The Dynamic Mapping Design Pattern

J. Gresh (johngresh@johngresh.com) Pfizer Inc, Groton, Connecticut 06340
J. McKim (mckimj@winthrop.edu) Winthrop University, Rock Hill, South Carolina 29733
H. Sanchez (hsanchez@email.sjsu.edu) San Jose State University, San Jose, California 95192

Copyright 2005 by J. Gresh. Permission is granted to copy for the PLoP 2005 conference.
All other rights reserved.

Abstract
A design pattern for the elimination of lengthy conditional statements named Dynamic Mapping is presented. The design pattern presented will provide software developers a powerful tool to argue against the implementation of systems that do not scale well as a consequence of the need to maintain conditional mappings such as lengthy if-then-else statements, hash maps, XML representations of conditional logic, etc.

The design pattern presented uses a collection of objects that implement a common interface and dynamic class loading to replace lengthy domain specific conditional logic. Dynamic Mapping is demonstrated to saves costs incurred through the use of other implementations previously described including Factory Patterns, Model View Controller Servlets and other Controller mechanisms.

Key Words
Dynamic Mapping, dynamic class loading, Factory Pattern, Model View Controller.

Introduction
This paper describes the Dynamic Mapping design pattern. The Dynamic Mapping design pattern uses a language's underlying class loading mechanism to eliminate explicit domain specific mappings. The benefits of the use of the Dynamic Mapping design pattern are illustrated using two examples: use of the Dynamic Mapping design pattern in the context of a Factory implementation and use of the Dynamic Mapping design pattern in the context of a Model View Controller (MVC) Controller implementation.

Intent
Use a language's underlying class loading mechanism to eliminate explicit domain specific mappings.

Also Known As
Collection Switch

Motivation and Applicability
Frameworks and design methodologies that promote the use of lengthy conditional statements continue to appear regularly in the professional and academic [1-14,16-19,22-28] literature. These conditional statements can take many forms including traditional inline coding (e.g. if-then-else and other conditional statements) [1,2,5,14,16,26], collections of objects or object references [6,7,9,21], definition of conditional logic in external files (e.g. XML) [4,8,10,11,17,18,19,24,28], or the persistence of the conditional logic in external persistent stores (e.g. databases) [27]. Creating domain specific mappings such as these has three distinct negative consequences that must be considered. First, domain specific mappings will grow and need to be otherwise modified as the domain grows and evolves. Second, these

changes can introduce errors into the system. Third, domain specific mappings cannot be reused outside the domain of the system. The use of Dynamic Mapping creates mappings that do not grow or change as a consequence of growth or change in the domain. Dynamic Mapping reduces the potential for the introduction of errors into a system by the elimination of the requirement to change the mapping as the domain grows or changes. Dynamic Mapping demonstrates reuse that cannot be achieved when domain specific mappings are used.

Previous publications have suggested the use of collections of classes to replace lengthy conditionals [7,13]. However, these publications do not provide a formal demonstration that the proposed solution is an improvement over existing solutions. These publications do not suggest the use of dynamic class loading [20] to entirely eliminate the need of the software developer to maintain the mapping. Previous work does support the assertion that the maintenance of the collection used to replace the lengthy conditional represents a substantial amount of effort [7].

In [12], Fowler suggests the use of dynamic class loading to implement a portion of the conditional logic required for a controller class in a web application. The solution presented by Fowler does not include request forwarding as part of the functionality provided by the dynamically loaded class. Instead, the request forwarding is hard coded in an inline conditional statement. This hard coded inline conditional statement is domain specific. The hard coded inline conditional statement will require modification if the request forwarding of the system is expanded or modified. The need for this type of modification can be eliminated through implementation of the Dynamic Mapping pattern.

In [25] Rupp suggests that MVC is not an adequate solution for the implementation of Controller logic for scalable J2EE web applications. However, Rupp does not provided details regarding how an improved solution can be implemented and does not demonstrate that the methodology he proposes represents an improvement over current best practices.

The consistent use of some representation of a mapping suggests that this mapping has value. However, there are costs also associated with creating domain specific mappings. The benefits of using domain specific mappings must be weighed against the cost of using domain specific mappings.

One value of the use of mappings proposed is the ability to use mappings to create aliases [17]. By creating aliases a single mapping can be changed and thereby change the action taken by any client referencing that mapping. For example, a mapping can be created for a web application that has 100 Java Server Pages that all reference an action that does some specific server side processing and ultimately forwards the user to a JSP called cat.jsp for a given request with an attribute named "action" define as "cat". The mapping can then be changed to do some different server side processing and ultimately forward the user to a JSP called feline.jsp. Thereby changing a single value in a mapping can change 100 JSP references to perform a different action. While this is convenient in the short term it might not be the best solution in the long term. Creating two names for the same entity creates a condition where different terms mean the same thing from different perspectives. What is perceived to be a cat from one perspective is perceived to be a feline from another. Furthermore, all perspectives must consult the mapping to gain any meaningful information. This is exacerbated when multiple alias are allowed. If both feline and cat are added to the mapping and both point to cat.jsp an implication is made. This implication is that there are two different entities: a cat and a feline. It is implied that feline

is somehow different from cat. This is not the case. This type of mapping should be discouraged. Consistent naming should be encouraged.

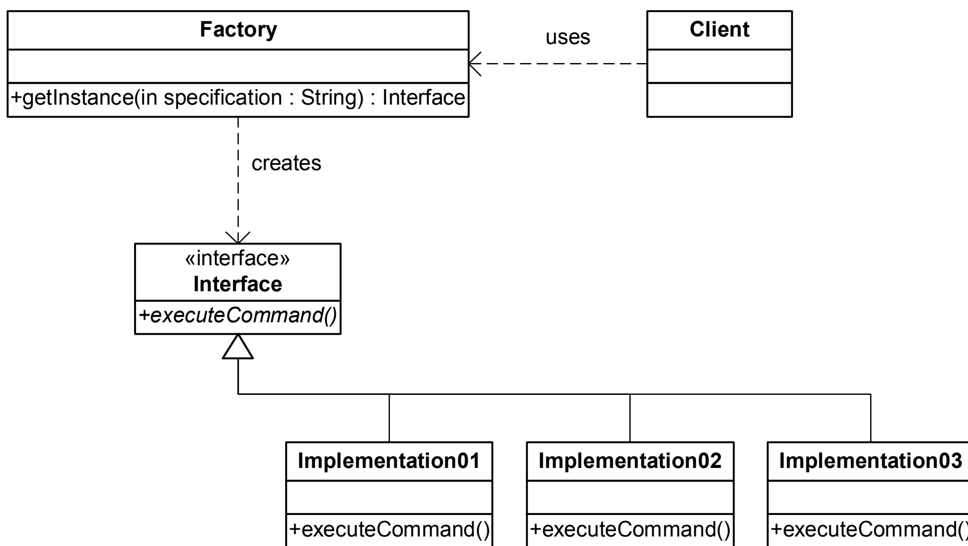
Dynamic Mapping defers mapping to the underlying class loading system of the language being used. This relieves developers of the task of creating, maintaining, and looking up mappings.

Applicability
Use the Dynamic Mapping pattern when

- An action needs to be taken based upon a specification. The action taken is determined by the specification.
- There is an expectation the number of actions is going to change.
- There is an expectation of an opportunity for reuse.

Pattern Structure and Participants

Structure
Figure 1: Dynamic Mapping Design Pattern's Structure.



Participants

- **Interface**
    - Defines the interface of the objects created by the Factory.
    - Defines methods that can be executed by either the Factory or the Client (or both, or delegates of either).
- **Factory**
    - Performs the action requested by the client.
    - Determines what action to take based upon the specification provided by the client.
    - Can execute one or more methods of the Interface.
    - May or may not return an instance of the Interface.
    - May or may not return an instance of some other class.
    - Does not use domain specific mappings.

- **Implementation**
    - Concrete definition of the Interface.
- **Client**
    - Makes requests to the Factory for a specific action.
    - Provides the specification regarding what action to take.

Collaborations

- Client depends upon Factory to provide an instance of the Interface.
- The factory creates an instance of the requested Implementation.
- Factory depends upon underlying language to provide the correct Implementation of the Interface based upon the given specification.

Consequences

Dynamic Mapping eliminates the need to create domain specific mappings. Dynamic mapping provides the following advantages over the use of domain specific mappings:

1. Reduction of errors: The potential for errors that can occur through the use of domain specific mappings is reduced.
2. Constant Size: Domain specific mappings will grow linearly with the number of mappings that are required. Implementation of Dynamic Mapping is concise and will not grow regardless of the number of mappings required.
3. Reuse: Domain specific mappings demonstrate little reuse. Use of the Dynamic Mapping design pattern produces classes that are highly reusable.

Dynamic Mapping uses a specification to determine the run time class provided by the Factory. This allows for potential run time errors by specifying a class that does not exist.

Implementation

A Dynamic Mapping Interface can be any valid interface and has the general form:

```
public interface Interface {
  public void executeCommand();
}
```

A Dynamic Mapping Factory class has the following general form:

```
public class Factory {
  public Interface getInstance(String spec) throws Exception {
    Interface interface = (Interface) Class.forName(spec).newInstance();
    interface.executeCommand();
    return interface;
  }
}
```

The getInstance method of the Factory implementation may have any valid return type and may or may not execute one or more methods of the Interface.

Sample Code
The value of Dynamic Mapping can be demonstrated by comparing a traditional implementation of a Factory pattern [15] with an implementation of the Factory pattern that uses Dynamic Mapping.

For example, consider a factory designed to generate instances of an Animal interface based upon a client request for an animal of a given type via a String. A traditional implementation can be defined as follows.

```
public class AnimalFactory {
  public Animal getAnimal(String name) throws Exception {
    if ("Ox".equals(name)) return new Ox();
    else if ("Rabbit".equals(name)) return new Rabbit();
    else if ("Ram".equals(name)) return new Ram();
    else if ("Rat".equals(name)) return new Rat();
    else if ("Rooster".equals(name)) return new Rooster();
    else if ("Tiger".equals(name)) return new Tiger();
    else if ("Dragon".equals(name)) return new Dragon();
    else if ("Snake".equals(name)) return new Snake();
    else if ("Horse".equals(name)) return new Horse();
    else if ("Monkey".equals(name)) return new Monkey();
    else if ("Dog".equals(name)) return new Dog();
    else if ("Pig".equals(name)) return new Pig();
    throw new Exception ("No animal defined for " + name);
  }
```

It is clear from this example that as each animal class is added to the system the factory class will need to be modified and will grow as a consequence of the addition of each new animal class. As the Factory class needs to be modified every time a new animal is added there is a potential that an error could be introduced to the factory class each time a new animal is added to the system.

Applying Dynamic Mapping the Factory class can be rewritten as shown below.

```
public class AnimalFactory {
  public static final String ANIMAL_PACKAGE_NAME = "animals";
  public Animal getAnimal(String name) throws Exception {
    String fullName = ANIMAL_PACKAGE_NAME  + "." + name;
    return (Animal) Class.forName(fullName).newInstance();
  }
```

Now the factory class does not need to be modified with the addition of new animal classes to the system and the factory class will not grow as a consequence of the addition of each new animal class. The potential for the introduction of errors to the Factory class with the addition of each new animal class is eliminated because addition of a new animal class to the system does not require any change to the factory class.

Additional reuse that can be gained using Dynamic Mapping can be demonstrated by comparing an implementation of a MVC Servlet based web application using the code described by Hall and Brown [16] and an implementation using a MVC Servlet based web application using Dynamic Mapping. Consider a variation of the Animal system described above recast as a web application required to process requests and forward a client to a JSP depending upon the value of a variable named

"operation". A Controller Servlet written using the method described by Hall and Brown could be written as follows.

```
public class Controller {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    String operation = request.getParameter("operation");
    if (operation == null) {
      operation = "unknown";
    }
    String address;
    if (operation.equals("Ox")) {
      address = "/WEB-INF/Ox.jsp";
    } else if (operation.equals("Rabbit")) {
      address = "/WEB-INF/Rabbit.jsp";
    } else if (operation.equals("Ram")) {
      address = "/WEB-INF/Ram.jsp";
    } else if (operation.equals("Rat")) {
      address = "/WEB-INF/Rat.jsp";
    } else if (operation.equals("Rooster")) {
      address = "/WEB-INF/Rooster.jsp";
    } else if (operation.equals("Tiger")) {
      address = "/WEB-INF/Tiger.jsp";
    } else if (operation.equals("Dragon")) {
      address = "/WEB-INF/Dragon.jsp";
    } else if (operation.equals("Snake")) {
      address = "/WEB-INF/Snake.jsp";
    } else if (operation.equals("Horse")) {
      address = "/WEB-INF/Horse.jsp";
    } else if (operation.equals("Monkey")) {
      address = "/WEB-INF/Monkey.jsp";
    } else if (operation.equals("Dog")) {
      address = "/WEB-INF/Dog.jsp";
    } else if (operation.equals("Pig")) {
      address = "/WEB-INF/Pig.jsp";
    } else {
      throw new ServletException(address + " unknown");
    }
    RequestDispatcher dispatcher =
      request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Applying Dynamic Mapping the Controller class can be rewritten as follows.

```
public class Controller extends HttpServlet {
  private String mActionName = "actionName";
  private String mPrefix = "";
  private String mPostFix = "";
  protected String getActionName() {
    return mActionName;
  }
  protected void setActionName(String actionName) {
    mActionName = actionName;
```

```
    }
    protected String getPrefix() {
      return mPrefix;
    }
    protected void setPrefix(String prefix) {
      mPrefix = prefix;
    }
    protected String getPostfix() {
      return mPostFix;
    }
    protected void setPostfix(String postfix) {
      mPostFix = postfix;
    }
    public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      processRequest(request, response);
    }
    public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      processRequest(request, response);
    }
    protected void processRequest(
        HttpServletRequest request,HttpServletResponse response)
        throws ServletException, IOException {
      ActionInterface action = null;
      String actionName = mPrefix + request.getParameter(mActionName) + mPostFix;
      try {
        action = (ActionInterface) Class.forName(actionName).newInstance();
        action.processRequest(request, response);
        RequestDispatcher disp =
          request.getRequestDispatcher(action.getNextUrl());
        disp.forward(request, response);
      } catch (Throwable t) {
        throw new ServletException(t);
      }
    }
  }
```

The ActionInterface used by the Dynamic Mapping Controller can be defined as follows.

```
  public interface ActionInterface {
    public void processRequest(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException;
    public String getNextUrl();
  }
```

Implementation of the Controller Servlet that does not use Dynamic Mapping shown above cannot be applied to other domains. The Controller Servlet that does use Dynamic Mapping could be used as a Controller for virtually any domain.

Alternative implementations of MVC Controller classes described in the literature will now be considered and compared to the Dynamic Mapping Controller implementation introduced here.

First, Jakarta Struts [3] must be considered. At the time of this writing Struts is arguably a contender for the gold standard for the implementation of J2EE web applications. It has been described as "the de facto framework for building J2EE web applications structure according to the MVC architectural pattern," [8] "A technology that no Web programmer can afford to ignore," [17], etc.
Struts (version 1.2) uses one or more XML configuration files to determine what action to take for a given request to the Controller [17]. Hypothetically, this approach does not solve the problem of eliminating the growth of the lengthy conditional as the size of the application grows. The XML file or files representing the lengthy conditional statement will grow as the number of actions used by the system grows. Therefore, in theory, Struts incurs the costs of increased potential for error, increase in size, and limited reuse that have been demonstrated here to be absent with a controller implemented using Dynamic Mapping.

Hypothetically, the potential for errors is exacerbated by the ability to create mappings that point to other mappings, i.e. to chain mappings together. When a series of mappings are chained, any incorrect mapping in the chain will cause an unexpected or undesired outcome. Any incorrect mapping in the chain will cause an error. Therefore, the longer the chain is the greater the potential for errors. The length of the mapping chain is reduced to 0 through the use of Dynamic Mapping. Thereby, any error in mapping is eliminated (with the obvious caveat that the original class name requested by the client must be valid).

Similar to Struts, Sun's "Pet Store" implementation uses an XML file to define controller action mappings [4]. Hypothetically, this implementation incurs the costs of increased potential for error, increase in size and limited reuse that have been demonstrated here to be absent with a controller implemented using the Dynamic Mapping design pattern.

Alure, Crupi, and Malks suggest encapsulation of mapping [6]. They suggest mappings can take two forms. One, the mapping can be direct whereby "the actual target object is held within the Map and can be directly retrieved." Two, the mapping can be indirect whereby "an indirect handle describes the scenario where the Map contains an object that refers to the ultimate target indirectly." These authors do not suggest the use of dynamic class loading to minimize the cost of this mapping. These authors do not demonstrate a controller implementation that is free from the costs of increased potential for error, increase in size and limited reuse. All these costs have been demonstrated here to be absent with a controller implemented using Dynamic Mapping.

Fowler has suggested the use of dynamic class loading if a JSP is used as a controller [12]. Fowler suggests the use of dynamic class loading to add an attribute to the page context of a JSP. The approach shown by Fowler does not define all navigation logic and thereby does not eliminate the need to modify the JSP with the addition of new navigation requirements. The code presented by Fowler is not as loosely coupled as it could be and is not as loosely coupled as the code presented here that uses Dynamic Mapping. The Servlet implementation that is provided by Fowler in [12] uses a mapping class to encapsulate mappings from the controller class. How this mapping class should be implemented is not provided.

Known Uses
Dynamic Mapping can be used in association with Factory patterns and Controller patterns.

Related Patterns
Related patterns include the factory patterns[15], Controller[22,23], J2EE web application Controller[24], and the Product Trader patterns[7].

Conclusions
Use of the Dynamic Mapping design pattern eliminates any representation of a lengthy conditional created and maintained by the authors of a system. The Dynamic Mapping design pattern defers this responsibility to the underlying language by using dynamic class loading and thereby reduces the number of potential errors that can be generated, reduces the amount that the system will grow, and adds additional opportunities for reuse.

References

[1] "The Java Boutique: Java Tutorial: A very simple JSP-architecture", Available at HTTP: http://javaboutique.internet.com/tutorials/Simple_JSP/controller.html.

[2] "JSP Templates: Model-View-Controller for Servlets", Available at HTTP: http://www.caucho.com/articles/jsp_templates.xtp.

[3] "The Apache Foundation: Struts", Available at HTTP: http://struts.apache.org

[4] "Java Pet Store", Available at HTTP: http://java.sun.com/developer/releases/petstore

[5] "Java Education and Learning Community: Abstract Factory", Available at HTTP: http://ww.javapractices.com/Topic128.cjp

[6] D. Alur, J. Crupi, and D. Malks, Core J2EE Patterns, Best practices and Design Strategies, 2nd. ed., Palo Alto, California, Sun Microsystems, 2003.

[7] D. Baumer and D. Riehle, "Product Trader," in Pattern Languages of Program Design 3, R.C. Martin, D. Riehle, and F. Buschmann, Eds. Reading MA: Addison-Wesley, 1997. Chapter 3.

[8] F. Bellas, D. Fernandez and A. Muino, "A flexible framework for engineering 'my' portals," Proceedings of the 13th international conference on World Wide Web, pp. 234-243, 2004.

[9] S. Burbeck, "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)", Available at HTTP: http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, 1987, revised 1992.

[10] L. G. Chun, W. Yanhua, and L. X. Hanhong, "A Novel Web Application Frame Developed by MVC", ACM SIGSOFT, 2003.

[11] J. Corwin, D.F. Bacon, D. Grove and C. Murthy, "MJ: A Rational Module System for Java and its Applications," ACM SIGPLAN Notices, vol. 38, no. 11, Oct., 2003.

[12] M. Fowler, Patterns of Enterprise Application Architecture, Boston, Addison-Wesley, 2003.

[13] M. Fowler, Refactoring: improving the design of existing code, Reading MA: Addison-Wesley, 1999.

[14] M. Grand, Patterns in Java, Volume 1, New York, John Wiley & Sons, 1998.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: elements of reusable object-oriented software, Boston: Addison-Wesley, 1994.

[16] M. Hall and L. Brown, Core Servlets and JavaServer Pages Volume 1: Core Technologies, Upper Saddle River, NJ: Prentice Hall, 2004.

[17] J. Holmes, Struts: the complete reference, H. Schildt, Ed., New York: McGraw-Hill, 2004.

[18] N. Kassem, Designing Enterprise Applications with the J2EE Platform, Second Edition, Addison-Wesley, Boston, 2000.

[19] S. Kojarski and D. H. Lorenz, "Domain Driven Web Development With WebJinn", OOPSLA, 2003.

[20] S. Liang and G. Bracha, "Dynamic Class Loading in the Java Virtual Machine," ACM SIGPLAN Notices, vol. 33, no. 10, 1998.

[21] D. Nguyen and S. B. Wong, "Design Patterns for Games", SIGCSE, 2002.

[22] T. Reenskaug, "Models-Views-Controllers", Available at HTTP: http://heim.ifi.uio.no/~trygver/themes/mcv/mvc-index.html, 1979.

[23] T. Reenskaug, "The Model-View-Controller (MVC) Its Past and Present", Available at HTTP: http://heim.ifi.uio.no/~trygver/themes/mcv/mvc-index.html, 2003.

[24] B. Searns, G. Murray, I. Singh, et. al., Enterprise Applications with the J2EE Platform, Second Edition, Available at HTTP: http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html

[25] N. A. Rupp, "Beyond MVC: A New Look at the Servlet Infrastructure", Available at HTTP: http://today.java.net/lpt/a/62, 2003.

[26] G. Seshadri, "Understanding JavaServer Pages Model 2 architecture", Available at HTTP: http://www.javaworld.com/javaworld/jw-12-1999/sw-12-ssj-jspmvc_p.html, 1999.

[27] J.W. Yoder, F. Balaguer, and R. Johnson, "Architecture and Design of Adaptive Object-Models," ACM SIGPLAN Notices, vol. 36, no. 12, Dec., pp. 50-60, 2001.

[28] J. Zhang, J. Chung, and C.K. Chang, "Towards Increasing Web Application Productivity," in Proceedeings of the 2004 ACM symposium on Applied computing, pp. 1677-1681, 2004.