing all the information needed to repeat the experiment in the future.

In order to provide the programmer with these functionalities, AIBench incorporates two internal modules: the Clipboard and the History. The Clipboard is a complex data structure repository which contains the output of the executed operations, classified by their type (class). This structure allows the user to examine what was produced during the current session and the possibility of forwarding these objects to subsequent operations. The History keeps track of what operations were executed and which objects were used as input. This structure allows the framework to entirely reconstruct the experiment in order to re-execute it in the future.

The Clipboard and the History belong to the Core of the AIBench framework. Figure 3 shows the components and their internal architecture comprising four packages. The aibench core package contains the framework main components. The aibench gui package contains the Workbench plugin, responsible for generating the user interface. The aibench service plugins are also part of the AIBench platform and provide helpful functionalities (see Subsection 3.3). Finally, the user’s plugins package represents any AIBench developed application composed of several plugins.

![Figure 3. AIBench components and internal architecture.](image-url)

The sequence diagram shown in Figure 4 describes the steps followed internally by the framework when executing an operation (i.e., an IPO process) and also includes the AIBench objects responsible for each task. When the user launches an AIBench operation through the user interface (MainWindow), a dialog (InputDialog) is dynamically generated by the Workbench with the goal of collecting the input parameters. Once the user introduces the operation parameters, the Core instance launches an execution thread (ExecutionThread) that processes the operation. When the operation results are obtained they are added to the Clipboard, which is responsible for retaining them for further operations. Additionally, every time the operation execution finalizes, the History stores the execution context.
3. Framework functionalities

This section briefly describes the main capabilities and services provided by AIBench, such as dynamic GUI generation, automatic script construction and its reusable component model.

3.1. Dynamic graphic user interface generation

In each new software project the design and generation of GUIs requires a considerable amount of time. One of the main objectives of AIBench is to automatically provide a default GUI for the application logic developed by the programmer. The operational model of AIBench, in which the application processes and their input/output are well defined, allows the framework to automatically infer and build many user interface components and behaviours. This functionality is achieved in three main ways:

(i). Providing the application main window and a basic event handling.

(ii). Generating input dialogs for requesting the user parameters before a given operation is executed.

(iii). Constructing default views for data-types whenever custom ones are not available.

By default, the main window of any AIBench application contains five zones (see Figure 5). All the implemented operations are located in the menu bar of the application (it is also possible to enable a generated toolbar, containing a specified subset of operations). A Clipboard tree displays the AIBench Clipboard contents, that is, all objects generated by the executed operations. A History tree shows the AIBench History, that is, all the operations executed with their corresponding inputs and generated outputs. The central panel of the frame is used to display the contents of the objects through the AIBench views (default or custom). Finally the bottom area is used to arrange tools or add-ins. One tool included by default with the framework is the Shell, which can be used to run previously generated scripts in order to automate the execution of operations (see Subsection 3.2).

Figure 4. Sequence diagram describing the execution of an AIBench operation.
The second service provided by the user interface of AIBench is the dynamic generation of input dialogs to gather the parameters of each operation. When the user requests the execution of a given operation (through the main menu or by performing a secondary-click over a data-type and choosing one of the applicable operations), a dynamically generated input dialog appears on the screen. Figure 6 shows an input dialog for an arbitrary operation with six input ports.

Figure 5. Screenshot showing the overall layout of a typical AIBench application (main window).

Figure 6. Automatically generated input dialog for an operation.
In order to automatically generate these dialogs from the input ports of a given operation, AIBench follows the mapping algorithm described in Table II. These standard input dialogs are suitable for every operation, though it is also possible to provide a customized input dialog for a given operation.

<table>
<thead>
<tr>
<th>Java type</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive type (byte, int, long, float, double, char)</td>
<td>Text field</td>
</tr>
<tr>
<td>Boolean</td>
<td>Check-box</td>
</tr>
<tr>
<td>Enum type (Java 1.5)</td>
<td>Combo-box with the enum constants</td>
</tr>
<tr>
<td>A class with a String-constructor (primitive wrappers, String, custom)</td>
<td>Text field</td>
</tr>
<tr>
<td>java.io.File</td>
<td>Text field with a browsing button to select a local file/directory</td>
</tr>
<tr>
<td>Other class</td>
<td>Combo-box containing the instances of the same type available in the Clipboard</td>
</tr>
</tbody>
</table>
| Array                                         | a) Select-Mode: Combo-box to select an array of the same type in the Clipboard  
|                                               | b) Create-Mode: Enhanced component containing the component inferred using the above criteria, plus an empty list where several elements can be added |

Thirdly, data-type views are deployed in a tabbed pane located in the middle of the main frame. When the user clicks an object in the Clipboard or History trees, the views associated with it (one or more) are instantiated and added to the pane. As was mentioned above, if a specific view is not available, AIBench provides a default view for every data-type, since it can display any java.lang.Object. Figure 7 depicts two examples of default views available in the proposed framework. These views are structured in two parts. The upper zone contains the toString result or, if the object is an array or inherits from java.util.Collection, a two dimensional table. The bottom zone contains a general purpose bean inspector.

![Figure 7. Example of default views in AIBench.](image)

3.2. Automatic script generation
One of the main objectives of the project was to support the exact reproduction of the original execution, that is, to allow the user to repeat experiments, keeping or changing the input data and relevant parameters.

As shown in Table II, when the user is about to execute an operation, the generated input dialog contains one control for each input port. If the type of one input is not simple (primitive, enum or a class with a String-based constructor), AIBench uses the Clipboard in order to allow the user to choose one object from it. In other words, the user can only specify (i) a primitive/enum value, (ii) a string to be used in a constructor or (iii) an object previously generated by the execution of an operation. Then, the source of any parameter is always well-known and traceable. This is a key idea, since it allows AIBench to store all necessary information about the experiment to make it repeatable. All this information is held in the AIBench History.

The AIBench module managing this information is the AIBench Shell, which actually consists of an external module (a plugin included by default) built on top of the open source BeanShell project [37]. Figure 8 shows the GUI of this component containing two text areas. The left one (not editable) automatically generates a script at the same time the user executes operations. The right one (editable) allows the user to write (or paste) code (scripts). The user could then copy and paste the source code from the left side into the right side in order to get the experiment reproduced. The session script can also be saved using a plain text file format, subsequently edited for changing parameter values or adding any additional BeanShell language structure (e.g., loops, conditional statements, etc.) and finally loaded for re-executing it.

This script-based approach is one of several alternatives to achieve the repeatability. For example, an XML representation of the experiment, which is also planned, could be used instead. However, the implemented option has two main advantages. Firstly, it assists the programmer to become more familiar with the internal API of AIBench. Secondly, the automatically generated script can be easily and thoroughly modified in order to change the parameter values or altering its flow including new loops to execute the entire experiment over multiple data.

3.3 Reusable component model
AIBench uses a custom fork of the open source Platonos Plugin Engine [38]. However, the development of this plugin engine is currently discontinued, so other engines are being considered for replacing it, like the Java Plugin Framework [39] or an implementation of the OSGi specification [40-42].
Plugin engines are suitable for applications that are extensible by nature, like frameworks. They add the concept of plugin: a software component providing a set of classes and/or resources that are plugged in the application and exposed at run-time. Plugins are isolated by default increasing the modularity and ensuring that accidental coupling is not introduced, but they can also interact by establishing dependencies or extension points. A dependency between plugins allows one plugin to require other plugins to be present at run-time and to be allowed to access their classes and/or resources. An extension point declares a place where some plugin can be extended by another plugin (extension), usually providing a specific interface implementation.

Developing an application with AIBench requires operations, data-types and views (with the needed classes and resources) to be bundled into a plugin that can consist of a folder or a .jar file. A special XML file, named plugin.xml, must be included. This file contains both essential information and customization parameters such as which operations, views, transformers, custom input dialogs or icons are included, dependencies on other plugins, etc. To perform some tasks at start-up, a plugin can define, via the plugin.xml file, a lifecycle with a special class extending PluginLifecycle, whose start method will be invoked when the application initializes.

Every AIBench application requires the programmer to supply at least one plugin, but it is also possible to split the application into more than one plugin and, specially, reuse other plugins by stabilising dependencies between them. The Core of AIBench and the Workbench are also plugins. In terms of plugins, the operations extend the Core plugin while the views extend the Workbench plugin.

In addition, the AIBench framework provides three different service plugins commonly used in scientific applications: (i) the update manager service, giving the capability of managing a customizable remote plugin repository to any deployed AIBench application, (ii) the serialization service, able to manage all data-types implementing the standard java interface java.io.Serializable by providing ready-to-use hard disk operations and (iii) the documentor service, for automatically generating a full HTML technical report with a detailed description of all components currently coded inside the AIBench application.

AIBench has an open API that can be used to create very specific behaviours. Since the Core and the Workbench are also plugins, it is also possible to create plugins which depend on them and thus access their methods. The AIBench Shell is, for example, an advanced but independent plugin that works tightly with the Core to generate a script as operations are being executed.

4. Programming applications
In order to gain a deeper insight into the proposed framework, this section presents several advanced functionalities together with a real successfully deployed AIBench application for accurate protein quantification. The AIBench website [43] contains a manual and several illustrative examples explaining how to start programming with the framework using the Eclipse IDE (a Netbeans version is also planned).

4.1. Essential files
Before starting, it is necessary to present the essential files of the platform and their location inside an AIBench application. Once the framework development kit (SDK) is downloaded from the AIBench website and extracted into a local directory, the programmer can start developing his application. The AIBench SDK is the starting point of the application, since the
source code, the resources and the configuration files are added and/or modified in this folder until the application is finished. Figure 9 shows the file structure of the SDK.

Figure 9. AIBench SDK file structure.

The most relevant files and directories are:

- `<conf>` directory. Contains the following configuration files.
  - `aibench.conf` - A small file to set the plugin engine in debug mode (more verbose at boot). It also specifies the splash screen image.
  - `core.conf` - Used by the Core plugin. Defines some miscellaneous options like the number of threads of the core execution engine and can also override the name and the default location of the operations defined in the programmer plugins.
  - `workbench.conf` - Used by the Workbench plugin. Almost all of the visual aspects, especially which tools are visible, can be tuned in this file.
  - `template.xml` - Defines the graphical layout of the application (i.e., where to place the different panels in the main window).
  - `log4jconfig` - The standard Log4j configuration file.

- `<doc>` - Contains the AIBench programmer manual.

- `<lib>` - Contains the core classes to bootstrap the framework and the Platonos plugin engine.

- `<plugins_bin>` - Stores the available plugins in binary format. They can also be placed in a .jar file. By default, this directory contains the built-in plugins like the
Core, Workbench, the AI Bench Shell, and some others. When the programmer creates his plugin, the binary form (.class files) must be placed here. Additional plugins can be added to the main application by only copying them into this directory and restarting AI Bench.

- `<plugins_src>` - This directory contains the source code of the plugins. As in the `plugins_bin` directory, each plugin has its own folder.

In addition, the root directory of any given plugin must contain the `plugin.xml` file, which is a descriptor of the component and one of the most important files. Figure 10 illustrates an example of this file which is also bundled with the SDK in order to be used as a starting point.

```xml
<plugin start="true">
  <uid>sampleplugin</uid>
  <name>A sample Plugin</name>
  <version>1.0</version>
  <!-- lifecycle class. Used to detect when the plugin is started. The class should inherit from org.platonos.pluginengine.PluginLifecycle -->
  <lifecycleclass>mylifecycleclass</lifecycleclass>

  <!-- DEPENDENCIES: If you need classes from other plugins, you need a dependency! NOTE: If you need classes from plugins that you extend (for example, the core) you don’t need to put the dependency. NOTE: This dependency is a sample. If you uncomment the following lines the plugin will fail! -->
  <dependencies>
    <dependency uid="aibench.workbench"/>
    <dependency uid="aibench.core"/>
    <dependency uid="sing.datatypes"/>
  </dependencies>

  <!-- EXTENSIONS: The extensions that this plugin is connected to -->
  <extensions>

    <!-- CORE EXTENSION SAMPLES: Operations, Datatypes... -->

    <!-- EXTENSION 1: A sample operation, extending the core. Please note: you can put as many operations as you wish: one extension per operation -->
    <extension uid="aibench.core" name="aibench.core.operation-definition" class="sampleplugin.ReadDocument">
      <operation-description name="Open document" path="1@File" uid= "sampleplugin.readdocument"/>
    </extension>

    <!-- Transformers". A transformer definition tells AI Bench that you have a method to convert an instance from one class to an instance of another class -->
    <extension uid="aibench.core" name="aibench.core.transformer-definition">
      <transformer-description sourceType="java.lang.String" destinyType="es.uvigo.ei.sing.aibench.shell.DummyDataType" transformerClass="es.uvigo.ei.sing.aibench.shell.Transformer" methodName="transform"/>
    </extension>

    <!-- WORKBENCH EXTENSION SAMPLES (GUI related) -->
    <extension uid="aibench.workbench" name="aibench.workbench.view">
      <view name="Sample Datatype View" datatype="sampleplugin.OneClass" class="sampleplugin.OneViewComponent"/>
    </extension>
  </extensions>
</plugin>
```
The file depicted in Figure 10 is structured in four main sections:

- **Plugin description and identification.** Defines a unique identifier, a name and a version number of the plugin. It is also possible to define a lifecycle class used if the programmer needs to perform some actions at the plugin start-up.
- **Dependencies.** If the plugin needs to use any class or resource available in another plugin, it must establish a dependency on it.
- **Core extensions.** This is one of the most important sections of the file. Here, the programmer connects operations to AIBench (in terms of the plugin engine, an operation is an extension of the Core). It is also possible to define data-types transformers (see Subsection 2.2).
- **Workbench extensions.** This section contains all the visual extensions and customizations. The most important extension is the one used for connecting a view to render a data-type. This section can also be used to define other visual customizations like (i) custom icons for operations and data-types, (ii) custom input dialogs to gather the user parameters for a given operation, (iii) advanced custom components placed in a given zone of the main window and (iv) change the operations visibility.

Figure 10. Code example of a plugin.xml file.
(by default an operation can be invoked by the user via the main menu or a pop-up menu over a data-type).

Once the plugin.xml contains the desired operation configuration, the programmer can start AIBench to see the results. The AIBench SDK provides an ANT script to build the project and two launching scripts to initialize the application (run.sh for UNIX/Linux and run.bat for Windows). The project can, of course, also be built and launched from the programmer’s favorite IDE.

4.2. Operation specification

AIBench operations are Java classes including declarative information through Java 1.5 annotations. In this context, the @Operation class-level annotation tags a class as an AIBench operation and the @Port method-level annotation specifies its input/output ports. Table III and IV describe the attributes of these annotations.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the operation. This will be used, for example, in menus. This value can also be established in the plugin.xml file.</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>A brief description of the operation. This text appears, for example, in the header of the dynamic dialogs generated by the Workbench.</td>
<td>&lt;empty string&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the port. This text will be used, for example, to render a label in the dynamic dialogs.</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>A brief description of the port. This text appears, for example, near the corresponding control in the dynamic dialogs.</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td>direction</td>
<td>Direction (enumerated type)</td>
<td>The data flow direction (INPUT, OUTPUT or BOTH).</td>
<td>Direction.BOTH</td>
</tr>
<tr>
<td>defaultValue</td>
<td>String</td>
<td>The default value of the port (must be an INPUT or BOTH port). This value will be used, for example, to populate the input dialog with default values. The value can not be used to set complex values, only primitives or classes with a String- constructor.</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td>validateMethod</td>
<td>String</td>
<td>The name of an existing method in the same class that receives the same parameter as the method of this @Port. This method should throw an exception if the input is not valid and do nothing otherwise.</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td>order</td>
<td>int</td>
<td>The ports will be called by the Core in this order. If two ports have the same order, an INPUT/BOTH port is called before than an OUTPUT port. If the two ports have the INPUT or BOTH directions, there is no determined behaviour.</td>
<td>-1</td>
</tr>
<tr>
<td>allowNull</td>
<td>boolean</td>
<td>The user can select NULL as input of this port, although there are objects of this type in the clipboard.</td>
<td>false</td>
</tr>
<tr>
<td>lock</td>
<td>boolean</td>
<td>Guarantees that the object passed through this port can not be sent to any other operation during the execution of the current operation. This is very useful if the operation makes changes to the object and mutual exclusion must be preserved.</td>
<td>false</td>
</tr>
</tbody>
</table>
In addition to the @Operation and @Port annotations, an operation can use another two optional annotations: @Progress and @Cancel, which are intended to take control over the running process, especially when it takes a long time to finish.

The @Progress annotation is used to monitor the advance of a running operation. It annotates one method of the operation class, which should return a Java bean with all the information related to the progress of the operation the programmer wants to show to the user. Normally, the operation should periodically call the setter methods of the bean during its process, whereas the Workbench uses the getter methods to show their values to the user. The @Progress annotation has no attributes.

The @Cancel annotation can be used to request a running operation to stop. It also annotates one method of the operation class. If this annotation is present, the user will be able to request the operation to stop. If this button is pressed, the annotated method will be called and all the outputs of the operation will be ignored, but there is no guarantee that the process will stop. The programmer is responsible for stopping his own procedure. The @Cancel annotation has no attributes.

### 4.3. Explicit data-types and transformers

In some situations, it is very useful to expose internal parts of complex data-types to the framework. AI-Bench contains a set of annotations for data-types to convert them into the so-called explicit data-types. The main annotation is @Datatype, which is a class-level annotation present in every explicit data-type used to specify the particular structure of the object. Table V shows the three different definable structures.

<table>
<thead>
<tr>
<th>Java type</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>This is the default and no annotation is needed.</td>
</tr>
<tr>
<td>List</td>
<td>It contains a variable number of sub-elements of the same type. Java arrays and java.util.Collection instances are automatically considered Lists.</td>
</tr>
<tr>
<td>Complex</td>
<td>A heterogeneous object with a finite set of sub-parts. For example, the object could internally have one List, one Simple and two Complex internal parts. Each item is published by a method annotated with the @Clipboard annotation. It must define a method (returning a java.util.List object) annotated with the @ListElements annotation giving access to all sub-elements.</td>
</tr>
</tbody>
</table>

There is also another useful annotation, named @Property, which can be used inside any explicit data-type; it could be used, for example, to display simple Strings in the AI-Bench tree.

Another interesting functionality related to data-types in AI-Bench is the transformer concept. As previously mentioned, transformers are especially useful in the integration of different projects or APIs, where the data-types are not the same, but compatible in some way.

In Java terms, a transformer could be a method of any class that takes an object of the source type as input parameter and returns another object of the target type. The method can be static or not, but in the latter case, the class must define a default constructor. In order to inform AI-Bench that there is an available transformer, the plugin.xml configuration file should be edited.

### 4.4. Custom views
Custom views are used to replace the default view provided for every data-type. A valid view is any class which (i) extends the standard visual class `JComponent` and (ii) contains a constructor with a parameter of the same type of the data-type it renders. One advantage of these two requisites is the total absence of intrusion, since there is no need to neither extend any class nor use any specific annotation. In addition, the use of `JComponent` eases the reuse of already implemented user interfaces from other projects. Finally, the view should be connected to the framework through the `plugin.xml` file.

Please note that the default view is still available (this behavior can be changed) because having multiple views for the same data-type is allowed.

A typical question is how to deal with subsequent modifications carried out by operations over the same data-type instance. AI Bench does not provide any abstraction to listen to these changes. The recommended way is to make use of the `Observer` design pattern, using the standard Java classes `java.util.Observer` and `java.util.Observable`.

4.5. Application interface customization

By this point, all the basic concepts have been presented. As the application evolves, the programmer can perform interface customizations. AI Bench has many configuration options that were added in different versions of the project as they were demanded by the programmer community. Some of the most interesting customizations are summarized below:

- **Main window layout.** One typical adaptation is to change which components are present in the main window and their position. The default components visibility can be defined in the `workbench.conf` file and the available positions and layout in the `template.xml` file, which is shown in Figure 11. This XML file establishes a set of slots and a document viewer in a tabular style, similar to HTML tables. Slots are identified by a user-defined symbolic name (e.g., `left`, `top`, `down`, etc.) and holds components (more than one is allowed) like the available trees, the AI Bench Shell, and other plugin custom components. Each component is attached to one slot by the configuration file (or even programmatically). The document viewer defined in the layout specifies where to place the data-type views. This simple solution allows the programmer to easily change the aspect of the final application drastically.

- **Custom input dialogs.** The automatically generated input dialog shown before an operation is executed, can also be programmed, in order to replace the default. This is an advanced task which implies working with an internal data structure used by AI Bench to represent the user parameter values and their precedence (basically indicates if the value is primitive, a string to be passed to a constructor or if it comes from the Clipboard). However, the default input dialogs are implemented in a package which was designed to be easily extended to avoid starting a custom dialog from scratch.

- **Overriding menu positions and visibility.** If a new application is started from an existing set of AI Bench plugins, a typical integration task is to exclude some unnecessary operations from user access or reorganize them in other menu positions to obtain a coherent and comprehensive application handling.

- **Other minor configurations.** Change the start-up splash screen (`workbench.conf`), set specific icons for both data-types and operations (`plugin.xml`), etc.
4.6. Case study: accurate protein quantification with the DPD AIBench application

The AIBench framework has been used to implement several successful applications from different scientific domains like metabolic engineering [44] or biomedical text mining [45]. In order to better support and explain the benefits of using the AIBench framework, and with the goal of showing different framework capabilities in action, here we sketch the main characteristics of DPD (Decision Peptide Driven) software, a novel AIBench application for accurate protein quantification.

Matrix-assisted laser desorption/ionization (MALDI) is an advanced soft ionization technique used in mass spectrometry for the analysis of a large number of important biomolecules (e.g., proteins, peptides, etc.). In this context, standard protein quantification methods based on protein separation have a number of drawbacks that make reliable quantification difficult [46]. Moreover, the interpretation of the MALDI data from direct and inverse labelling experiments is time-consuming, requiring a significant amount of time when performing comparisons manually. In such a situation, there is a need for a computer tool able to extract and subsequently identify peptides that remain constant in expression level through different sets of a typical in-gel digestion workflow.

The DPD software shortens and simplifies the searching of those peptides that must be used for quantification from a week to just some minutes. To do so, it takes as input several MALDI spectra and aids the researcher in an automatic mode (i) to compare data from direct and inverse $^{18}$O-labeling experiments, calculating the corresponding ratios to determine those peptides with paralleled losses throughout different sets of experiments, and (ii) allows the use those peptides as internal standards for subsequent accurate protein quantification using previous labelling results.

Figure 12 shows a screenshot of the DPD (Decision Peptide Driven) software. As can be seen, the DPD application is configured taking advantage of several AIBench customization capabilities (see Subsection 4.5): (i) the History tree for easing the management of the application is not shown, (ii) the operations menu is replaced by an AIBench toolbar allowing only
the execution of problem-related (high-level) operations and (iii) the bottom area (used to arrange tools or add-ins) is hidden to gain more work space.

Figure 12. Screenshot of the DPD application showing the initial splash screen and the about dialog box.

The first type of analysis supported by the DPD application (finding out reproducible peptides from experimental data) comprises several steps that should be executed in a linear predefined order. Figure 13 shows a screenshot of the DPD software after loading the required data. As can be seen from Figure 13, the ‘Create Labeling Experiment’ button is now active, and an automatically generated pop-up dialog with seven different input ports is displayed for capturing the user parameters. Moreover, a custom view showing raw peak intensities is presented in the central panel. This custom view is configurable by selecting different visualization options and it is also enhanced with a graphic representation of peak intensities using the JFreeChart library [47] as an embedded plugin.
The second main functionality of the DPD software is provided by the protein quantification module. By previously executing the ‘Load Quantification Data’ operation, the ‘Create Quantification Experiment’ button becomes active in the toolbar and the user can interactively (i) specify different experimental parameters (ii) re-execute the current experiment and (iii) visualize the updated result data, all in the same advanced custom view (Figure 14 demonstrates this functionality). In contrast to the step-by-step workflow shown in Figure 13, in this case a cyclic process is modulated by using only one custom view. Moreover, from Figure 14 it can be also seen how the Clipboard is able to represent different explicit data-types (i.e., list and complex). In addition, multiple data-type views can be opened at the same time in order to represent different elements from the Clipboard.
Figure 14. Protein quantification experiment using the DPD software.

The Decision Peptide Driven AIBench-based application is free, multiplatform (Mac OS / Linux / Windows) and designed to be intuitively used by wet-lab researchers. DPD is available at: http://sing.ei.uvigo.es/DPD/ together with sample data. There is also the possibility of installing/executing the DPD application by using Java Web Start technology. A brief overview and a detailed tutorial are also provided from the web.

5. Conclusions and future directions
This paper introduces AIBench, a new Java desktop application framework oriented to the scientific community, which was born inside a research group interested in increasing its software development productivity. The proposed framework presents several advantages including:

(i). It provides the programmer with a proven design and architecture. By implementing the IPO (input-process-output) model following the MVC (model-view-controller) design pattern, the applications developed with AIBench are divided into three types of well-defined objects: operations, data-types and data-type views, which identify units of work with a very high coherence that can be easily combined and thus reused.

(ii). It provides the programmer with services which are independent of the application scope, but useful for any scientific application such as input dialog generation, application context management, experiment repeatability, concurrent execution of operations, etc. The programmer can spend more time on the problem-specific requirements rather than on the low level details.

(iii). It has a plugin-based architecture allowing applications to be easily developed by adding new components, each one containing a set of AIBench objects. The coarse-
grained integration between functionalities is carried out by establishing dependencies between these plugins. This allows reusing and integrating functionalities of past and future developments based on this framework.

In our daily use of AIBench, every new application helps us in finding bugs and raises new ideas for novel features. Moreover, our experience designing and using AIBench led us to several interesting conclusions and taught us lessons.

The objective that every workflow should be repeatable, forced us to design the framework in such a way that the origin of every input should be known. In the case of non-primitive objects, their origin is the Clipboard. This may complicate the design of some applications, forcing the programmers to add operations for creating non-primitive objects from primitive ones. In order to alleviate this requirement, there are two alternatives, both including the creation of custom input dialogs. On the one hand, the programmer can give up the repeatability feature by passing a null value (unknown) as the origin of an object. On the other hand, and inspired in the prototype design pattern [31], the programmer can give a serialized version of the complex object, so AIBench can recreate this non-primitive object in further sessions.

Another possible difficulty appears when developing highly interactive applications. In such cases, the AIBench input-process-output architecture may be very coarse grained. Actions like sorting tables, manipulating graphs, etc. can be implemented inside custom views and are not included in the main workflow as normal AIBench operations. However, there are situations in which this decision is not clear and the programmer is forced to decide between an explosion of small operations or giving up the repeatability of such actions. Nevertheless, since AIBench operations are simple classes with well defined input and output (ports), these small operations could be easily included in macro-operations.

In its original concept, AIBench was designed to create prototypical applications, but not final user applications. This led us to the non-intrusiveness principle of the framework, in order to reuse the application logic (operations, data-types and views) in a final application outside AIBench. However, user feedback forced us to improve the framework by adding many customization options (e.g., personalized input dialogs, configurable application layout, etc.) in order to create real final applications with AIBench.

AIBench is free software (distributed under the terms of the GNU Lesser General Public License) and both the source code and the SDK can be downloaded from its website [42]. Recently, we have opened the CVS access to the source code and installed some other collaborative tools such as a wiki and a discussion forum.

The future directions of the framework include (i) new improvements e.g. more customization capabilities, a graphical environment to design complete operation workflows and (ii) a new web-based Workbench enabling AIBench to deliver server-side Internet scientific applications.

Acknowledgements
This work was partially funded by the Spanish Ministry of Science and Innovation, the Plan E from the Spanish Government and the European Union from the ERDF (TIN2009-14057-C03-02). M. Reboiro-Jato was supported by a pre-doctoral fellowship from University of Vigo. We want to thank all the members of the Bioscope group for their brilliant conception of the DPD application.

References
7. Ruby on Rails. Available at: http://rubyonrails.org [02 November 2010]
8. Symfony Web PHP Framework. Available at: http://www.symfony-project.org [02 November 2010]
15. JUnit. Available at: http://junit.org [02 November 2010]
17. Sun Microsystems. The Java Compiler Compiler. Available at: https://javacc.dev.java.net/ [02 November 2010]
19. FFmpeg. Available at: http://ffmpeg.mplayerhq.hu [02 November 2010]
37. BeanShell - Lightweight Scripting for Java. Available at: http://www.beanshell.org [02 November 2010]
40. Knopflerfish OSGi. Available at: http://www.knopflerfish.org/ [02 November 2010]
42. The Eclipse Foundation. Equinox. Available at: http://www.eclipse.org/equinox/ [02 November 2010]
43. AIBench – AI Workbench!. Available at: http://www.aibench.org [02 November 2010]
47. JFreeChart. Available at: http://www.jfree.org/ [02 November 2010]