Development and implementation of a distributed-object job-execution environment

Rod Fatoohi[a,*] and Lance Smith[b]

[a]*San Jose State University, San Jose, CA 95192, USA and NASA Ames Research Center*

[b]*XUMA, San Francisco, CA, USA*

This paper describes the development and implementation of a distributed job execution environment for highly iterative jobs. An iterative job is defined here as a binary code that is run multiple times with incremental changes in the input values for each run. An execution environment is a set of resources on a computing platform that can be made available to run the job and hold the output until it is collected. The goal is to design a complete, object-oriented execution system that runs a variety of jobs with minimal changes. Areas of code that are unique to a specific type of job are decoupled from the rest. The system allows for fine-grained job control, timely status notification and dynamic registration and deregistration of execution platforms depending on resources available. Several objected-oriented technologies are employed: Java, CORBA, UML, and software design patterns. The environment has been tested using a simulation code, INS2D.

Keywords: Problem-solving environment, Object-oriented design patterns, Java, CORBA

1. Introduction

The current trend in data processing is a 3-tier solution. Lightweight clients communicate with middle tier application servers. These, in turn, communicate with the backend databases or execution engines. Advances in platform independent languages and frameworks have eased the process of porting and executing code based on different platforms. Java is a platform independent language that has gained widespread ac-

ceptance in the developer community. The Common Object Request Broker Architecture (CORBA) [15] is a distributed object request broker that removes the details of remote object communication and provides services that are needed in distributed applications. With CORBA, remote object methods and data are accessed via local proxy objects. This allows the programmer to write code as if the remote objects were local.

There has been an increased interest in a specific class of environments called Problem Solving Environments (PSEs). A PSE is defined as a computer system that provides facilities and services needed to solve a class of problems [11]. Many PSEs have been developed to help users solve problems in different areas. These environments differ in many aspects such as the type of the problems they solve, the features they provide, the platforms they support as well as their architecture and their design. They can be divided broadly into two main categories.

The first category is concerned with the development of advanced solution methods, the automatic and semiautomatic selection of these methods, and the integration of these methods to solve computational sciences applications. Examples of this category include products that are commercially available such as: Matlab, an interactive system for numerical linear algebra, and Diffpack, an object-oriented problem solving environment for multi-physics simulation [12]. Included in this category are experimental systems designed to solve more complex problems such as EDSS and PELLPACK. The Environmental Decision Support System (EDSS) [5] is a PSE developed by North Carolina Supercomputing Center to couple models based on knowledge from several disciplines in environmental science and decision support systems. It includes components and tools to build air quality models from interchangeable components. The Parallel (//) ELL-PACK [10] is a PSE for partial differential equations based applications developed at Purdue University.

The main goal of the second category of PSEs is to provide an efficient, easy to use environment for solving problems using distributed computational resources.

*Corresponding author: Professor Rod Fatoohi, Computer Engineering, College of Engineering, One Washington Square, San Jose State University, San Jose, California 95192, USA. Tel.: +1 408 924 4059; Fax: +1 408 924 4153; E-mail: rfatoohi@email.sjsu.edu.

Examples of this category include: Globus, Legion, NetSolve, DIAS, IceT, PRE, and Arcade. Globus [6], developed at Argonne National Lab and University of Southern California, is a set of services that provides basic capabilities and interfaces in areas such as communication, information, resource location, scheduling, security, and data access to geographically distributed resources. Legion [13], developed at the University of Virginia, is an object-oriented system designed to integrate a large number of machines and resources and provides the illusion of a single virtual machine to users. NetSolve [2], developed by Oak Ridge National Lab and University of Tennessee at Knoxville, is a system that allows users to access computational resources distributed across a network. The Dynamic Information Architecture System (DIAS) [1] is an object-oriented simulation system designed to provide an integrating framework for legacy applications at Argonne National Lab. IceT [9] is a framework for collaborative and high performance distributed computing based on Java developed at Emory University. The Production Realization Environment (PRE) [22], developed at Sandia National Labs, is a CORBA-based framework for the integration of a variety of applications such as legacy applications, databases, in-house tools, and commercial products. Finally, Arcade [3], developed by ICASE and Old Dominion University, is a web-based environment for designing, executing, monitoring, and controlling distributed applications.

We designed a simple but a powerful environment to execute highly iterative jobs. These jobs need to run several times to produce similar results based on incremental changes in the input values for each run. They are quite common in many simulation codes used to solve scientific and engineering problems. An example of these codes is the highly iterative Computational Fluid Dynamics (CFD) codes. These CFD codes (also called flow solvers) are numerically intensive applications that model airflow over wings or airframes. Our designed system has many features and capabilities of the above environments but it is unique in many ways. First it is solely based on object-oriented design and analysis, which make it quite flexible and easy to upgrade and maintain. Second, it is quite portable since it uses Java and CORBA with their inherent platform independence. Third, it has a scheduling capability that many other systems lack. And forth, it has been tested using a real simulation code, INS2D.

One of the driving forces behind this research was to find a clean, effective way to acquire the processing power of clusters of distributed machines. As machines capable of hosting an execution environment free up the resources to do so, they will register with and be assigned work from a dispatching object. Other machines might register based on the time of day. During the evening hours, they register and process jobs. During the day, they deregister and free up resources for other duties. The idea is to quickly maximize the available computing resources that would otherwise sit idle. By taking advantage of solid object-oriented design idioms, platform independent languages and industry standard broker architectures, a flexible, cost effective and dynamic problem solving environment can be achieved.

Our system architecture is designed with maximum reuse in mind. One of the goals is to develop a solution that enables different simulation codes to be "plugged in" with minimal code having to be rewritten. The architecture has been tested using the INS2D code, which solves the incompressible Navier-Stokes equations for steady state and time varying flow. In our design, we use the Unified Modeling Language (UML) to map out the architecture. We use Java as an implementation language and CORBA for communication and control between the tiers [16].

CORBA was chosen because of the flexibility it offers in the choice of programming languages. With the exception of having to acquire the Interoperable Object References (IORs) of different objects, the Java code is written as if all the pieces were running on the same virtual machine. There is no need to call socket libraries and create communication ports. All of these complicated details are abstracted out. This vastly simplifies the complexity of the code.

In this paper, we present the architecture, implementation, and design issues of applying object-oriented techniques to develop a distributed job execution environment. Several technologies are employed in our approach: Java, CORBA, UML, and software design patterns. First the architecture is introduced as a three-tier system. Then the objects in each tier are described in details. This is followed by an operational overview of the whole system. More details of the framework and design patterns are given afterward. Finally, testing results and some conclusions are drawn. Some of the environment features are discussed in more than one place for completeness. Throughout the paper, we focus on the implementation aspect of our system highlighting its main features and capabilities.

2. Java/CORBA architecture

2.1. Client/Server architecture

The design of this system is a batch-oriented and falls roughly along the lines of classic three-tier architecture. Within these three tiers are three broad categories of objects: CORBA objects, worker objects and utility objects. The design falls along well-defined boundaries for each type of object. Only the CORBA objects have knowledge of their workers and then only through their interface types. The worker objects are a loosely coupled set of Java interfaces, abstract adapter classes, base classes and, usually, specialization classes. Each worker object implements or extends the one above it (in the layers of responsibility described later). This allows workers to be "unplugged" and replaced without code changes to CORBA objects. Workers have no knowledge of each other. Utility objects streamline the code by providing static methods for common operations such as acquiring a reference to the CORBA naming service or getting a formatted timestamp. Other utility objects are worker creation factories. Factories themselves can be swapped in or out just like the workers.

CORBA provides the means for abstracting out the details of the underlying communication bus. Once a remote reference to a CORBA object is obtained via the CORBA naming service, the reference is treated as if the remote object is local to the current Java virtual machine. This vastly simplifies the problems of dealing with disparate computer platforms and operation systems.

The architecture has three interoperation pieces: Admin, Dispatcher and Solver, as shown in Fig. 1. Each of these high level objects has a set of worker objects that perform the low level tasks. The Admin object selects the jobs to be run. Its workers check each job's runtime requirements and repackage data into generic containers for transport. The collection of jobs is then moved to the Dispatcher object, where workers parse input files, record status and queue the jobs up for distribution. Solver objects come on-line and register with Dispatcher. Once a Solver object is registered, it asks for work. This Solver object then pulls a subset of jobs from the queue, and, via its worker objects, builds the needed execution environment and executes the jobs.

Abstraction and delegation allow the individual components to be only aware of the objects with which they have to communicate. Worker objects only interact with the high level objects that created them. Any number of Solver objects can request jobs from the same Dispatcher. Solver objects can be added or removed during the processing of the jobs in the queue. Solver objects can dynamically register with a Dispatcher object depending if resource loads on their host machines drop to sufficient levels and un-register if loads are too high (theoretically since that this feature has not been implemented yet, but the design of the feature may follow host monitored presented in [14]). Solver objects could also be time-based. They may only ask for work during a given time frame (such as midnight to six o'clock am). How much work a Solver object can handle at any given time is dependent of the machine on which it is running.

2.2. IDL files and interfaces

There are two Interface Definition Language (IDL) files that define the interfaces and structures for this system [15]. Though one file could have been used, the overriding design goal in this system is functional decomposition. These files are core.idl, which describes the CORBA interfaces that all types of jobs use, and simulation.idl, which defines a CORBA structure that is specific to a simulation code such as INS2D (in this case it is called ins2d.idl). The core.idl file defines three interfaces: Admin, Dispatcher and Solver. Their operations are defined below. The simulation.idl file defines an Environment structure that holds all the information needed to run a single job. Each job has a corresponding Environment object, which is passed from the dispatcher to the solvers. At the lower levels, data transfer is done via a CORBA Any (CORBA Any is a data type that can hold any primitive or user-defined CORBA type). This allows the simulation specific Environment object to be passed inside a more generic Any.

2.3. Object definitions

The three object categories – CORBA, worker, and utilities – are defined below:

2.3.1. CORBA objects
- *Admin* – (1st tier) connects with the user interface. It is represented by the AdminImpl object and collects job information and populates data structures that can be passed along the Object Request Broker (ORB). Data transfers from the console to Admin as Java objects and from Admin to Dispatcher as CORBA Any.
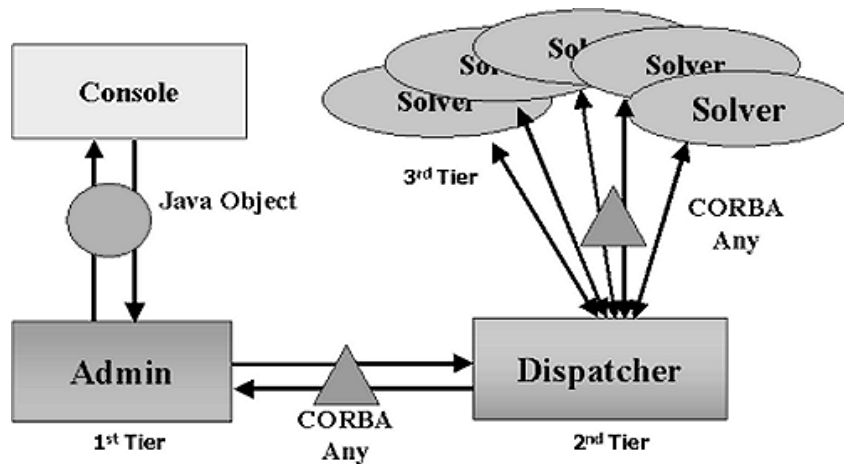
Fig. 1. Architecture framework.

– *Dispatcher* – (2nd tier) contains the scheduling logic, performs parsing, queuing and routing functions. It is represented by the DispatcherImpl object. Jobs are unpacked from their initial data structures and repacked into Environment structures. Initial values for worker objects are obtained by reading a properties file, BaseDispatcherProps.txt, at startup. Data is transferred to Solver and back as CORBA Any.
– *Solver* – (3rd tier) requests collections of jobs from Dispatcher, builds the pads, transports the input and data files and either runs the job itself or passes it on to a sub scheduler. It is represented by the SolverImpl object. Initial values for the worker objects are obtained by reading a properties file, BaseSolverProps.txt, at startup.

2.3.2. Worker objects

Worker objects perform the mechanics of data structure loading and job manipulation. The workers are composed of a pair of classes. Each one contains an abstract class and a concrete implementation (base) class. Dividing the responsibilities of these objects into two classes decouples the generic tasks from the specific ones. For example, all jobs must be checked to see if they have all the required attributes, but for each type of job those attributes will be different. By moving the details of requirements checking into the base class, the abstract adapter does not need to be changed with each new type of job.

There are three groups of worker objects: Admin workers (at the 1st tier, Fig. 2), Dispatcher workers (at the 2nd tier, Fig. 3), and Solver workers (at the 3rd tier, Fig. 4). They are described below:

2.3.2.1. Admin workers
– *Translator* – acts as the bridge between the existing GUI and the CORBA AdminImpl Object. It collects GUI values and input and data file locations. These values are wrapped in a generic Java object and passed to AdminImpl.
– *Packaging* – checks required job attributes and repackages the GUI data into job transportation structures, which are packed into CORBA Any for transport across the ORB.

2.3.2.2. Dispatcher workers
– *Parser* – parses the required and optional attributes and populates the Environment object.
– *Queue* – initializes and maintains the job queue. It sorts and loads the environments onto the queue. It also dispatches jobs to Solver objects if job requirements and Solver runtime values match. It may build a script files if required to do so.
– *DispatcherStatus* – records and returns Dispatcher and job status.

2.3.2.3. Solver workers
– *Receiver* – accepts collections of incoming jobs and builds the execution pad, which is a unique set of directories that holds the input, data and output files.
– *Transport* – handles the details of moving files from one platform to another.
– *Engine* – executes the jobs or passes them to a secondary scheduler (ex: Unix commands at and cron).
– *Post* – handles job post-processing.
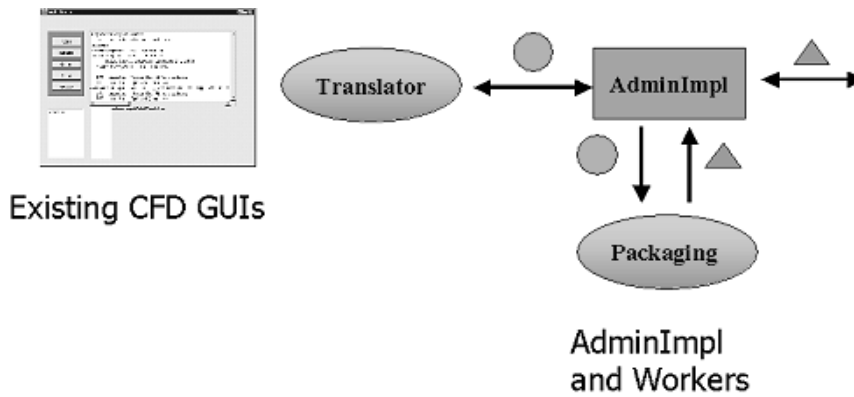– *SolverStatus* – records and returns Solver and job status.
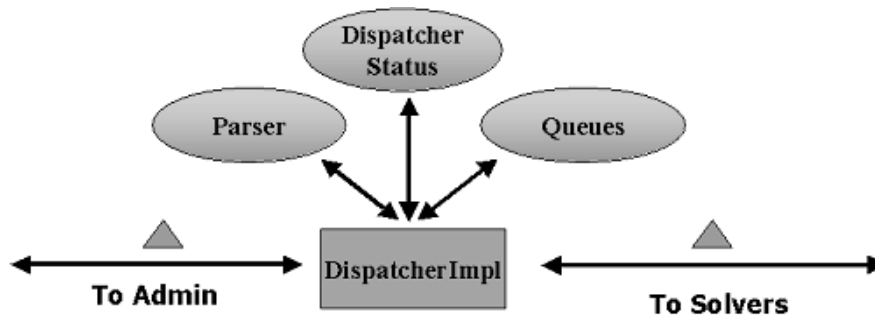
Fig. 2. The 1st tier.

Fig. 3. The 2nd tier.

2.3.3. Utility objects

Utility objects fall into three categories:

- *Data Holders* – storage objects for groups of related data.
- *Tools* – an object composed of static routines for getting hostnames, ORB references, formatted times, etc.
- *Factories* – dynamically loaded classes that build Dispatcher and Solver workers.

2.4. Layers of responsibility

The architecture for Admin, Dispatcher and Solver interfaces is designed around four layers, as shown in Fig. 5:

- Specification
- Manipulation
- Initialization
- Specialization

Specification is the interface level. All CORBA and worker objects implement interfaces. This is where the mechanics of the system are defined. All objects that interact with each other are referenced through their interface type. Manipulation, on the other hand, is the abstract adapter level. The generic functions common to all types of jobs are implemented in the abstract classes. Timing and lifecycle issues such as how long a Solver object will wait for jobs before shutting down completely are defined here.

Initialization is the base class level. The base classes provide default values for the queue and status maps. They also provide the implementation of operations specific to each type of job such as pad creation. Finally, specialization is used when a programmer wants to override the default values and operations provided in the initialization level.

3. Operational overview

3.1. Job definition

A job is defined to be a process that operates on a collection of attributes. Attributes are divided into two categories: required and optional. Unlike the required
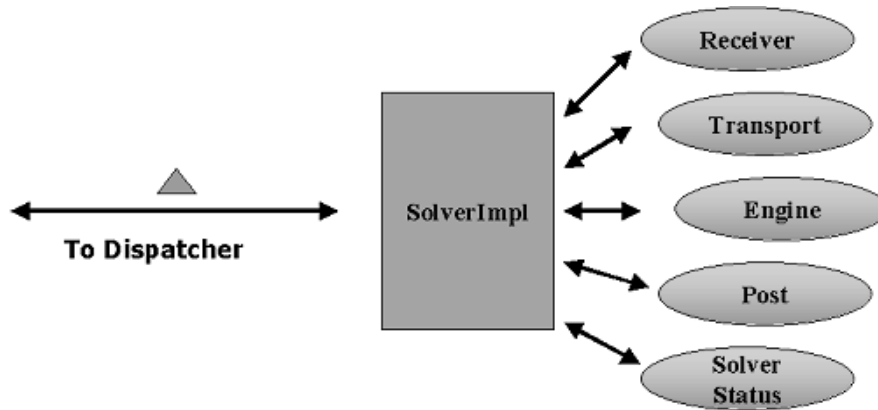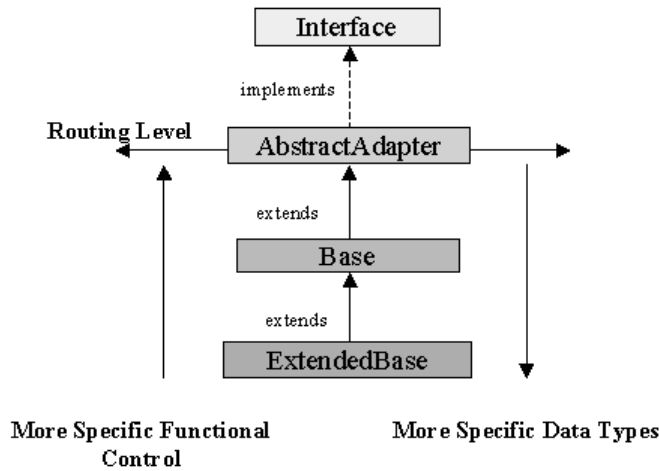
Fig. 4. The 3rd tier.

Fig. 5. Layers of responsibility.

attributes, the optional attributes may be present and may affect the way a job is run. Examples for required attributes are input and data files and for optional attributes are restart flag files, alert level, run level, and status level. The alert level helps the user in specifying how to be notified when the job status changes – by email, log, or pager. The job priority can be specified in run levels – using five levels. The status level defines how the user wants to be notified of changes in the job status – three levels are defined here: low (when the job starts and finishes only), medium (when the job moves from one CORBA object to another), and high (when the job moves from one method to another). The combination of required and optional data is contained in a data structure called Environment. A single Environment represents a job.

3.2. Acquiring jobs

Collections of jobs are acquired via a user console. A graphical user interface called the Basic Visual Frame (BVF) has been developed. It allows the user to select jobs, tag them with marker data (such as run level) and send the collection to the Admin object as a generic Java Object, as shown in Fig. 1. The Admin object, along with its workers, performs the following tasks: a) unpack the jobs, b) check the validity of these jobs (required files), c) repackage the required and optional data into an IDL defined data structure, JobContainer, d) push jobs onto JobList, a CORBA sequence of Job-Containers, and e) pack JobList into a CORBA Any to be sent to Dispatcher.

3.3. Parsing jobs

The Dispatcher object accepts a collection of jobs, as a JobList object, and transforms them into a collection of Environment objects. Each Environment object is populated with the required and optional attributes. The array of Environment objects are packed into a CORBA Any object.

3.4. Queuing jobs

Jobs are queued onto a job queue using a JobList object. The job queue has the following features: synchronized access, efficient sorting, discarding duplicate jobs, and automatic expansion. The queue is synchronized so that only one Solver at a time can access it. Concurrent access during job dispatching could result in having the same job being sent to multiple Solvers. Sorting functions are needed to propagate the queue according to some user-defined criteria. Due to the high-level of calls to the queue, a fast and efficient sorting algorithm is needed. Duplicate jobs cannot be allowed; therefore, jobs are referenced by a unique, randomly generated ID. Finally, the queue needs to expand automatically when new jobs are presented and parsed. After all jobs are pushed onto the queue, the queue is sorted according to a run level (provided as an optional attribute).

3.5. Dispatching jobs

Registration and deregistration of Solver objects are performed by Dispatcher. The Solver objects poll Dispatcher and continually ask for work as long as there are jobs to do. The Dispatcher object looks up the requested Solver runtime values and passes specific values to the Queue worker object, which checks each job against the Solver runtime values. If the job does not match the Solver values, it remains on the queue. If a job meets the Solver requirements, a script file is built to run the job, provided that the script option is enabled. After the script is generated, it is transformed into a byte array and placed in an Environment structure.

3.6. Running jobs

The process of running the jobs has four steps: building pads, transporting data and input files, executing the jobs, and post-processing.

3.6.1. Building pads

For each job, the Receiver worker object builds a pad, which is a set of temporary subdirectories where the job's data, input and output files are residing during job execution. The job ID is used as the name of the top-level directory in the pad. If a script file is used to run the job, the file is recreated from the byte array. The Receiver object builds a Map object that contains the job ID, the names of the pad directories, and files to be transported. The Map object is used by both the Transport and Engine worker objects.

3.6.2. Transporting files

The Transport object moves data and input files from Dispatcher to Solver. Transport takes a Map object and builds a set of download strings by combining download directories with filenames. Currently we are using the Trivial File Transfer Protocol (TFTP) to do the actual transport since TFTP does not require login or validation procedures.

3.6.3. Executing jobs

Jobs have two possible execution paths: direct execution or sub-scheduler execution. In direct execution, the Engine object runs the executable binary in a runtime process while in sub-scheduler execution, a Unix command (such as at) executes the shell script in a run time process. A run time process is created by getting an instance of the java.lang.runtime object and creating a new process via its exec() method. This process exists outside the Java virtual machine. An output stream is created from the process object to capture any characters that would normally go to System.out.

3.6.4. Post-processing

Post-processing involves manipulating the output files and/or transport off the Solver machine. Post-processing is not part of this research. An interface, abstract class and a base class have been provided as shells for future development. These empty classes are instantiated with the rest of the Solver worker objects. When this project moves to a production phase, post-processing will be fully implemented.

4. Frameworks and design patterns

The framework has been designed with a visual modeling tool, Rational Rose [17], using the Unified Modeling Language (UML). By using UML to define the architecture, the software engineer is forced to have a

solid design before code writing can begin. This ultimately speeds up the development cycle by finding problems with the methodology before they become difficult to fix.

We used several software design patterns throughout the design process [7,21]. Design patterns, which are a collection of well-known idioms, offer reusable solutions to software problems. CORBA offers some of these design patterns in its architecture such as delegation, proxy and broker [16]. In addition, we used the template method pattern and the factory pattern. In the following we describe some of these patterns.

4.1. Template method pattern

The template method is a behavioral pattern. It is used in designs where the class will be used in multiple programs but the overall responsibility of the class remains the same [8]. The class is implemented as an abstract class. Only the methods that provide the generic class function are implemented. The specialization logic is contained in the abstract methods. This requires programmers to implement the class specific logic in the base class that extends the abstract one.

Our design takes the template method and extends it. All the worker objects implement this pattern. The abstract class implements an interface and the base class is usually, but not always, overridden. The interface requires the abstract class to only accept and return well known data types. The class that extends the base class accepts these types but casts them to the specific types that it needs. The workers have one class for each specification level. The real power of this method comes when there are several operations that are performed inside some kind of a loop. The abstract class controls the loop logic and calls sequence of abstract methods. These methods accept and return generic data types (in this case, Sets and Anys) and are implemented in the base class.

In the base class or the class that extends it, these methods cast the incoming arguments to implementation specific data types, perform the detail logic and then return an implementation specific type wrapped in an Any. The abstract class has no previous knowledge of the type that has been returned to it. Any is then routed to other worker objects or to another CORBA object. By combining the template method pattern with the functional separation of control and detail logic, we develop a powerful, flexible base for the worker objects.

4.2. Factory pattern

In the factory design pattern [20], factories create methods to dynamically create new objects. We used this pattern in the DispatcherImpl and SolverImpl objects. Overall, there are two factories and eight workers, as shown in Fig. 6. To make the code more flexible and easy to maintain, SolverImpl uses a factory pattern to decouple the instantiation of the concrete class from SolverImpl. This pattern requires the use of a delegate object, called a factory, which creates the Engine object for SolverImpl.

The factory, BaseSolverFactory, has a getEngine() method that creates a new BaseEngine object and returns a reference to it. SolverImpl instantiates a factory and then calls the factory's getEngine() method. SolverImpl now has access to a BaseEngine object that it references through the Engine interface. It has been decoupled from the concrete details of BaseEngine. The factory method pattern provides an application-independent object (SolverImpl) with an application-specific object (BaseSolverFactory) to which it can delegate the creation of other application specific objects (BaseEngine and other workers).

The down side to this pattern is that SolverImpl needs to have prior knowledge of the factory's concrete type. Unfortunately, this has the effect of substituting one maintenance problem for another. SolverImpl still has to have knowledge about the BaseSolverFactory class. This problem can be overcome by making SolverImpl reference the factory's interface, SolverFactory, instead of the concrete base class. The base class is then loaded at run time using dynamic class loading.

4.3. Dynamic class loading

The Java language and virtual machine support the ability to load classes dynamically at runtime. For each class or type, the Java Runtime Environment (JRE) maintains an immutable Class object that contains information about the class. A Class object represents, or reflects, the class. Calling the newInstance() method of Class creates a new instance of the class. All that is needed is the fully qualified name of the class that you need to create. The combination of the factory pattern with dynamic loading is a powerful combination. It provides all the advantages of the factory pattern without its inherent limitations.
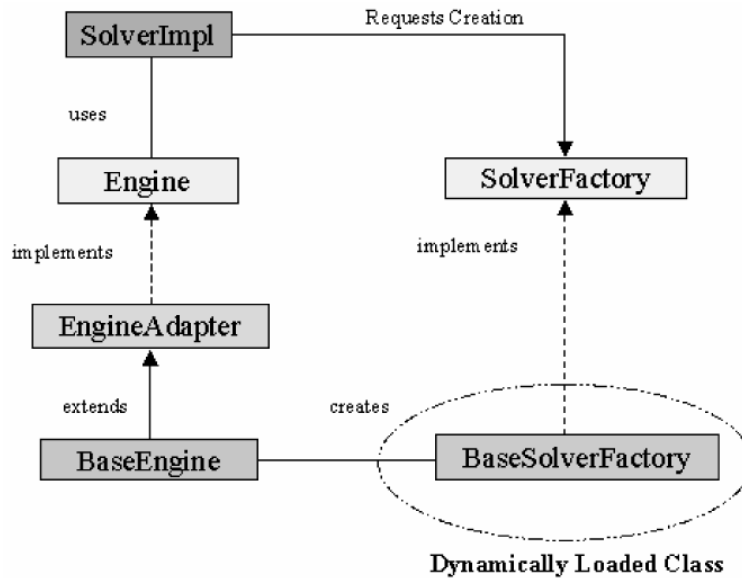
Fig. 6. Factory design pattern.

5. Testing and observations

5.1. Testing

Initial code development was done on a Windows NT platform and tested on a workgroup of NT machines. The whole environment was tested successfully on two SUN Solaris workstations using one of the INS2D test cases. Also, an early version of the environment was tested on an SGI workstation. The largest test run to date has one client and 12 servers processing 500 jobs.

Our environment is designed to replace the current environment used to run simulation codes as INS2D. The INS2D code typifies simulation codes running at NASA Ames Research Center, Moffett Field, California, and is the first code used for testing our environment. It solves the incompressible Navier-Stokes equations in two-dimensional generalized coordinates for both steady state and time varying flow [18]. These codes, mostly written in FORTRAN, are currently run using a complex collection of shell scripts that performs remote launching (batching), pre-processing and post-processing of data files. These steps include parsing input data files, moving them to the machine hosting the execution environment, executing the job, post-processing of the output files and moving these to a flat-file database. Job scheduling is performed through a specialized job scheduler. These scripts, of several thousand lines of code, are "boxed" inside one another. Post-processing is run from remote execution, which is run from pre-processing.

Part of the complexity of the scripts is due to having to run hundreds of jobs without operator intervention. Each job consists of a single grid with 1 to N angles of attack (alphas). For example, ten grids with ten alphas each means that INS2D must be run one hundred times. For each run, input files have to be created and/or moved to the target machine.

Under the current scripting system, all data assigned to a shell variable on a given machine must be written to a file before the job moves over. The file is transferred and must be re-parsed upon its arrival. Under the scripting system, sections of code are marked as "belonging" to a specific solver. Under our object-oriented system, any hard coded values needed by a given solver are listed in a Properties file. This keeps machine specific data on that machine. The values become known to the dispatcher when the solver registers. There is no a similar registration process with the scripts. All values are hard coded.

However, the job parameters and requirements are similar for both environments. The original FORTRAN binaries require that a job be started in the directory that contains the data and input files. All output files will be generated in that directory. After the jobs are done, the output files are moved to a storage location. If another job had to be run, the user would have to move to another sub-directory, copy over the data files, and change the parameters on the input files. This requirement naturally leads to a scripting system that automates this process. In this respect, the sub-directory system of

the execution environment has remained very similar to the shell scripts. A nested set of sub-directory (pad) is built on the machine hosting a solver.

5.2. Portability and performance

While all UNIX machines have shell programs, different shells have different commands. By using Java as the implementation language, our system is more portable than a shell system. By using CORBA as the communications layer, both platform and language independence are achieved. The components themselves are independent units. The CORBA objects can be written in any language that supports a CORBA IDL mapping, the worker objects can be pulled out and popped back in without recompiling the Java code of the other objects. Worker objects that have to communicate with each other do so through Java events. They are completely decoupled from each other. The binary that the solver runs is itself a property. As long as the nested subdirectory structure of the pad is compatible with a different binary, the code would not have to be recompiled; only the value of the property needs to be changed.

By working from a layered abstraction model, this system has several places where a future developer might "take over" an object. The given object classes could be extended. Because of using CORBA for the communication and control, the language independent CORBA IDL contract could be implemented in C, C++, or Smalltalk. By using Java as the main implementation language, platform independence is achieved.

One possible drawback of our approach is using Java due to low performance relative to a machine-level compiled code. In our applications, the time it takes to run a job is a function of the number of data points in the grid file. The vast majority of the run time is taken up by the number crunching done by the legacy flow solver binary.

Another performance factor is CORBA. There have been several studies of performance of CORBA. One of the early studies was the work by Schmidt et al. [19] where they compared the performance of two implementations of CORBA (Orbix from IONA Technologies and ORBeline from Post Modern Computing) with BSD Sockets. Their study shows that the overhead of early CORBA implementations is significant on high-speed networks like ATM or FDDI but less significant on low-speed networks like Ethernet and Token Ring. Another study by Fatoohi [4] shows that performance

of an early CORBA implementation (Orbix 1.3) lacks behind BSD Sockets but it is comparable to other communication packages such as PVM. In addition, Orfali and Harkey [16] compared the performance of CORBA VisiBroker for Java 3.1 with Java Sockets, Java RMI, Java Servlets, HTTP/CGI, and Microsoft DCOM using a simple Ping program written in Java (as well as in C++). Their results show that among these technologies only Java Sockets, using buffered data stream, outperforms CORBA in their test.

In summary, these results – which are quite dependent on many factors including hardware platforms, communication networks, and implementation languages – show that there is a penalty in using a high-level programming model like CORBA compared to a lower-level model like BSD Sockets and Java Sockets. But we believe that CORBA and Java overheads are insignificant in the total execution time of simulation codes that our environment is designed for.

6. Concluding remarks

We are currently in the process of applying our environment to a 3-D CFD code provided by NASA Ames. We do not anticipate major problems in dealing with other codes since our approach fits well with any type of processing where the job can be broken up into smaller units and then reassembled. We are also investigating the security aspect of our environment by focusing on two main components: access control and authentication. Among our options is incorporating security components from CORBA and Java.

In summary, we have designed and implemented an object-oriented approach for a job execution environment to run highly iterative jobs. Several technologies are employed: Java, CORBA, UML and software design patterns. Early results are quite promising. Our object-oriented environment is simple, flexible, and easy to upgrade and maintain.

References

[1] A.P. Campbell and J.R. Hummel, The Dynamic Information Architecture System: An Advanced Simulation Framework for Military and Civilian Applications, The Society for Computer Simulation International, *Simulation Series* **30**(4) (1998), 212–217.

[2] H. Casanova and J. Dongarra, NetSolve: A Network Server for Solving Computational Science Problems, *The International Journal of Supercomputer Applications and High Performance Computing* **11**(3) (1997), 212–223.

[3] Z. Chen et al., Web-based Framework for Distributed Computing, *Concurrency: Practice and Experience, Java Special Issue* **9**(11) (1997), 1175–1180.

[4] R. Fatoohi, Performance Evaluation of Communication Software Systems for Distributed Computing, *Distributed Systems Engineering Journal* **4**(3) (1997).

[5] S. Fine et al., The Environmental Decision Support System: Simulation Support and Information Management in an Integrated Computational Framework, Proceedings of Eco-Informa '96: Global Networks for Environmental Information, Vol. II, Lake Buena Vista, FL, 1996.

[6] I. Foster and C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, *Int. Journal of Supercomputing Applications* **11**(2) (1997), 115–128.

[7] E. Gamma et al., Design Patters, Addison-Wesley, 1995.

[8] M. Grand, *Patterns in Java,* (Vol. 1), Wiley, 1998.

[9] P. Gray and V. Sunderam, IceT: Distributed Computing with Java, *Concurrency: Practice and Experience, Java Special Issue* **9**(11) (1997), 1161–1168.

[10] E.N. Houstis et al., PELLPACK: A Problem Solving Environment for PDE Based Applications on Multicomputer Platforms, *ACM Trans. Math. Software* **24** (1998), 30–73.

[11] E. Houstis, J. Rice and E. Gallopoulos, *Enabling Technologies for Computational Science – Frameworks, Middleware and Environments,* Kluwer Academic Pub, 2000.

[12] Langtangen, *Computational Partial Differential Equations, Numerical Methods and Diffpack Programming, Lecture Notes in Computational Science and Engineering,* (Vol. 2), Springer-Verlag, 1999.

[13] M. Lewis and A. Grimshaw, The Core Legion Object Model, Proceedings of the 5th IEEE Int. Symp. On High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos, CA, 1996.

[14] M. Nibhanapudi and B.K. Szymanski, BSP-based Adaptive Parallel Processing, in: *High Performance Cluster Computing, Vol. 1, Architectures and Systems,* R. Buyya, ed., Prentice Hall, New York, 1999, pp. 702–721.

[15] OMG (Object Management Group), *The Common Object Request Broker: Architecture and Specification,* Revision 2.0, 1995.

[16] R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA,* (2nd ed.), Wiley & Sons, Inc., 1998.

[17] T. Quatrani, *Visual Modeling with Rational Rose and UML,* Addison Wesley, 1998.

[18] S.E. Rogers and D. Kwak, An Upwind Differencing Scheme for the Time Accurate Incompressible Navier-Stoke Equations, *AIAA Journal* **28**(2) (1990), 253–262.

[19] C. Schmidt, T. Harrison and E. Al-Shaer, Object-Oriented Components for High-speed Network Programming, *Proc. Conf. On Object-Oriented Technologies,* Monterey, CA, 1995.

[20] A. Vogel and K. Duddy, *Java Programming with CORBA,* (2nd ed.), Wiley & Sons, 1998.

[21] N. Warren and B. Bishop, *Java in Practice: Design Styles and Idioms for Effective Java,* Addison-Wesley, 1999.

[22] R.A. Whiteside, E.J. Friedman-Hill and R.J. Detry, PRE: A framework for enterprise integration, in: *Information Infrastructure Systems for Manufacturing II,* Mills and Kimura, eds, Kluwer Academic Pub, 1999.

Advances in
Multimedia

The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Hindawi

Submit your manuscripts at
http://www.hindawi.com

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering