Java and UML: A Practical Application

Presented by

Paul R. Reed, Jr.

of

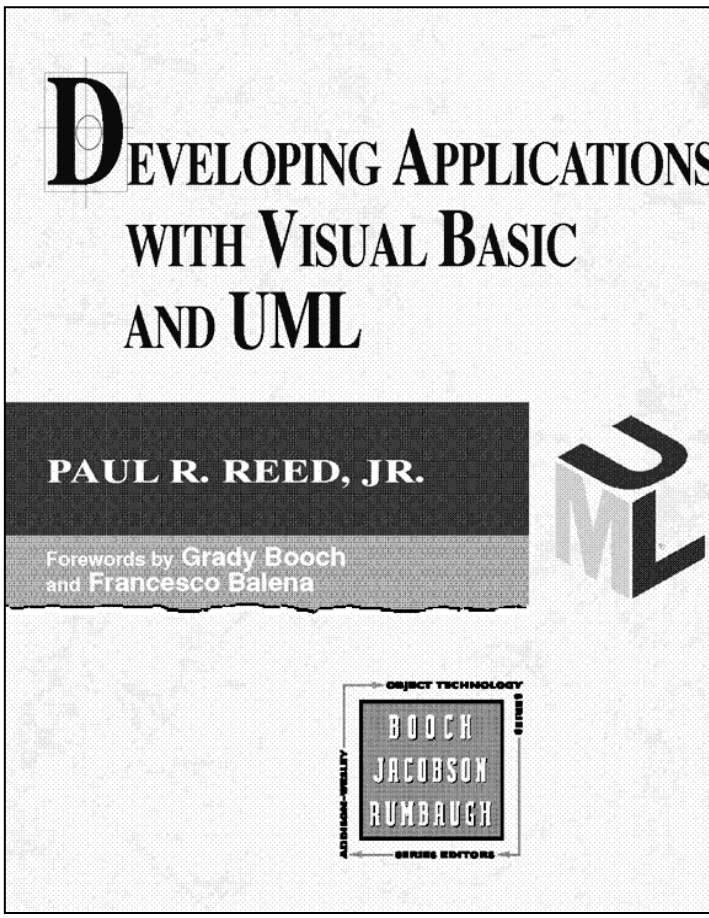
Jackson-Reed, Inc.

www.jacksonreed.com
prreed@jacksonreed.com

6660 Delmonico Dr.
Suite D-508
Colorado Springs, CO. 80919
888-598-8615 or 719-598-8615

Presenter Introduction

Jackson-Reed, Inc.



Presentation Objectives

- **Understand the principal components of the Unified Modeling Language (UML)**
- **Understand which UML diagrams give you the biggest payback**
- **Review the role of a process when utilizing the UML**
- **Discuss the Use Case Design Pattern**
- **Discuss the Model 2 architecture in non-EJB solutions**
- **Discuss the Model 2 architecture in EJB solutions**

This course takes a pragmatic approach to developing software with an Object-Oriented perspective. It is necessary, initially, to explore the benefits of Object-Oriented software development. It is also necessary to distinguish between Object-Oriented Design and our more structured brethren, Structured Analysis and Design.

The Unified Modeling Language (UML) is not new. It has been in the making for over 15 years. It combines the best work of many practitioners. This course provides a rigorous review of the UML, its diagrams and its constructs.

The UML would be a failure without a companion process model. Remember that a process model dictates “when we do what…and in what sequence”. The UML is just a notation. It is a way to describe the application domain. In this course, you will learn a representative approach to applying process to the application analysis, design and development process.

The UML can’t be everything to everyone. The UML purposely avoids platform specific issues such as User Interface Design and Database Design. However, applications wouldn’t be applications without either of these key components. This course covers these areas and presents approaches for incorporating them into the application analysis and design.

Goals of the UML

- **Provide users a ready-to-use, expressive visual modeling language, so they can develop and exchange meaningful models**
- **Provide extensibility and specialization mechanisms to extend the core concepts**
- **Be independent of particular programming languages and development processes**
- **Provide a formal basis for understanding the modeling language**
- **Encourage the growth of the OO tools market**

1-4

As will be noted later, the above goals are the cornerstone of the Unified Modeling Language. The UML is meant to begin to attack the root of our existing software dilemma, and that is to get the "blueprint" right. Just as Whitney found that firearms couldn't be built in mass production until the end-product was broken down into a collection of component diagrams, the software solutions of today and the future must also have relevant and meaningful diagrams.

The UML does just that. It provides a framework of 9 different diagrams that work together to spell out the "blueprint" of the application challenge at hand.

The UML is NOT

- **A systems development lifecycle - it doesn't dictate the steps to follow to accomplish a particular phase of a project (i.e., methodology based)**
- **A software process model - it doesn't dictate how a project should flow through which stages (i.e., spiral, waterfall)**

1-5

It is also key to note that the UML is just what it says, a modeling language. The UML does not tell the project team when to do what. What are some the questions the UML does not answer?

• When should I use which diagram?

• When should I not use a particular diagram?

• What is the recipe for successfully using the UML?

• Should we use incremental/iterative development or big bang?

• What is the translation of the diagrams into Relational Database design schemas ?

• What role does the company's architecture play in translating the analysis into design?

• Are there some diagrams that are better suited for analysis vs. design?

• Can I use UML diagrams to identify any issues dealing with distributed implementation of both data and process?

• What indications do I look for in prioritizing high-risk elements of the project?

Not to worry, this course addresses these issues in the recommended process model. Keep in mind that there are many process models available; some free, others for a hefty fee.

WITHOUT A PROCESS…YOU WILL FAIL!!!!

UML Components

- **The UML is comprised of 9 different diagrams**
- **These diagrams describe the system along different perspectives:**
  - **Static (use case, class, package)**
  - **Dynamic (use case, sequence, collaboration, state, activity)**
  - **Architectural (component, deployment)**
- **The UML does not dictate when something should be done or in what context (e.g. analysis vs design)**

1-6

The UML puts forward 9 different diagrams. These diagrams are broken into different perspectives:

- Static
  - Use Case Diagram (diagram itself)
  - Class Diagram
  - Package Diagram
- Dynamic
  - Use Case Diagram (underlying pathways)
  - Sequence Diagram
  - State Diagram
  - Collaboration Diagram
  - Activity Diagram
- Architectural
  - Component Diagram
  - Deployment Diagram

At a minimum, every project will produce Use Case, Class and Sequence diagrams.

What the UML Doesn't Address

- **The UML intentionally doesn't address:**
  - **Graphical User Interface**
  - **Architecture mapping**
  - **Object distribution**
  - **Object to Relational Mapping (if needed)**
  - **Network impact analysis**

1-7

Some practitioners fault the UML because it doesn't address certain aspects of the development life cycle. Some examples:

What about process and data distribution?

What about relational database design?

What about graphical interface design?

Remember, that one of the early goals of the UML was to be flexible and extensible. The UML would be tied to processor and platform issues if it were to deal with the above questions. It is important to note that the additional tools presented in this seminar draw directly from the output of the UML models reviewed in this course. The additional tools are all complementary to the UML, not evolutionary or revolutionary.

Sound Familiar

Jackson-Reed, Inc.

"We're really on a tight schedule and just don't have time for full-blown requirements"

Anonymous Manager

Sound All Too Familiar

Jackson-Reed, Inc.

"A Development Process only produces paper and gets in the way of the real reason we're here – to code "

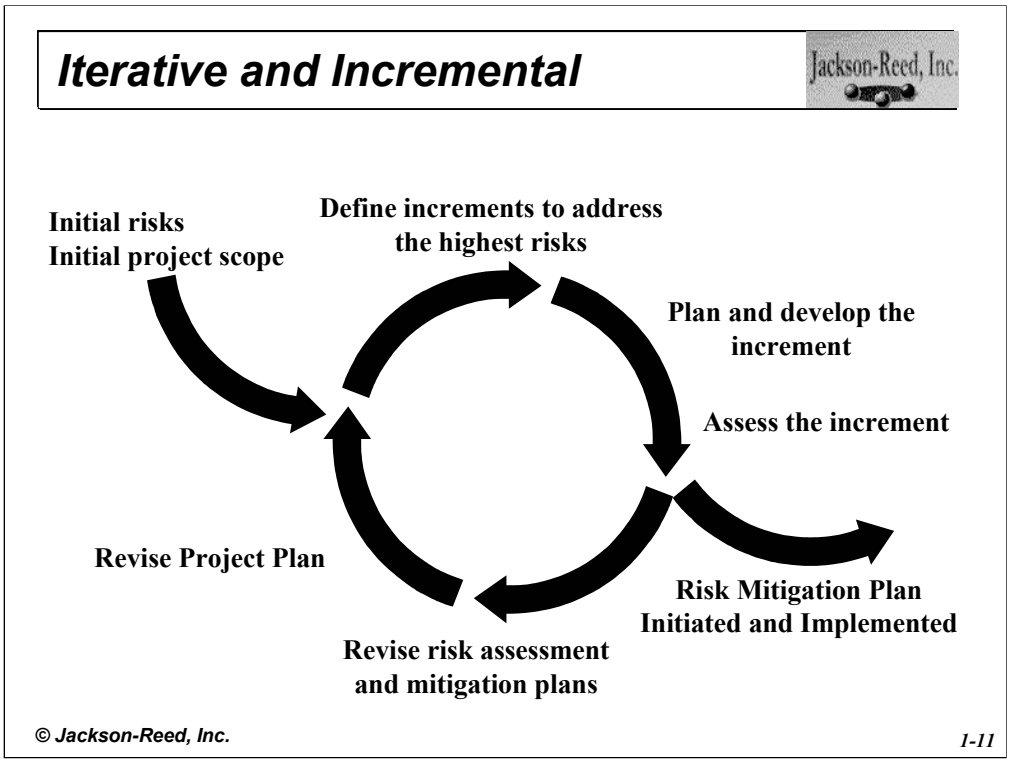
Anonymous Developer

Why Is It Then, That?

- **The Standish Group finds only 16% of projects today are completed on time, on budget, and inclusive of all promised functionality. The remainder are either considered "project challenged" or "project impaired"**
- **The ESPITI organization traced 54% of "Major Software Problems" to either poor requirements or lack or requirements management**

Source: Standish Group
European Software Process Improvement Training Initiative

1-10

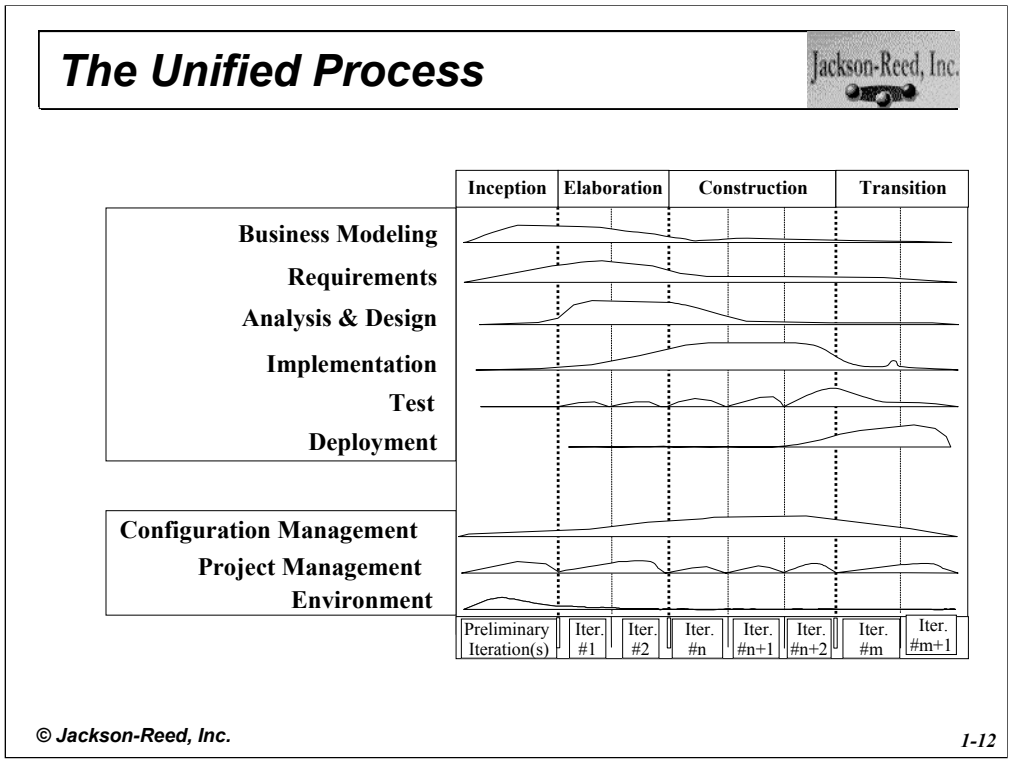


In an iterative and incremental life cycle, development proceeds as a series of iterations that clarify the problem, leading to a deliverable, or an increment, of the final system. Iteration relates to refinement, while increment relates to added functionality.

Each iteration consists of one or more of the following process components: requirements capture, analysis, design, implementation and test. The developers don't assume that all requirements are known at the beginning of the life cycle; indeed, change is anticipated throughout all the phases. The result of a successful iteration is another increment of the system.

This type of life cycle is called "risk-mitigating". Technical and business risks are assessed and prioritized early in the life cycle, and are revised during the development of each iteration. Risks are attached to each iteration, so that successful completion of the iteration alleviates the risks attached to it. The releases are scheduled to ensure that the highest risks are tackled first.

Building the system in this fashion exposes and mitigates the risks of the system early in the life cycle. The result of this life cycle approach is less risk, coupled with minimal investment.



The process model for RUP initially looks confusing.

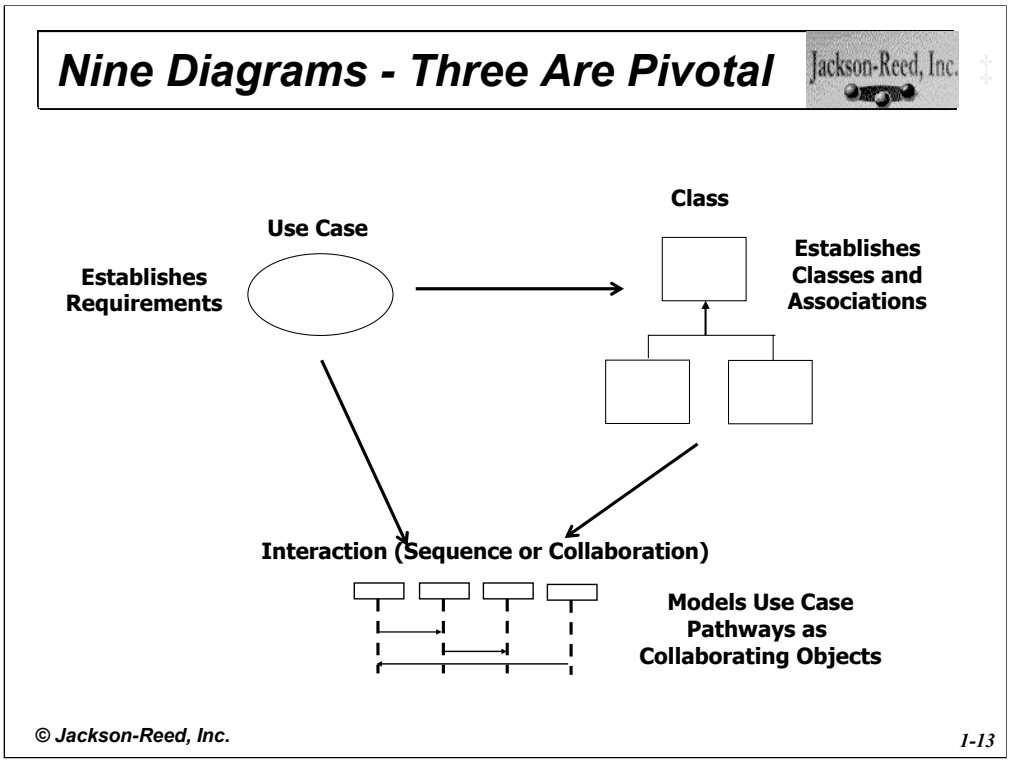
The easiest way to view the process is to think of phases as chunks of work, resulting in the completion of a milestone (i.e., Inception: Lifecycle Objective). The workflows represent activities (schedulable work) that we iteratively apply to complete an iteration.

For instance, If you view the vertical Inception phase, scan below it and you will see varying degrees of importance placed on each of the workflows. You see more emphasis in activities that are related to business modeling and requirements than you do in analysis and design, implementation and test.

However, you could have some activities being applied in these workflows during inception if you were doing some high-level prototyping perhaps just to stimulate ideas (IKIWISI – I'll know it when I see it).

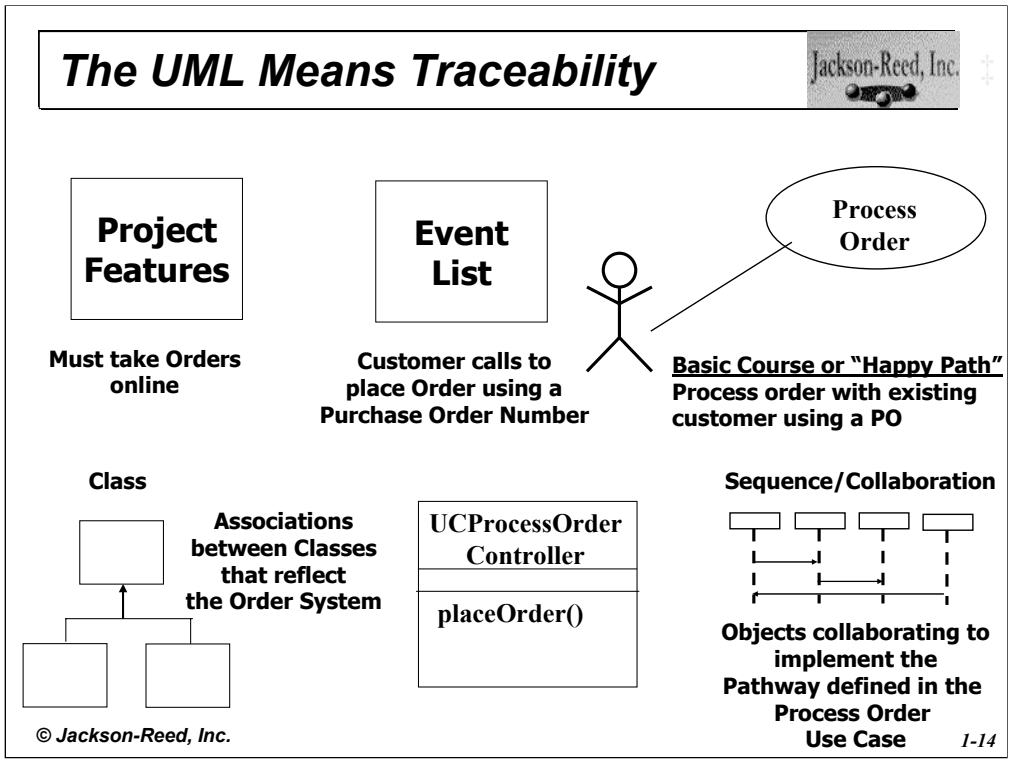
There are two types of project plans in RUP. One is a macro plan based on the phases, called the phase plan. The other are more micro plans and there will be one for every iteration, called the iteration plans.

A project will have have only one phase plan but many individual iteration plans.

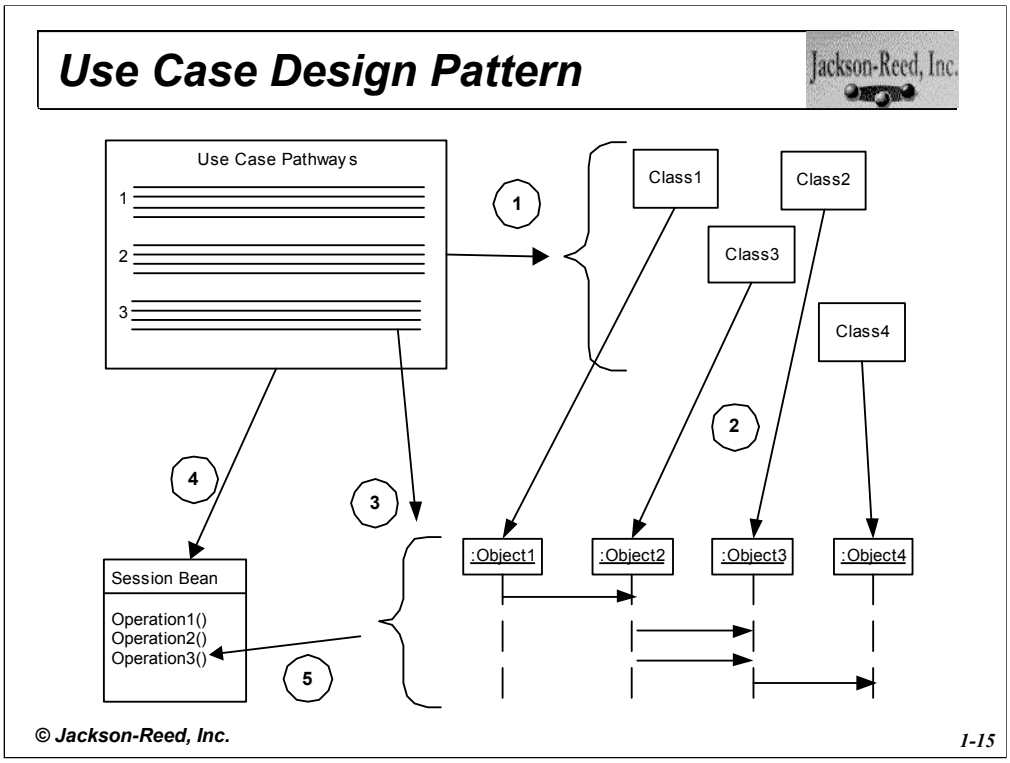


Don't think that you will never use other diagrams, but, from a practical perspective, the most mileage comes from those represented above.

Remember, most visual modeling tools today only generate the code skeletons from the class diagram.



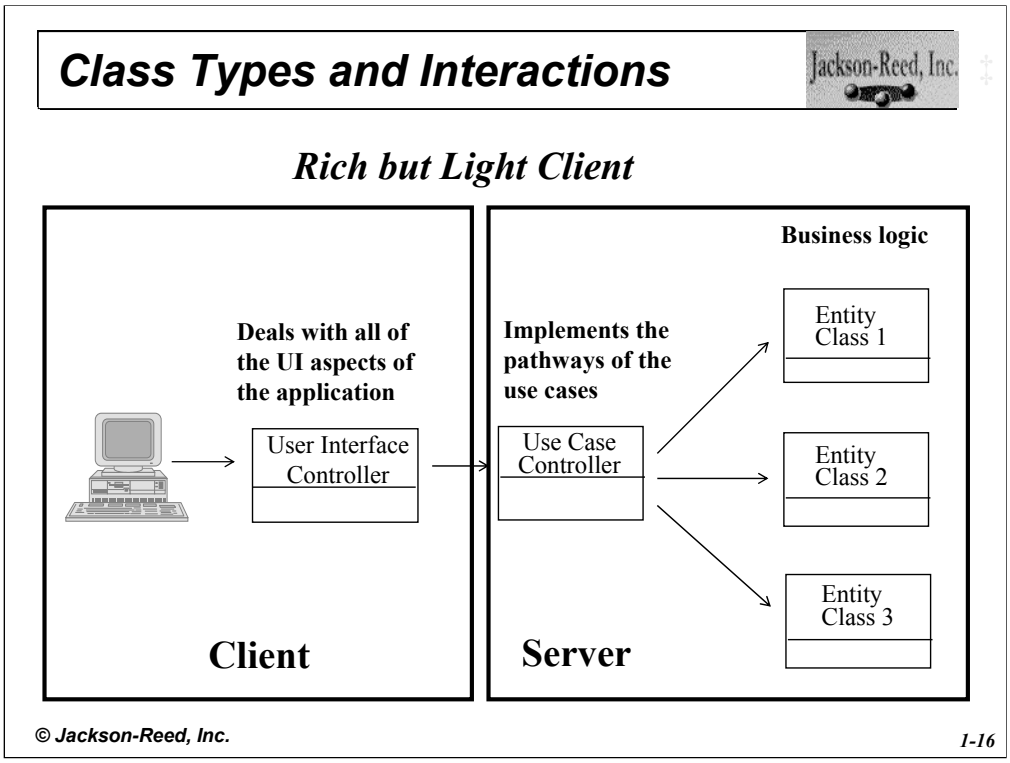
Many times we wish we could trace a business requirement to its eventual realization as a software unit. Other times, it would be beneficial to trace a software unit back to its business requirement. With the UML, there is finally a method to the madness of ensuring traceability.



Source: Paul R. Reed, Jr. from "Developing Applications with Java and UML", Addison-Wesley, 2002.

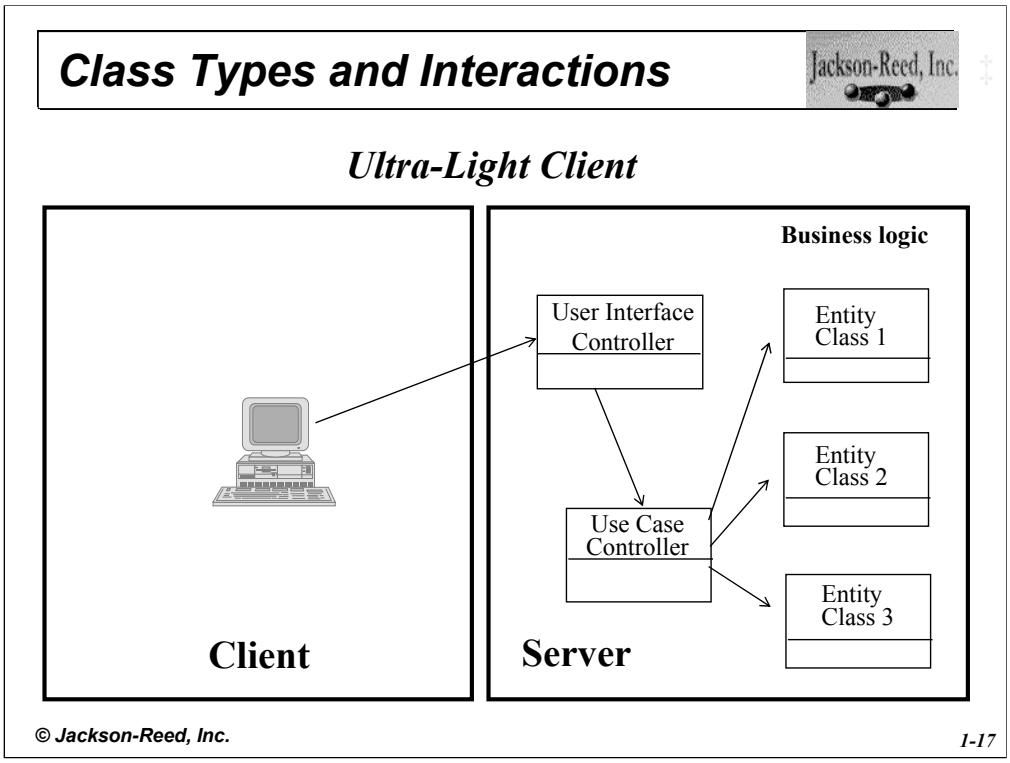
The above use case design pattern is reflecting an architecture of Enterprise Java Beans. However, it applies to any architecture you choose. There is an order to the progression of mapping to components: (1) From use-cases we find our entity classes (2, 3). (4) From the use-cases we create interaction diagrams (e.g., sequence/collaboration) that model our classes now acting as living objects sending messages to one another with the sole purpose of accomplishing the goal of the use-case pathway. (5) For each use-case, a use-case controller will be implemented as either an SFSB or an SLSB. This use-case controller bean will contain operations that implement the orchestration logic to carry out an individual interaction diagram. Again, these map directly to the use-case pathways.

This pattern also quite successfully predicts how transaction boundaries are distinguished. Many times I see projects struggle with where to place the logic that controls the boundaries for the pathways through the application. With the use case pattern the guesswork is removed. All transaction demarcation occurs within the use case control classes, ALWAYS. In the case of EJB applications, each method within each use case session bean is assessed as to its need for a transaction. All update-oriented operations in the session beans will be marked "RequiresNew". All other operations in the session beans will be marked "Supports". All entity beans can be marked at the bean level as "Supports".



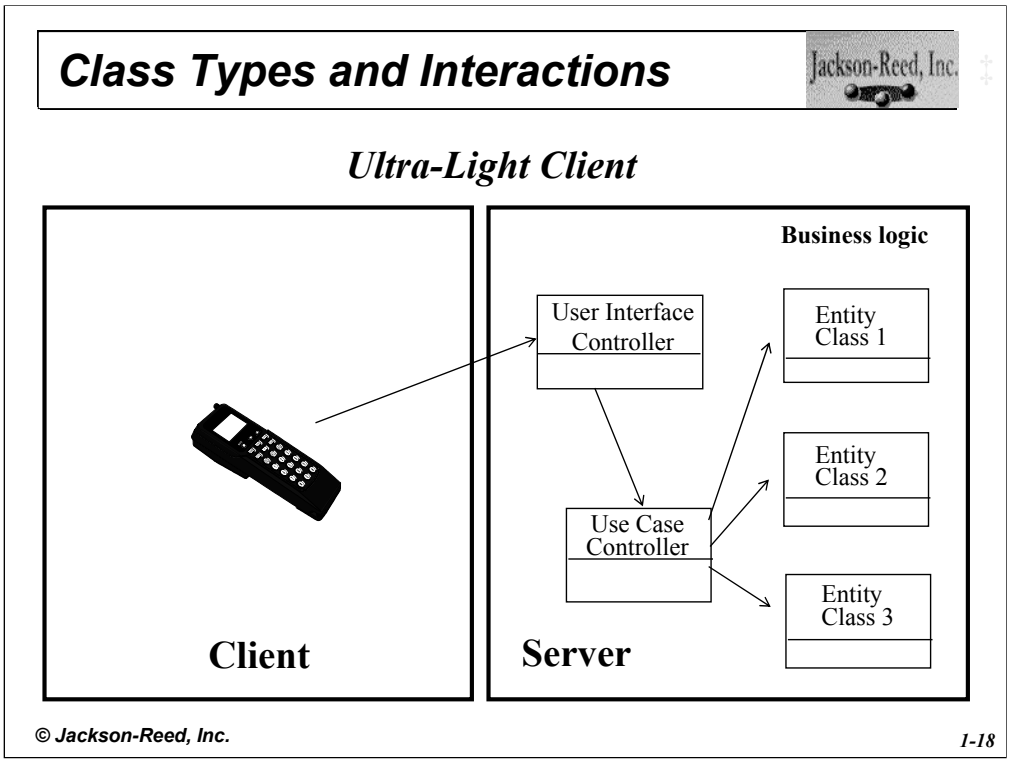
Using and applying the concepts of boundary (interface), controller, and entity (domain) classes renders the application's architecture much more resilient.

Typically, the GUI-specific logic and the user interface controllers will reside on the client. The use case controller and entity classes will be deployed on an application server.



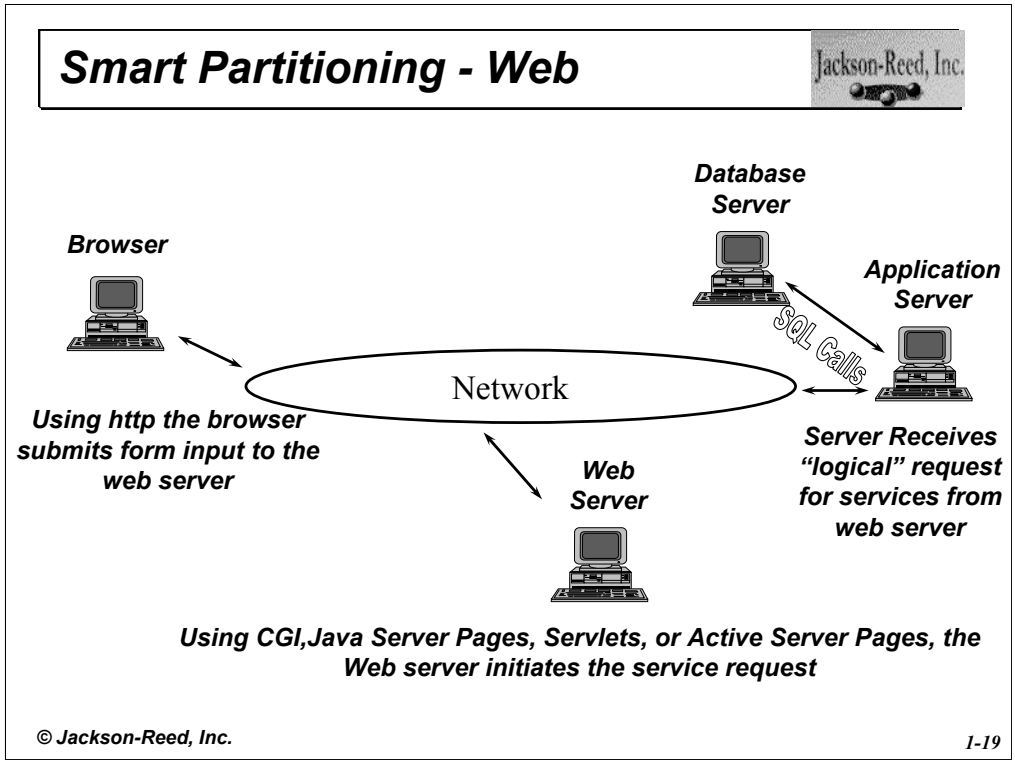
Using and applying the concepts of boundary (interface), controller, and entity (domain) classes renders the application's architecture much more resilient.

Typically, the GUI-specific logic and the user interface controllers will reside on the client. The use case controller and entity classes will be deployed on an application server.

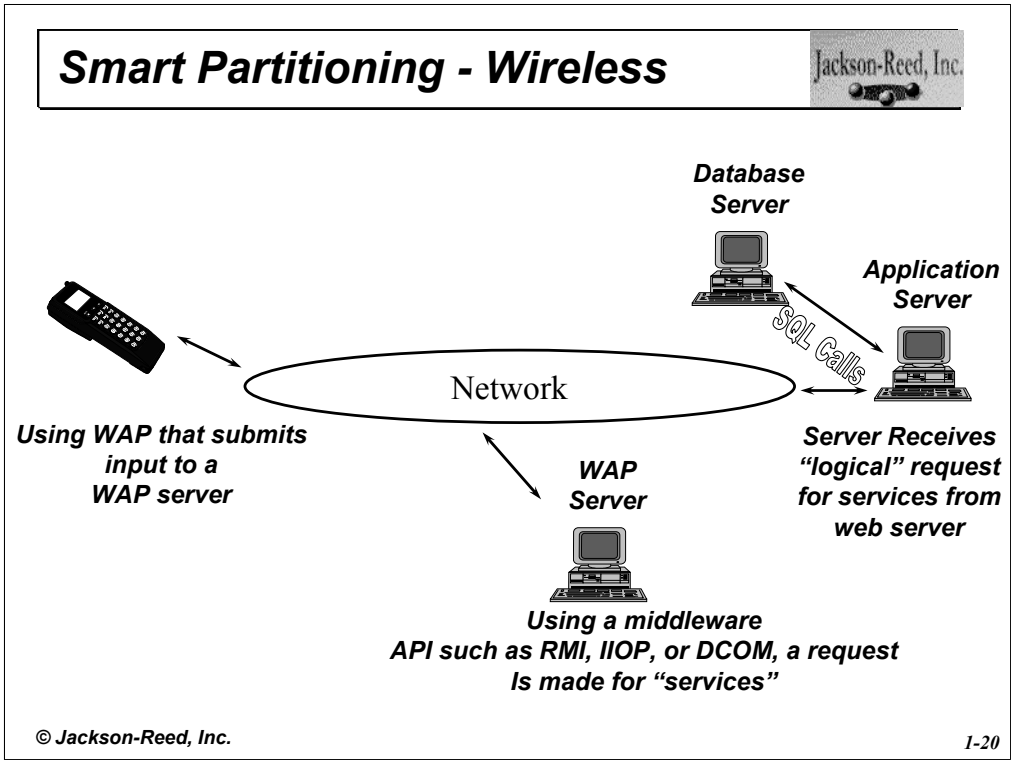


Using and applying the concepts of boundary (interface), controller, and entity (domain) classes renders the application's architecture much more resilient.

Typically, the GUI-specific logic and the user interface controllers will reside on the client. The use case controller and entity classes will be deployed on an application server.

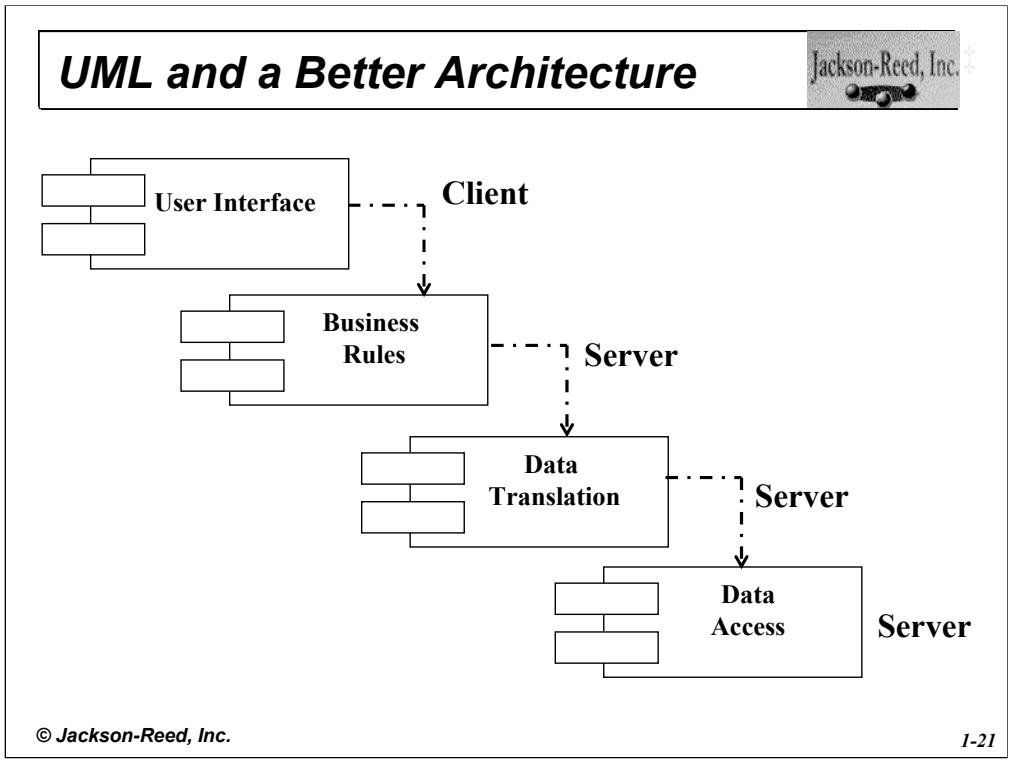


Who would have thought five years ago that a thin client would be the rage and that the client of choice would be a browser? Many client/server applications were thrown into the "legacy" bucket when this occurred because of the tight coupling between the presentation and business logic.



With the advent of new presentation technologies, it becomes even more important to separate the layers. With this model, the business logic doesn't care who requests its services.

In the case above, Wireless Access Protocol (WAP) is used to send requests to a WAP server which then talks to the business layer. The powerful statement here is that the same application server can be serving up traditional client-centric applications, web applications and wireless applications.

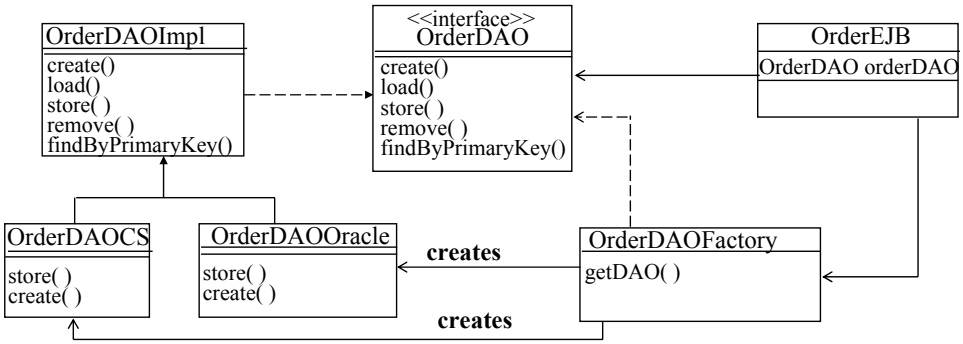


Remember that components are the realization of the classes that have been nurtured through the analysis and design process. Depending on the language and operating system that will implement the design, a component may implement either one class, or hundreds of classes.

In the Microsoft world, components can be EXEs and DLLs; in the case of Java, they may be just ".class" files.

GOF - Factory Pattern

- **This pattern abstracts the creation objects that aren't known until runtime**
- **Candidates for usage are in situations where *if-then-else* logic is being used to determine a service or functional component to include**



Factory is a classic pattern to mask the creation of objects. Polymorphism isn't appropriate here as it is only usable "after" objects are created. In this case, we don't know until runtime which object to create.

In the example above, the client needs to be able talk to a database. However, the particular database back-end is unique at each location (common in packaged software applications like PeopleSoft or SAP). To avoid a cascading if statement to cycle through which database API to use, the Factory pattern is employed to provide a level of abstraction. DAOIF is an interface which defines four operations.

Here is what happens:

• Client (OrderEJB) asks OrderDAOFactory to return an object which implements the OrderDAO interface so it can do database activities. The returned object is stored in the Client's orderDAO local variable which is of type OrderDAO.

•OrderDAOFactory determines which database API to use (either via a parameters file or through a JNDI lookup) unique to this location.

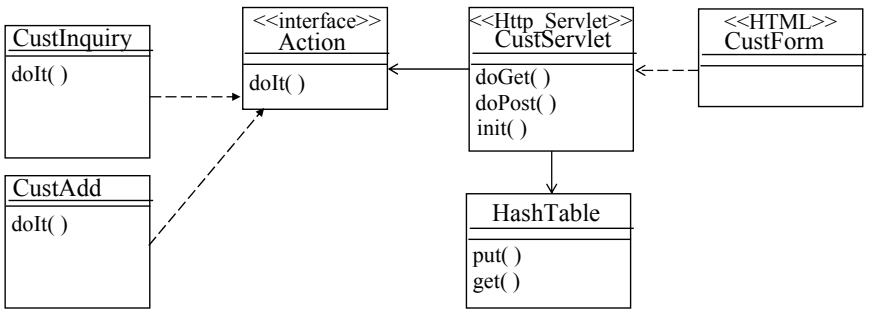
•OrderDAOFactory instantiates either the concrete class OrderDAOCS or OrderDAOOracle and returns this reference typed as OrderDAO.

•Now the Client messages to its local orderDAO attribute requesting it to create( ), load(), store( ), retrieve( ), remove( ), findByPrimaryKey() and the appropriate concrete class (OrderDAOCS, OrderDAOOracle) actually receives the messages.

GOF – Command Pattern

Jackson-Reed, Inc.

- **This pattern creates classes for each unique "action" that a program must handle.**
- **Candidates for usage are in situations where an action occurs and *if-then-else* logic is being used to carry out the work**



The command pattern is very applicable in both GUI applications (implementing do/undo scenarios) and in the web. In the case of the web, it is a very powerful technique to simplify servlets and the processing of unique actions instigated via HTML input. The premise is that every action (command) that a servlet must respond to has a corresponding class defined for it that implements the Action interface. The Action interface has one operation, called doIt( ). Every unique command implements this interface. The doIt( ) operation is typically mapped, one-to-one, with each use case pathway. Here is what happens:

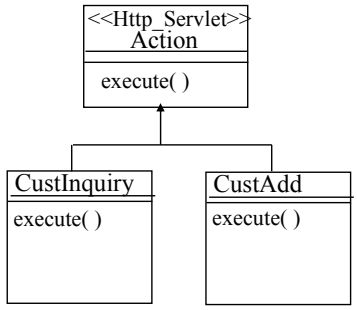
Html form posts to the servlet, CustServlet its input. One hidden field that is posted along with user input is an "Action" field that contains a hardcoded action id.

The servlet issues a get( ) on a hashtable using the "Action" id as the key and retrieves its related object, typed as an Action object. The hashtable can be built one of two ways 1). in the init( ) method of the Servlet (downside is that the action id/object pairs are hardcoded). 2). As servlet initialization parameters or as an external XML file that is traversed. The benefit of option 2 is that new actions can be added that the servlet must support without recompiling any code.

CustServlet invokes the doIt( ) operation on the object returned from the hashtable.

Command Pattern - Framework

- **The "struts" framework from the Jakarta group delivers a pre-cooked command pattern framework ideal for routing request specific actions to the appropriate handler**
- **Candidates for usage are in situations where an action occurs and *if-then-else* logic is being used to carry out the work**




```
<struts-config>
  <action-mappings>
    <action
       path="/inquiry"
       type="CustInquiry" />
    <action
       path="/add"
       type="CustAdd" />
  </action-mappings>
</struts-config>
```


With struts, the control servlet comes in the framework. All the developer has to do is add the entries to the STRUTS-CONFIG.XML file and build the handler class with subclasses the "Action" class provided by Struts. In the future, when a new action needs to be supported, no recompiles are required. Simply add the entry to the XML file and install the new subclass.

Model/View/Controller - Extended

- **The *Model* represents the entity classes with their contained attributes and behavior**
- **The *View* represents the external representation of information found in the model**
- **The *User Interface Controller* acts as the broker of external stimulus between the View and the *Use Case Controller***
- ***The Use Case Controller* orchestrates the messaging as designed in the sequence/collaboration diagrams. These "interaction" diagrams are a direct mapping to the pathways found in the use cases**

1-25

The two controllers are important to further isolate the application from future change.

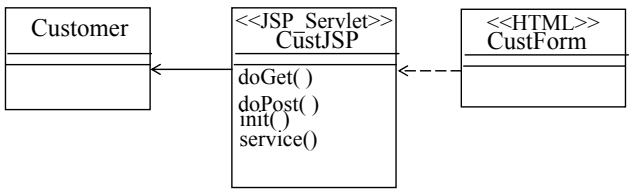
The use interface controller translates all the GUI interaction aspects of the Actor's session into technology neutral requests. That is to say, someone entering a Customer Id into a form and clicking the submit button must be repackaged into a message to some object to do some work in retrieving the Customer.

The use case controller contains the operations that implements the messaging to all the entity classes to satisfy the goal of the actor. It is also the use case controller that starts and ends transactions. They govern the concept and management of the "unit of work".

Java Server Pages – Model 1

Jackson-Reed, Inc.

- **Model 1 was the initial technology use of Java Server Pages in Web architectures**
- **HTML is posted to JSP pages which not only provided View support but also acted as the controller**
- **This will lead to bloated JSP pages and a component that has very low cohesion**



Remember that a Java Server Page is compiled into a Servlet at runtime. Notice that the stereotype on the CustJSP is <<JSP_Servlet>> to reinforce this. The difference to the developer is that they have no knowledge of the manipulation of the Servlet operations.

JSPs in the Model 1 scenario can also message to both Java Beans as well as Enterprise Java Beans. The biggest negative of the Model 1 approach is that control logic and display logic (Controller/View) are commingled.

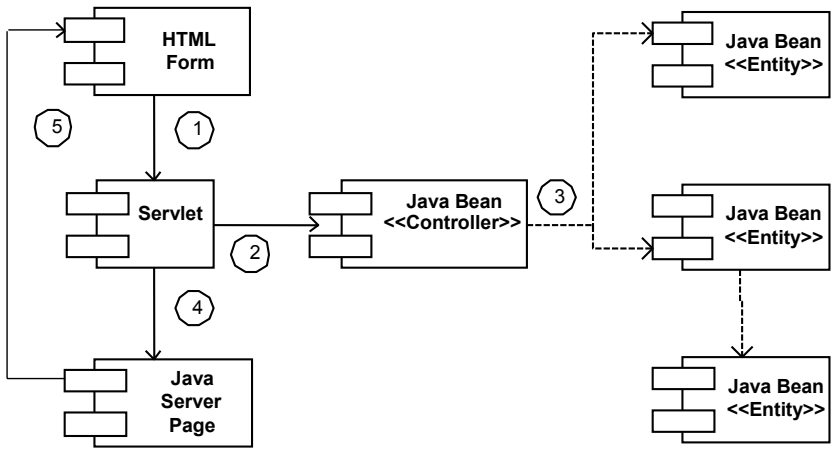
Java Server Pages – Model 2

- **Model 2 is the current technology architecture used on the Web today**
- **HTML is posted to a Servlet (user interface controller) which then messages to a Java Bean or Session EJB (use case controller) to satisfy the goal of the actor**
- **Each component is highly cohesive**

Model 2 is by far the most common JSP architecture found today.

The next few slides show example scenarios.

Model 2 with Java Beans

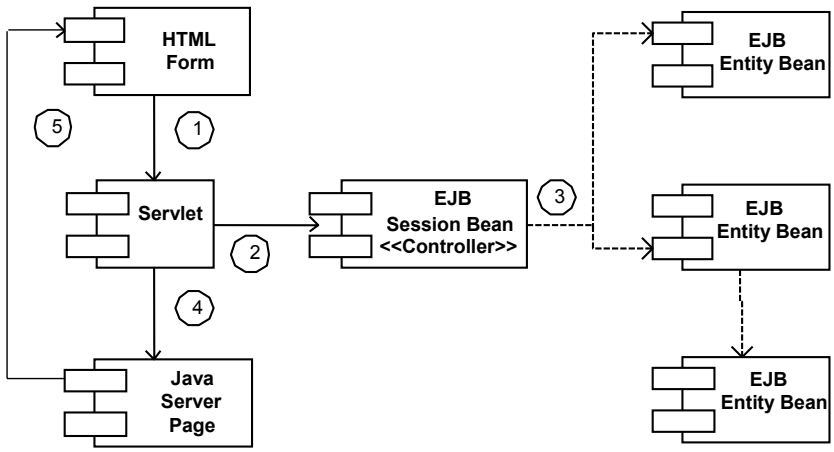


UML Component Diagram

Source: Paul R. Reed, Jr. from "Developing Applications with Java and UML", Addison-Wesley, 2002.

1. Html form requests a resource which is mapped to a Servlet.
2. The Servlet instantiates a use case controller class (Java Bean). The Servlet, based on the "action" requested by the form, sends a message to the use case controller class.
3. The message invoked in the use case controller class implements the messaging outlined in the sequence/collaboration diagrams.
4. A representation (these may be Value objects which attribute-only objects representing information to be displayed in a display neutral format) of what is to be displayed finds its way back to the Servlet as a result of the work done by the use case controller. This object or objects are inserted into the request scope of the Servlet. The Servlet then forwards the request to the appropriate Java Server Page.
5. The Java Server Page, using the information recently placed in the Servlet's request scope, formats a return page bound for the requesting browser.

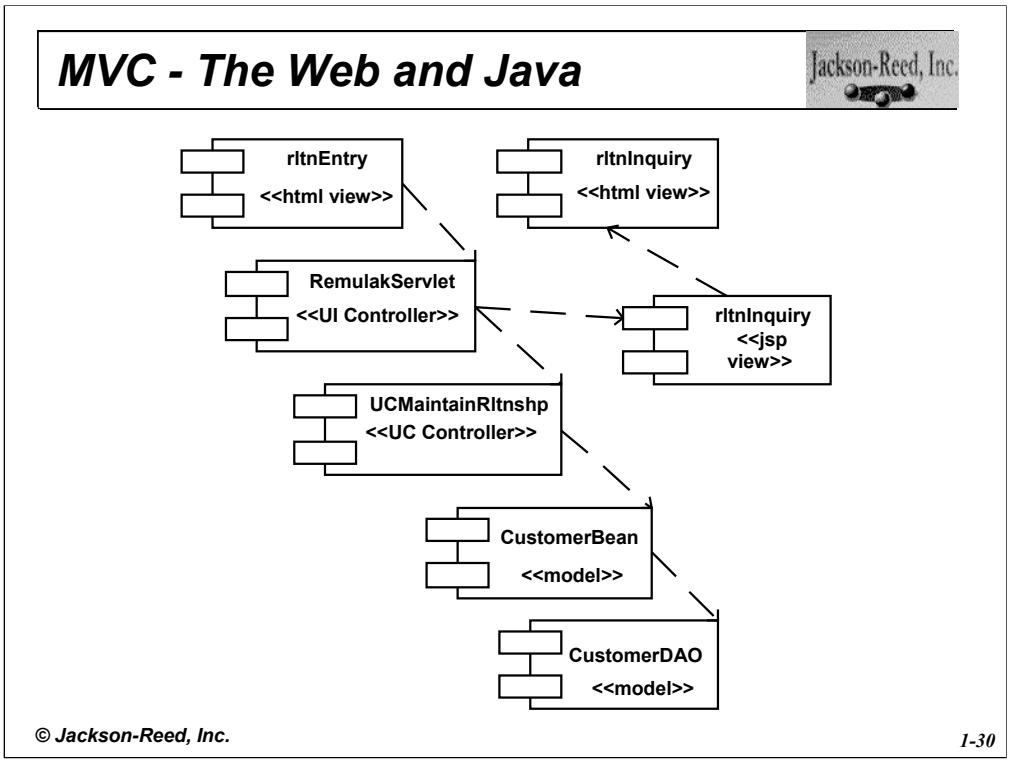
Model 2 with Enterprise Java Beans



UML Component Diagram

Source: Paul R. Reed, Jr. from "Developing Applications with Java and UML", Addison-Wesley, 2002.

1. Html form requests a resource which is mapped to a Servlet.
2. The Servlet instantiates a use case controller class (Session EJB). The Servlet, based on the "action" requested by the form, sends a message to the use case controller class.
3. The message invoked in the use case controller class implements the messaging outlined in the sequence/collaboration diagrams.
4. A representation (these may be Value objects which are attribute-only objects representing information to be displayed in a display neutral format) of what is to be returned finds its way back to the Servlet as a result of the work done by the use case controller. This object or objects are inserted into the request scope of the Servlet. The Servlet then forwards the request to the appropriate Java Server Page.
5. The Java Server Page, using the information recently placed in the Servlet's request scope, formats a return page bound for the requesting browser.



Source: Paul R. Reed, Jr. from "Developing Applications with Java and UML", Addison-Wesley, 2002.

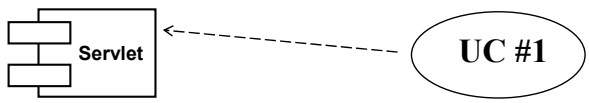
This is an example of an MVC pattern in practice for a web application.

The dependency relationship between RemulakServlet and the rltnInquiry() JSP is on the return trip back to the browser. Note that with a well-designed MVC architecture, simply changing the outbound JSP page to something else, perhaps one that returns Wireless Markup Language (WML) bound for a wireless PDA, would require absolutely no changes to the controller or the model components. They are none the wiser. Their role is simply to return a display-neutral object representing the customer. It is the role of the view to transform that object into something desired by the user.

Notice that there are two types of controllers in our architecture: a user interface controller and a use-case controller. The user interface controller is responsible for dealing with unique interface architecture (e.g., Web, wireless, voice response unit). The use-case controller is responsible for implementing the pathways defined in the use-cases and eventually modeled by sequence diagrams. The use-case controller doesn't care who requests its services or what technology ultimately delivers it.

Single Use Case Per Servlet

- **One Servlet per use case is a more natural mapping of components**
- **Traceability is enhanced should the functionality in the use case change in the future**
- **This allows high cohesion and lower levels of regression testing should the application change in the future**

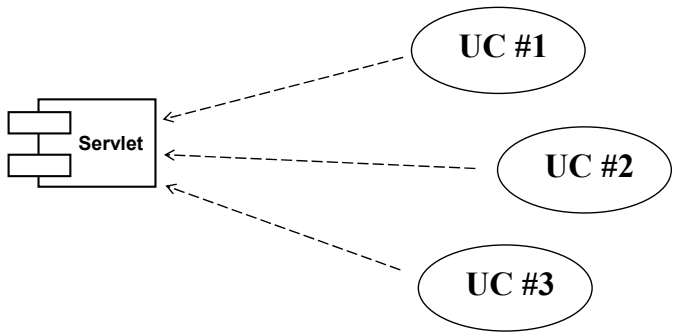


1-31

Single use case per Servlet is a sound design strategy. It allows for future growth that may be transparent, given the mechanism used to identify the request through the Servlet.

Multiple Use Cases Per Servlet

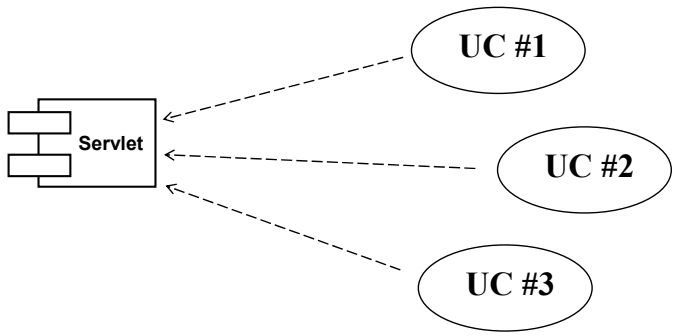
- **Several pathways from several use cases can be mapped to the same Servlet**
- **This reduces the number of objects that must be coded and managed**
- **This can lead to low cohesion and higher levels of regression testing should the application change in the future**



The designers must be careful of the “100 pound Servlet” syndrome. This is when the Servlet becomes bloated with operations and difficult to manage and maintain.

Using Action Objects and XML

- **It is feasible to have multiple use case per Servlet if the Servlet uses action objects implementing the Command pattern**
- **To add a new use case pathway requires only a new action class to deal with the functionality**
- **An XML stream can be used to define the action/class mapping allowing even further flexibility**



The Command Pattern (reviewed earlier) provides a level of abstraction making it much easier to add new functionality to the application in the future.

Action objects remove the need for this in your Servlets:

```
if ("Customer Inquiry".equals(action)) {
   doRltnCustomerInquiry(request, response);
   }
else if ("New Customer".equals(action)) {
   doRltnCustomerNew(request, response);
   }
else if ("Edit Customer".equals(action)) {
   doRltnCustomerEdit(request, response);
   }
else if ("Delete Customer".equals(action)) {
   doRltnCustomerDelete(request, response);
   }
else {
   response.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED);
    }
```

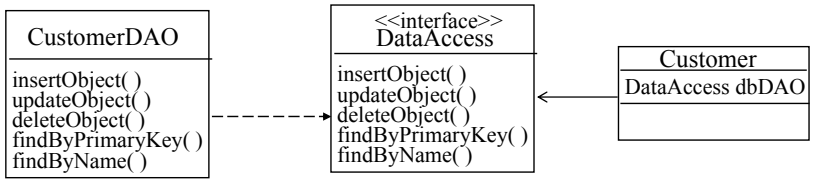
Value Objects

- **Server-side beans that may travel across the network to other servers in the same complex, can damage performance due to singleton get/set operations on attributes**
- **Create a proxy class that only contains attribute values and get/set operations**
- **These classes contain no business logic**
- **The main mission is to reduce network traffic by passing the object image in its entirety**

Value objects are quite popular in the Enterprise Java Bean world. They can vastly improve the performance of the container product (I.e., BEA Weblogic, IBM Websphere) by reducing the overhead of communicating with other objects.

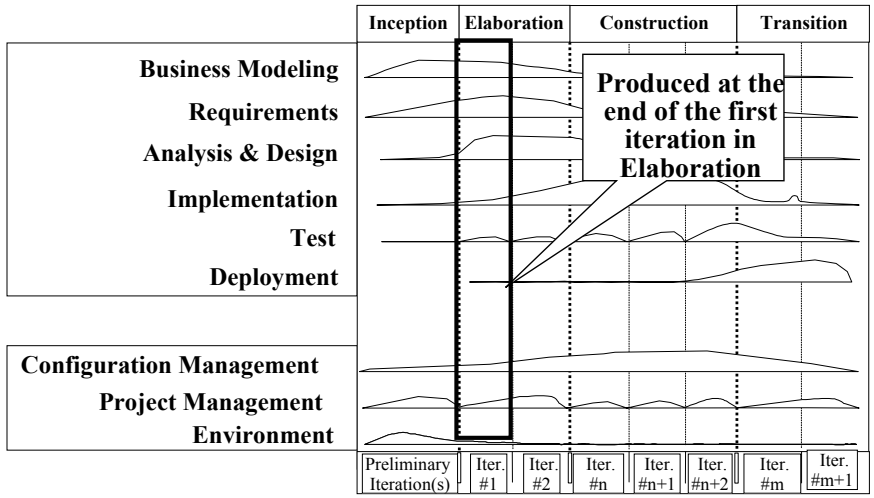
Data Access Objects

- **These objects encapsulate all the Structure Query Language (SQL) logic necessary for an entity class to carry out work on behalf of some interested client that requires data manipulation**
- **In the case of Enterprise Java Beans (EJB), these objects are only necessary if Bean Managed Persistence (BMP) is used. With Container Managed Persistence (CMP), there is no SQL to write**



Data Access Objects are the SQL workhorse in the design. Typically, there is one DAO class for each entity class. In these classes would be found all the different types of SQL Commands (i.e., Insert, Update, Delete, Select) for the underlying table.

The Architectural Prototype



No Hard Architecture Decisions Upon Completion

1-36

The Architectural Prototype is critical to a projects success. It comprises all use case pathways that are deemed “architecturally significant”. Upon the completion of this milestone, there should be no hard architecture decisions to make.

Architectural Prototype: EJB Impact

- **ALWAYS use a 100% CMP approach to flesh out the design model for the architectural prototype**
  - **Stabilizes relationships and associations**
  - **Verifies integrity**
  - **Identifies coupling and cohesion issues**
- **ALWAYS prototype a selection of set-oriented use case pathways using Session Beans**
- **ALWAYS prototype a selection of batch-oriented use case pathways**
- **ALWAYS try to avoid Stateful Session Beans (SFSB), opting for utilization of HTTPSESSION object**

Finding Architecturally Significant Requirements

- **Find Use Case pathways that touch these areas:**
  - **New technology usage**
  - **New organization processes**
  - **Temporal or timer based processing**
  - **Batch processing**
  - **Multi-panel interactions requiring state management across panels**
- **Don't select the "easy stuff" like reports and simple Create, Read, Update, Delete (CRUD) use case pathways**

1-38

Focus on Risk

Jackson-Reed, Inc.

"If you don't actively attack risks in your project, they will actively attack you"

Tom Gilb

Managing Change

Jackson-Reed, Inc.

"Change is not the Enemy,

Unmanaged Change is"