

The “Restaurant Pattern:” A Framework for Developing Robust, Declarative, Loosely Coupled Enterprise Software Components

Dan Stieglitz
Principal Consultant
Stieglitech, LLC
dan@stieglitech.com

Abstract

Modern enterprise application environments have traditionally been (and probably will be for the foreseeable future) composed of disparate applications, technologies and data sources that are orchestrated in a meaningful way. Since technologies and products evolve gradually and are constantly changing, the requirement to integrate such components constantly challenges enterprise software architects. Loosely coupled environments are often built to facilitate building and maintaining such systems; most recently, such systems have been designed in a message-based environment. This paper discusses a software pattern for building robust infrastructure components in a loosely coupled, message-based environment. The pattern leverages techniques that create easily extensible and maintainable components that can be exposed simultaneously on multiple orthogonal interfaces.

Introduction

The first enterprise information technology departments, forming in an ad-hoc manner somewhere in the mid-1960s, had markedly less complexity to deal with in their business systems than we do today. At that time entire business systems were built and supported by a single vendor, the most prominent example being IBM. As the enterprise software industry evolved, however, a myriad new vendors and their products appeared on the market. To this day, the challenge of orchestrating business systems has been a driver for many new programming applications, systems and technologies. From the monolithic systems of early business computing to the development of distributed components, clustering, enterprise Java, and now Web Services and service-oriented architectures, integrating distributed systems together continues to be a major concern of all enterprises.

The advent of message-based systems is an attempt to provide a generic means of intra-system communication. Messaging is by no means a new software paradigm; most modern operating systems rely on messaging for inter-process communication. Unlike operating systems, however, businesses are rapidly changing entities that require interoperability within their enterprise and among other enterprises, all of which are adopting new standards and technologies constantly in an effort to reduce costs and

introduce new business models. Today, a rich set of technologies is available that give enterprises a smorgasbord of standards, tools and free software that allow robust message based systems to be built. A software pattern that can be used to build robust applications using these technologies in such an environment is discussed here.

Foundation Technologies

Java and XML

Java is one of the newest, most popular object-oriented development languages for (but not limited to) the enterprise. Java provides a very clean development model and an incredibly rich set of APIs out-of-the-box. The Java community is widespread and the amount of free information available to developers is staggering. Java's limitations include the fact that it is an interpreted language, and as such does not run as fast as its machine-language-compiled counterparts. However, the speed of modern business servers, where Java normally runs (as opposed to the client-side), allow Java-based applications to run at speeds suitable for very high loads.

XML, the eXtensible Markup Language, is a self-describing markup language that delivers an elegant solution for developing messaging protocols. XML, much like HTML, is an efficient way to deliver data and metadata (data about data) in a well-defined package. Additionally, a rich, mature standards-based toolset is available for operations on XML, including transformation of documents, search, and data query. Java APIs and free open-source toolsets for working with XML are easily available and include the popular Jakarta Xalan, Xerces, and XSLTC packages.

XML Schema and Data Binding

The latest addition to the XML standards toolset is XML Schema. XML Schema is essentially a language for describing XML documents, whose syntax is (not surprisingly) specified in XML itself. The schema-document relationship is not unlike the class-instance relationship found in OOP; indeed, these two technologies are very closely related. Now that rigorous definitions of documents can be specified, toolsets have been developed in Java (again, not surprisingly) that automatically generate and bind specific documents to Java classes.

The consequence of this is that it becomes trivial to define messages and their programmatic bindings simultaneously. This facilitates the design and implementation of message-based systems by abstracting the protocol development requirements to essentially one file: the schema itself. Developers who define protocols in XML Schema and use Java binding tools such as Castor or JAXB get the message parsers (and therefore some basic error handling and validation) for free [Castor, JAXB]. Using such toolsets allows for declarative systems, like the one described in this paper, to be easily built and maintained as well.

Java Reflection

One of the most interesting and powerful aspects of the Java programming language is the reflection API. Java, being an interpreted language, runs inside a “Virtual Machine,” (VM) which is responsible for loading, linking and executing classes. The reflection API exposes the class loading and linking mechanism to the developer, allowing classes to be loaded explicitly by the software developer at run-time from within programs. The advantages are numerous; for example, classes implementing particular interfaces can be written separately from the original code base, loaded to a production implementation after it’s been rolled out, and executed without any other changes. Additionally, design patterns like the GoF Command pattern are enhanced by the ability to add functionality (commands, in this case) in a similar manner as described above [Gamma]. The reflection API is a cornerstone of the restaurant pattern as well. There is a small performance price to pay when using Java reflection; indeed, this price has dropped recently. The first releases of the reflection API contained code that many considered too slow to roll out to a production enterprise system, but the performance of Java 1.4 reflection is much improved since the original releases.

Declarative Programming

Declarative programming refers to the practice of refactoring application logic out of the code and into a reference file. The Jakarta Struts framework is an excellent example of a declarative system: application execution path is determined by a configuration file that describes which pages to load upon specific user actions [Jakarta]. A declarative approach facilitates quick system maintenance and the ability to “hot swap” features of a deployed application. Declarative software systems are not without drawbacks, however, often there is much complexity in the type and number of configuration files required to successfully deploy and maintain such an application.

The “Restaurant Pattern”

The pattern described in this paper is called the “Restaurant Pattern,” but in fact is not strictly a design pattern as described in [Gamma]. Furthermore, the “rule of three” has not been met for this pattern¹, but for the sake of simplicity we refer to it as a pattern (“DeclarativeCommandStrategy with Adapters” seems a bit complex for a name). Also, as with most contemporary patterns, it consists of a composition of concepts and elements of other patterns.

¹ The rule of three states that some configuration of classes is a pattern when it has been successfully applied to three or more applications.

Pattern Goals

This pattern was born out of two basic requirements: (i) to be able to declaratively orchestrate a number of classes (think of it as a declarative command pattern or workflow system) and (ii) to be able to expose this orchestration from different software layers, such as a Web Service and a JMS destination simultaneously.

Pattern Architecture

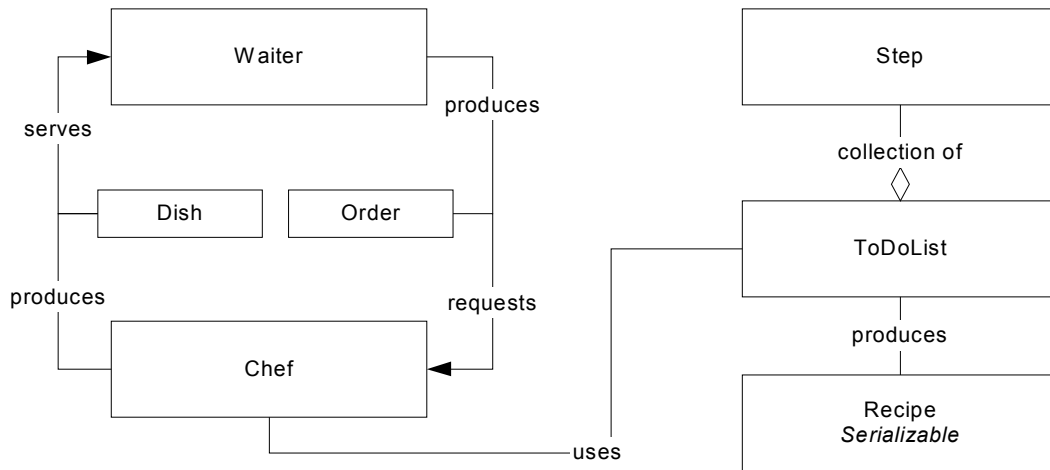


Figure 1: The Restaurant Pattern

Pattern Collaborators

Waiter: The Waiter is essentially a Facade to the underlying service. Each Waiter represents a different interface to the system, for example, `WebServiceWaiter`, `MDBWaiter`, `SocketWaiter`; depending on the application requirements. This also facilitates a single application to have multiple simultaneous gateways on disparate platforms. Each Waiter should implement a common business interface, although this is not a requirement. Waiters can be chained together to implement a flexible system of intercepting messages before processing. Such a chain is an effective way to implement basic security.

Order: An Order is a JavaBean that encapsulates all the information required by the Chef to produce the results (Dish). A generic set of methods for adding and retrieving named objects from the order is provided. JavaBean-style getter and setter methods can also be added if an introspector is to be used.

Dish: The Dish is a JavaBean that encapsulates the response produced by the Chef to be returned to the client. The dish can be passed along to the steps as a parameter to the

execute() method and in this manner be used as a shared data space (like a work in progress). This provides a single place for request state and allows Steps to be atomic and static, which facilitates building pools of Steps and ToDoLists to improve performance.

Chef: The Chef is the processing engine for the service. In concert with the ToDoList (see below), the Chef processes requests according to a specific algorithm. Different Chefs can use different algorithms for achieving goals (using the Factory pattern to swap Chefs) [Gamma].

ToDoList: A ToDoList is a list of tasks for a particular request. The Chef uses ToDoLists to determine what needs to be done and what has been completed. ToDoLists implement an iterative model and should act as a Factory for Steps. This is essentially an implementation of the Strategy pattern combined with Chain of Command [Gamma]. ToDoLists can be serialized and de-serialized by a Recipe object.

It is up to the application developer to determining exactly what manner ToDoLists return Steps to a Chef when requested; one option is to return a new instance of a particular step each time, another is to return pre-built Steps. Note that there are threading and multi-user implications for the latter paradigm. It is also up to the application developer to determine how ToDoLists interact with the main application. One potential implementation is to use a Singleton ToDoList that produces clones of a template step for each request; another is to assign each incoming request its own list. Again, threading implications exist for each paradigm and the developer should engineer a solution suitable for their requirements.

Step: The atomic component of a ToDoList, a step is “executed” by a Chef. Steps can be ordered, and dependency trees can be built. Steps also allow for excellent error localization, fail-over and error reporting capability. Steps should be atomic and static; that is, they should not maintain any state and instead use the Dish object to retrieve and store data. This model is similar to the JavaSpaces model, which describes objects operating on data in a shared space [Sun].

Recipe: A Recipe is the adapter between the Serialized version of a ToDoList (an XML file, or a text file) and the runtime version. Recipes produce ToDoLists when requested. Recipes know how to parse their serialized version to produce ToDoLists.

Potential Applications

XML Content Router

Messaging systems based on XML will doubtless become more prevalent in enterprise architectures as XML gains popularity. It is already the cornerstone of Web Services messaging, and XML Schema provides an excellent foundation for business metadata. Content routing refers to the passing of XML documents through some predetermined workflow, which is a common process for many businesses. Using XML and XSLT,

messages can be transformed to whatever format is required between endpoints. This technique can be used to glue different vendor-provided or home-built systems together.

A Restaurant-based XML content router would provide Waiters to accept messages (and possibly perform security), Steps that deliver and transform documents, and Recipes that describe the workflow for various processes. The workflows are loaded based on the incoming messages and are stored as XML files on the filesystem. As workflows change, new endpoints are added or modified, and new business processes are developed, the XML recipes can be modified to adjust the system to the new requirements.

XML Data Service

XML is slowly but surely claiming its stake as the most robust data transfer protocol for business application to date. Modern databases, however, are not as forthcoming when it comes to providing XML support and native-XML databases have not been as prevalent as vendors hoped. A compromise solution that many enterprises develop is a thin XML layer that sits on top of their existing databases. This layer accepts data requests and provides results in XML format. In between request and response, however, some non-trivial table-to-tree mapping must take place.

MVC Implementation for Service-Oriented Architectures

Leveraging an XML data layer or a set of course-grained Web Services, an MVC implementation could be developed based on the Restaurant pattern. Contemporary model-view-controller (MVC) implementations such as Jakarta Struts rely on JavaBeans to represent various parts of an HTML page and the underlying data (the view and model components) [Jakarta]. With XML and XSLT, the model components could be represented by XML documents (requested from the aforementioned XML data service or Web Services) and transforming these model components with some template HTML could render the view components. This technique is already used in some existing frameworks that allow the use of XSLT to render view components, such as Open Symphony's WebWork 2 [OpenSymphony].

Sample Java Bindings

This section provides some samples of the various components of the Restaurant pattern.

```
public interface Order {  
    /**  
     * Add some arbitrary object to the Order for use by the system  
     * e.g., name, request, message, etc.  
     */  
    public void addItem(String name, Object property);  
  
    public Object getItem(String name);  
  
    public List getItemNames();  
}
```

```

}

public interface Waiter {

    /* Used for chaining Waiters */
    public void setSuccessor(Waiter nextWaiter);

}

public interface Dish {

    /**
     * A dish has an environment that is used for storing request
     * State. Objects can be added and retrieved.
     */

    public Object getFromEnvironment(String name);

    public void putToEnvironment(String name, Object obj);

}

public interface Chef {

    public Dish cook(Order anOrder);

}

public interface Step {

    public void execute(Dish workInProgress);

}

public interface ToDoList {

    public boolean hasMoreSteps();

    public Step getNextStep();

}

```

Examples

The following is an example of a default Chef implementation. The application developer extends this class and implements methods for returning ToDoLists and Dishes. The cook() method here simple iterates through the steps, executing them and returning the final state of the Dish.

```

abstract public class DefaultChef {

```

```

public Dish cook(Order anOrder) {

    Dish aNewDish = getNewDish(anOrder);
    ToDoList workList = getWorkList(anOrder);

    while (workList.hasMoreSteps()) {
        Step nextStep = workList.getNextStep();
        nextStep.execute(aNewDish);
    }

    return aNewDish;
}

/**
Returns a new Dish object for processing. The Order object is
passed in to allow for order-specific dishes to be returned, for
example, a particular order may have a flag set indicating it
should be processed in some special way, and this method can
return an appropriate object.

*/
abstract public Dish getNewDish(Order anOrder);

/**
How ToDoLists are created is up to the application developer. This
method can return a pre-built list from a pool or create a new
instance each time. The Order object is passed as a parameter to
allow for order-specific lists to be returned; for example, a
particular order may have a flag set indicating it should be
processed in some special way and this method can insert order-
specific steps before returning the list.

*/
abstract public ToDoList getWorkList(Order anOrder);
}

```

Acknowledgements

Thanks to Matt Donovan for developing the original design of the pattern with me and to Daniel Honig for discussions on its potential applications in the real world.

References

- [Castor] <http://castor.exolab.org>
- [Guinee] Guinee, Kathleen, A Journey Through the History of Information Technology.
<http://www.cs.princeton.edu/~kguinee/Thesis/Computer.html>

- [Gamma] Design Patterns - Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison Wesley. 1995.
- [Jakarta] <http://jakarta.apache.org/struts>
- [JAXB] <http://java.sun.com/xml/jaxb>
- [OpenSymphony] <http://www.opensymphony.com/>
- [Sun] <http://java.sun.com/>

Copyright © 2004 Stieglitech, LLC. All rights reserved. Portions of this document contain material copyrighted by other parties. No portion of this document may be reproduced for commercial purposes without express written consent of the copyright holder. Reproduction for personal or academic use is permitted. All source code is distributed under the GNU General Public License (GPL). See <http://www.gnu.org/gpl> for a copy of the GNU GPL.