An Object-Oriented Job Execution Environment[1]

Lance Smith & Rod Fatoohi[2]
San Jose State University & NASA Ames Research Center

Abstract

This is a project for developing a distributed job execution environment for highly iterative jobs. An iterative job is one where the same binary code is run hundreds of 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 scheduling system that will run a variety of jobs with minimal changes. Areas of code that are unique to one 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 CFD code, INS2D.

Key words: job scheduling, object-orientation, Java, and CORBA.

Introduction

Current solutions to the job scheduling and execution problem take two different approaches. The first requires dedicated job scheduling systems that route, execute and recover output. The second approach is a home grown scheduling environment written in a scripting language. Problems with the first solution are cost and portability. Obtaining licenses for all the available machines can be expensive and the scheduler may not work on all the available platforms. Scheduling systems of this type tend to favor high-end machines. The problem with the second is robustness. Home-grown solutions tend to work well at first, but as the job load grows, so does the script. Hard coded values become buried and the script looses flexibility, which is the reason why a scripting solution was chosen in the first place. Unless it was carefully engineered from the outset, it will eventually become a mass of arcane code that only the author can maintain.

The current trend in data processing is 3 tier and N tier solutions. Lightweight clients communicate with middle tier application servers. These, in turn, communicate with the backend databases or execution engines. New advances in platform independent languages and frameworks have removed many problems associated with porting code

---

[1] 0-7803-9802-5/2000/$10.00 (c) 2000 IEEE.

[2] Point of Contact: Prof. Rod Fatoohi - Computer Engineering, San Jose State University, San Jose, CA 95192. Email: rfatoohi@email.sjsu.edu

bases from one platform to the next. 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.

As a proof of concept, we design our architecture around highly iterative Computational Fluid Dynamics (CFD) codes. These CFD codes, or flow solvers, are numerically intensive applications that model airflow over wings or airframes. The scheduling system architecture is designed with maximum reuse in mind. One of the goals is to develop a solution that enables different CFD 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. In our implementation, we use Java due to its inherent platform independence. The Common Object Request Broker Architecture (CORBA) is used for communication and control between the tiers [2, 6].

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 the 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.

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. The proof of concept was to get a room full of desktop computers to share the processing of a batch of time and resource intensive jobs. There are other job-scheduling systems available, either commercially or free-of-charge, that are used in more production-type environments. Among these systems are Network Queuing System (NQS) originally from Sterling Software, Condor from University of Wisconsin, Portable Batch System (PBS) from NASA Ames Research Center, LoadLeveler from IBM, and Load Sharing Facility (LSF) from Platform Computing. They differ in type of job supported (serial or parallel), platform supported, tools provided, and many others. Our approach is unique since it is based on object-oriented design and analysis and uses CORBA and Java for implementation.

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. The current environment is briefly described first. Then the architecture is

introduced as a three-tier system. The objects in each tier are described in details. Finally, the results and some conclusions are drawn.

Typical Scenario for running CFD codes

The INS2D code typifies CFD codes running at NASA Ames Research Center, 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 [4]. It is currently run using a complex collection of shell scripts that performs remote launching (batching), pre-processing and post-processing of data files [5]. 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 job scheduler, such as PBS at NASA Ames. These scripts are "boxed" inside one another. Post-processing is run from remote execution, which is run from pre-processing. This strategy has the effect of leaving several processes sleeping while the job processing moves from stage to stage.

This collection of scripts evolved over a period of time. Script construction is linear. There are no procedures. Numerous *while* loops and *goto* like escapes make reading through them difficult. There are many hard coded values and sub sections for different types of platforms. Since scripts are stateless, collections of related data are written out to temporary files, which means all inter-machine communication is done at the file level. These files have to be re-parsed on the host machine in order to retrieve the data. If a fatal error occurs, all processing will grind to a halt.

The scripts represent a large undertaking in terms of design effort. They successfully process hundreds of jobs dealing with thousands of data files and variables. The problems with them are the problems that occur in all grown systems: fungus architecture is easy to break, hard to maintain, and almost impossible to understand. The more it grows, the harder it will be to change it.

Java/CORBA Architecture

Object Definition

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 implements or extends the one before it. This allows workers to be "unplugged" and replaced without code changes to the 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 Figure 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. *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.

IDL Files and Interfaces

There are two Interface Definition Language (IDL) files that define the interfaces and structures for this system [2, 6]. 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 *cfd.idl*, which defines a CORBA structure that is specific to a CFD 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 *cfd.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 CFD specific *Environment* object to be passed inside a more generic *Any*.

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). It transfers data from the Console to the *AdminAdapter* object as Java objects and from *AdminAdapter* to *Dispatcher* as CORBA *Any*.

- **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, 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 a CORBA *Any*.

- **Solver** – (3rd tier) represented by the *SolverImpl* object and requests collections of jobs from the dispatcher, builds the pads, transports the input and data files and either runs the job itself or passes it on to a sub scheduler. Initial values for the worker objects are obtained by reading a properties file, *BaseSolverProps.txt* at startup.

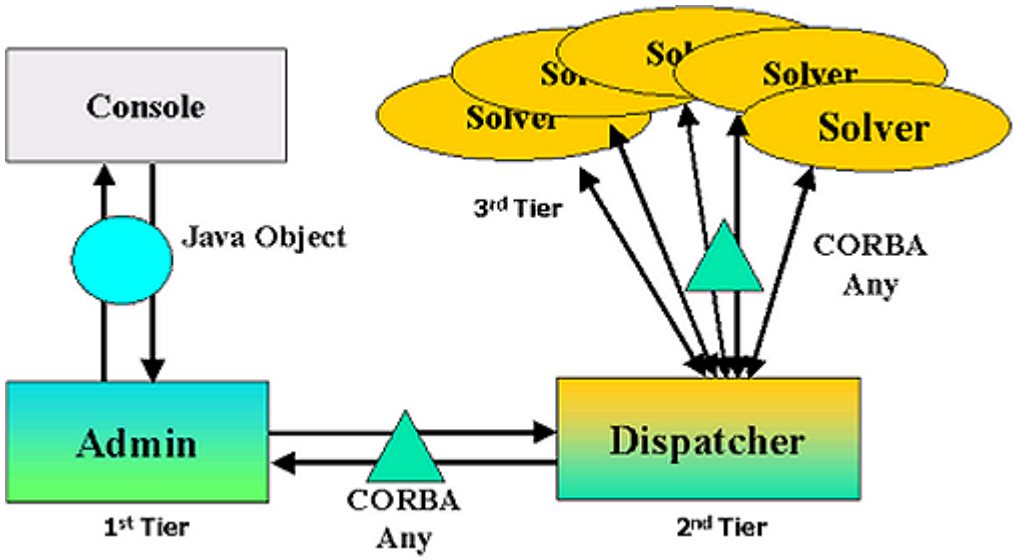

Figure 1. Architecture Framework

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), Dispatcher workers (at the 2nd tier), and Solver workers (at the 3rd tier). They are described below:

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.

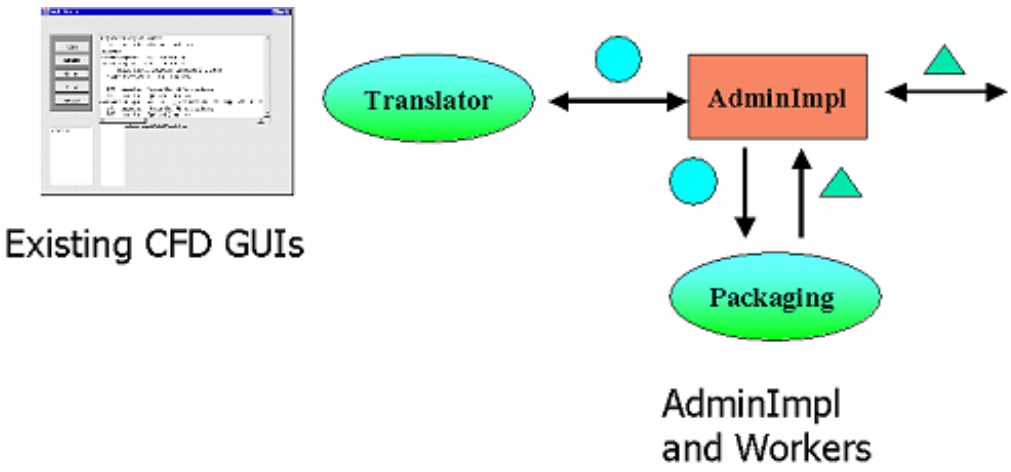

Figure 2. The 1st Tier.

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.

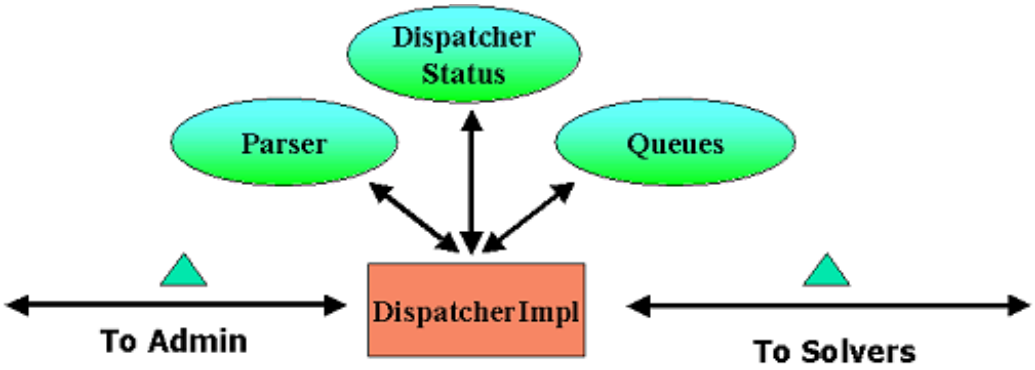

Figure 3. The 2nd Tier.

Solver Workers

- *Receiver* – accepts collections of incoming jobs and builds the execution pad. A pad is a unique set of directories that will hold 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 (at, cron)
- *Post* - handles job post-processing.
- *SolverStatus* - records and returns Solver and Job status

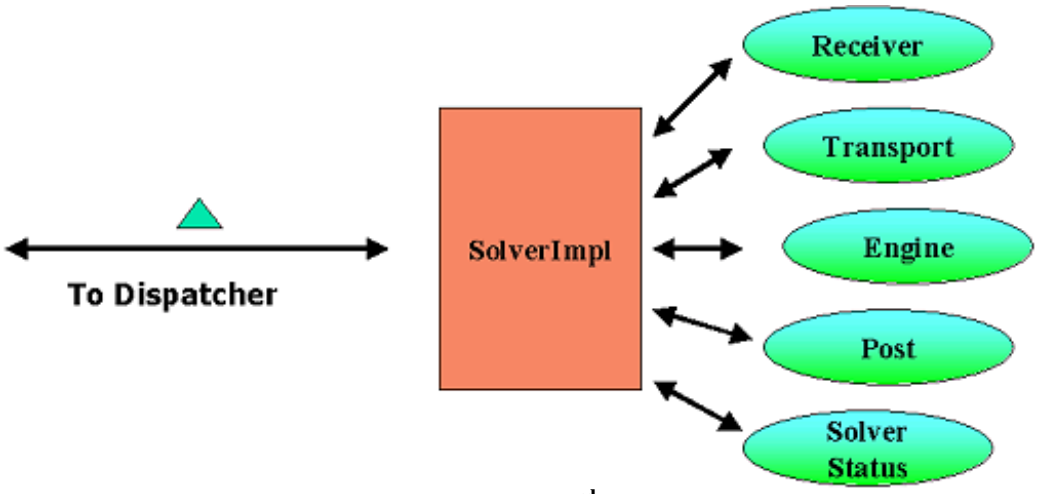

Figure 4. The 3rd Tier.

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

Layers of Responsibility

The architecture for *Admin, Dispatcher* and *Solver* interfaces is designed around four layers, as shown in Figure 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 is defined. All objects that interact with each other are referenced through their interface type. Manipulation 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. Specialization is used when a programmer wants to override the default values and operations provided in the initialization level.

Frameworks and Design Patterns

The framework has been designed with a visual modeling tool, Rational Rose [3], 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]. 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 [6]. In addition, we used the template method pattern and the factory pattern.

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 [1]. 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 it.

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 *Any*s) and are implemented in the base class.

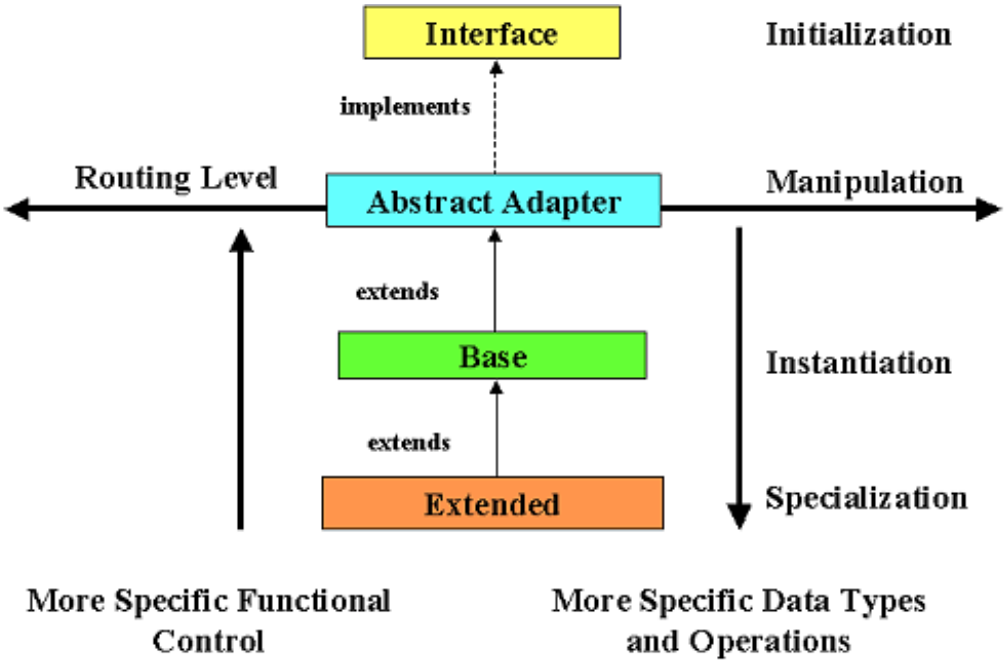

Figure 5. Layers of Responsibility

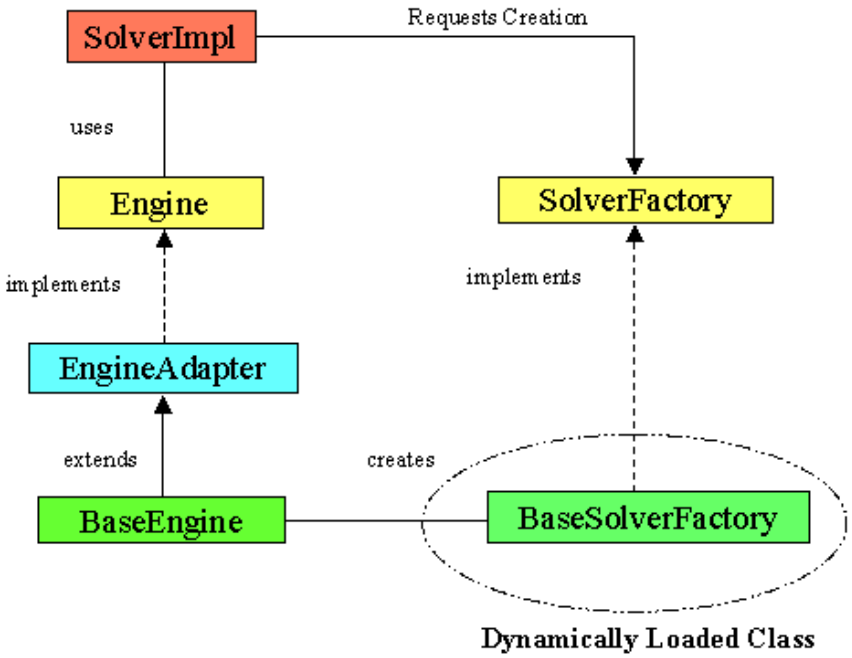

Figure 6. Factory Design Pattern

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.

Factory Pattern

In the factory design pattern [6], 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 Figure 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 runtime using dynamic class loading.

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 Class's *newInstance( )* method creates a new instance of the class. All that is needed is the fully qualified name of the class that you need to create. One small drawback to using dynamic loading is that the object being created must have a default constructor that takes no arguments. 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.

System Comparison and Observations

Our OO execution environment is not a direct replacement of a shell script that has several thousand lines of code; therefore, a section-by-section comparison between the two environments is not possible. There is no clean break between one script and the next. Under the 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 OO 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 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 file and grids. 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, called a Pad, is built on the machine hosting a solver.

Portability and Performance

While all UNIX machines have shell programs, different shells have different commands. There are only a few ports of UNIX shells to Windows environments. By using Java as the implementation language, our system is far 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.

The current price one must pay for using Java is poor performance relative to 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. Even though we do not have measured performance numbers (we are working on it), we expect that the amount of time the Java code runs is negligible.

Current Status and Concluding Remarks

Initial code development was done on NT and tested on a workgroup of NT machines. The whole environment was tested successfully on two SUN Solaris workstations, located at San Jose State University, using one of the INS2D test cases. Also, an early version of the environment was tested on an SGI workstation at NASA Ames Research Center. The largest test run to date has one client and 12 servers processing 500 jobs.

In summary, we have designed and implemented an object-oriented approach for a job scheduling and execution environment to run highly iterative jobs. Several technologies are employed: Java, CORBA, UML and software design patterns. Early results are quite promising. We have applied our solution to run a 2-D CFD code, INS2D. We are currently applying it to a 3-D CFD code, such as TIGER, provided by NASA Ames. We

are also looking into parallel CFD codes. We do not anticipate major problems in dealing with other, sequential or parallel, codes since our approach fits well with any type of processing where the job can be broken up into smaller units and then reassembled. Our object-oriented environment is simple, flexible, and easy to upgrade and maintain.

Acknowledgements

We would like to thank Mary Livingston of NASA Ames for her advice and recommendations throughout this project. We also thankful to Edward Tejnil of MCAT, Inc. at NASA Ames for his assistance in explaining the script and running some test codes. In addition, we are grateful to Stuart Rogers of NASA Ames for providing the INS2D code for our work. This project was funded through the Cooperative Agreement No. NCC2-1015 between NASA Ames and San Jose State University.

References

1. Grand, M., 1998. *Patterns in Java*, Vol. 1, Wiley Computing publishing.
2. Orfali, R. and Harkey, D., 1998. *Client/Server Programming with Java and CORBA, 2nd Ed.*, Wiley & Sons, Inc.
3. Quatrani, T., 1998. *Visual Modeling with Rational Rose and UML*, Addison Wesley.
4. Rogers, S. E. and Kwak, D., 1990. *An Upwind Differencing Scheme for the Time Accurate Incompressible Navier-Stoke Equations*, AIAA Journal, Vol. 28, No. 2, pp. 253—262, http://george.arc.nasa.gov/~srogers/#ins2d.
5. Tejnil, E., 1998. *User's Manual for the "INS2D Script System" (adtt/Eagle/Newton)*, MCAT, Inc., NASA Ames Research Center, Moffett Field, CA 94035.
6. Vogel, A. and Duddy, K., 1998. *Java Programming with CORBA, 2nd Ed.*, Wiley & Sons, Inc.
7. Warren, N. and Bishop, B., 1999. *Java in Practice: design styles and idioms for effective Java*, Addison Wesley Longman Limited.