8-8-2007

Applying Grid-Partitioning To The Architecture of the Disaster Response Mitigation (DISarm) System

Aline Vogt
University of New Orleans

Applying Grid-Partitioning To The Architecture of the Disaster Response Mitigation (DISarm) System

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
The Department of Computer Science

By

Aline Vogt

B.S. University of New Orleans, 2003

August, 2007

Acknowledgements

I would like to thank Dr. Shengru Tu, my advisor, for providing me with the guidance needed to see this research project through to its completion, and for the support and prodding that kept me working when I would have given up.

I would also like to thank Dr. Mahdi Abdelguerfi and Dr. Vassil Roussev for being a part of my thesis committee.

Table of Contents

List of Figures

List of Tables

Abstract

The need for a robust system architecture to support software development is well known. In enterprise software development, this must be realized in a multi-tier environment for deployment to a software framework. Many popular integrated development environment (IDE) tools for component-based frameworks push multi-tier partitioning by assisting developers with convenient code generation tools and software deployment tools which package the code. However, if components are not packaged wisely, modifying and adding components becomes difficult and expensive. To help manage change, vertical partitioning can be applied to compartmentalize components according to function and role, resulting in a grid partitioning. This thesis is to advocate a design methodology that enforces vertical partitioning on top of the horizontal multi-tier partitioning, and to provide guidelines that document the grid partitioning realization in enterprise software development processes as applied in the J2EE framework.

System architecture
Software modeling
Enterprise software development
J2EE system design
Grid-partitioning software models

Chapter 1:Introduction

The model-view-controller (MVC) pattern is one of the mainstream software design principles in software design. Applied to the system-level design of Web-based information systems, the MVC pattern suggests separating the components along a number of divisions in which the *model*-layer consists of the databases and the objects in the persistent layer; the *view*-layer consists of Web page documents, Web browsers, and Web servers; the *controller*-layer is represented by the logic in application servers and the programs (such as servlets and Web services) that connect the application servers and the Web presentation documents. Modern software frameworks such as the J2EE server and the .NET framework server realize the MVC pattern in multiple tiers, and enforce partitioning in their architectures into high-level components such as Web servers, application servers, databases, and Web clients.

Many popular integrated development environment (IDE) tools for component-based frameworks such as IBM's WebSphere and BEA's WebLogic push multi-tier partitioning further by assisting developers with convenient code generation tools and software deployment tools which package the code along the MVC divisions. For example, nearly every enterprise-level IDE can automatically generate all the required entity beans that relate to the data store, and pack them into easy-to-deploy components. This great convenience often leads to unexpected inflexibility. If the entity beans are not wisely packaged according to proper partitions, any modification of an entity bean would require either regeneration of the entire entity bean package, or tedious updates to beans and related files, such as deployment descriptors.

Almost every tutorial, textbook and technical article on development of J2EE Web systems that I have reviewed advocates and teaches the multiple-tier architecture in an overly simplified manner. Nearly every example in those training documents illustrates a facile

partitioning in deployment of the programs. Both the IDE tools and the tutorials encourage their users to pack the components in each tier into a deployment unit – package. However, I have realized in my practice that as a system becomes more complex, such a seemingly straightforward partitioning results in a structure that is inflexible, and does not localize the impact of software changes, especially the changes triggered when business functions are added.

This thesis is to advocate a design methodology that enforces vertical partitioning on top of the horizontal multi-tier partitioning. Specifically, my methodology for development uses the J2EE platform for enterprise software development. By vertical partitioning, I mean that elements of the system will be compartmentalized according to function and role. The implementation and deployment views will be partitioned as well. Thus, the servlets, session beans, enterprise Java beans (EJBs) and the database structure will be compartmentalized according to their function and role in a vertical partitioning. Using both MVC-guided division and vertical partitioning will result in a design that forms a grid-like partitioning. Grid partitioning is not a new idea; Moore and his group at IBM advocated its use and presented an example of the design outcomes of such practice for a simple web application in [5].

There are many different development processes that have been advocated. However, as Moore states in [9], "One of the reasons there is such a great variety in software development processes is the fact that each project is different from every other project." The reason that I emphasize this approach in my thesis is that I have found no systematic methodology of realization of grid partitioning as applied to an enterprise level project in any literature. It has been largely ignored in classroom software education. The purpose of this thesis is to provide guidelines that document the grid partitioning realization in enterprise software development processes.

I will illustrate an example system that is adequately complex to justify the needs of the grid-partitioning approach, and at the same time is small enough to be presented in this thesis. This is the Disaster Response Mitigation (DISarm) System, a Web-based system that is to gather information concerning the type and scope of natural and man-made disasters, and the needs of citizens and governments during and after disasters. These metrics may then be used to gauge the timeliness, adequacy and effectiveness of governmental response. The first-phase module of the DISarm system was a subsystem that gathers information concerning the handling of garbage and debris during and after disasters. Additional modules were to provide decision-making assistance for use by emergency and disaster response teams, and to allow citizens to register their locations while they are displaced during a disaster.

Particularly, I have documented the complete development process of the DISarm system. This is important because tracing the impact to the design due to changes in requirements provides us with clues as to how to systematically realize the grid-partitioning approach. The DISarm system had to be powerful enough to capture the complex data needed for study and analysis of disaster response, but at the same time had to present an interface to the user that was easy to understand and navigate. J2EE was the choice of platform because this is a proven robust, scalable and secure technology. However, using J2EE added additional complexity and design issues to the development of the system. Careful planning and meticulous design has been necessary. I have documented the lessons learned from the essential system development process starting from collecting use cases, to use case realization, component design, implementation supported by code generation, and deployment.

In order to validate the benefits of the grid-partitioning approach, such as flexibility and extensibility, I have carried out a comparison study. In doing so, the system was designed in two

versions. The first version was the design of the first-phase module and used simple tier-by-tier divisions. Then additional requirements relating to one of the additional modules (contact information register) and updates to existing requirements were incorporated in the second version. The effect of the changes was gauged. The second version of the design was carried out according to the grid-partitioning approach. A comparison between the designs resulting from the simple tier-by-tier partitioning approach and the grid-partitioning approach illustrated that the grid-partitioning approach has the capability to minimize the impact caused by components modification, addition or removal by localizing the effect of changes triggered by new or updated system requirements.

The remaining parts of this thesis are organized as follows: Chapter 2 sets out the background of the Unified Process and J2EE. Chapter 3 details the initial system requirements, the actor catalog, and use cases. Chapter 4 contains the initial design. Chapter 5 sets out additional requirements for the system. Chapter 6 discusses the impact of the additional requirements on the original model, and the weaknesses of the original design. Chapter 7 updates the model, applying vertical partitioning and a clear delineation between components. Finally, Chapter 8 concludes.

Chapter 2:Background

The Unified Process Methodology

Software development is traditionally divided into phases: requirements, design, implementation, testing and release. This study focuses on the requirements and design phases of software development, since the concentration herein is on system architecture.

Several methodologies were considered for design of the DISarm system. The traditional waterfall methodology progresses in a linear fashion from requirements gathering to the final release of the product. This method was not used because the DISarm system will evolve over time as new requirements are gathered and implemented, and thus an iterative approach was felt to be more appropriate.

There are a number of iterative approaches that could have been used. From these, the Unified Process (UP) was chosen because of its clearly defined phases (Inception, Elaboration, Construction, Transition) and workflow iterations within phases. The five core workflows are: requirements, analysis, design, implementation, and test [1]. In the requirements workflow, user requirements are gathered to capture system scope and functionality; in analysis, requirements are refined and restructured; and in design, the system architecture is created [2]. The model elements used to build the system architecture of the DISarm system are based on the "4+1" View espoused by Philippe Kruchten [3]. Four views, the logical view, the physical view, the process view and the development view, are organized around the fifth view, which is the use case view [3].

The logical view describes functionality provided to the users. In the DISarm architecture, this view is realized in a class diagram, as well as the entity-relationship diagram (ERD). The process view is a variation on the logical view that includes non-functional

elements, such as performance and concurrency. The development view describes the system's organization, and is depicted in a package diagram. The physical view shows how the software is deployed onto the hardware. It is represented in a deployment diagram. The fifth view, which is the use case view, both unifies the other views, and provides a foundation from which the other views may be developed.

The model diagrams for the DISarm system were produced using the Unified Modeling Language (UML), which is an Object Management Group (OMG) standard for modeling software artifacts [6]. There is a great deal of flexibility in how UML is used, and models may be incomplete, or even inconsistent. Model elements may be hidden in some diagrams and shown in others, depending on the purpose for which the diagram is constructed. However, model semantics must be included for the model to have meaning [2].

Model-View-Controller Pattern and Multi-Tier System Perspective

The Model-View-Controller Pattern is illustrated in Figure 2-1. The model contains the data components and business rules, which include accessing data and updating data in the data store. The view handles the presentation of data to the user, and takes inputs from the user. The controller acts as an intermediary between the model and view. It processes inputs from the view and turns them into actions to be performed by the model [8].

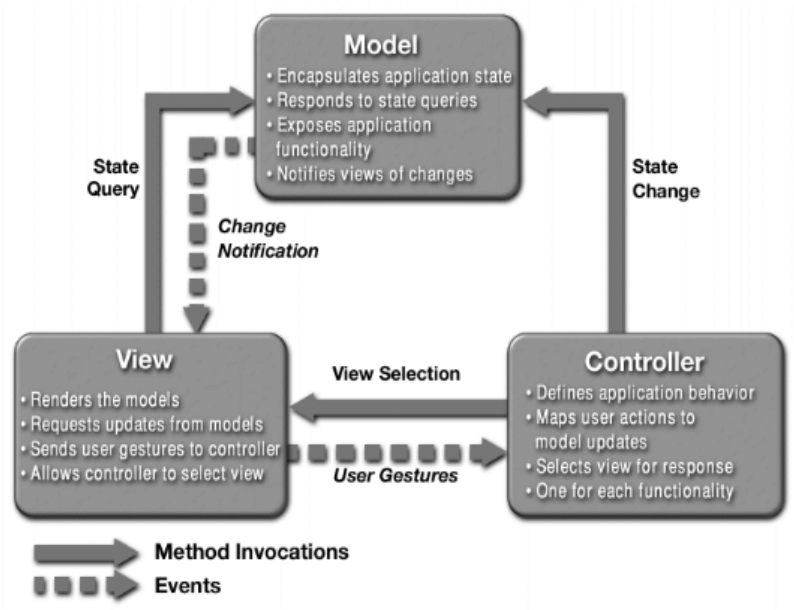

Figure 2-1: MVC [8]

When applying the MVC pattern to the Web information systems at the system level, the interactions ("state query", "change notification") between the Model and the View are cut off. Rather, these interactions are allowed between the Model and the Controller. Thus, the widely accepted multi-tier system architecture is defined as shown in the left-most column in Table 2-1. The correspondence between the perspectives of the multi-tier architecture and the MVC is listed in the right-most column of Table 2-1.

Table 2-1: The multi-tier architecture and the MVC perspective

Multi-tier system perspective	*Example components*	*MVC perspective*
Web server: presentation and Web control	JSP, HTML	View
	Servlet, Web service	Control
Application server: business logic	Session bean	
	Entity bean	Model
Database: data layer	Database	

Grid-Partition Approach

Moore, et al suggests using the divide and conquer approach of partitioning an application into components for development [5], and state that when this approach is used, there is a clear division of functionality into vertical partitions.

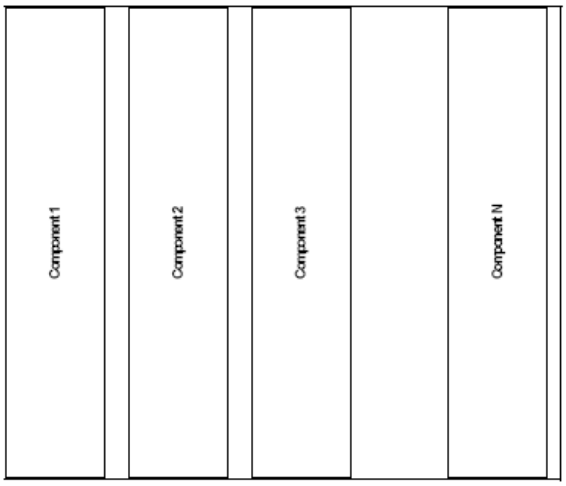

Figure 2-2: Vertical partitioning [5]

This is a fundamental approach incorporated in UP – which defines a component as a "physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces."

We take this approach a step further to document guidelines that will produce components that may be split either into vertical partitions or the divisions of MVC, as shown in Figure 2-3.

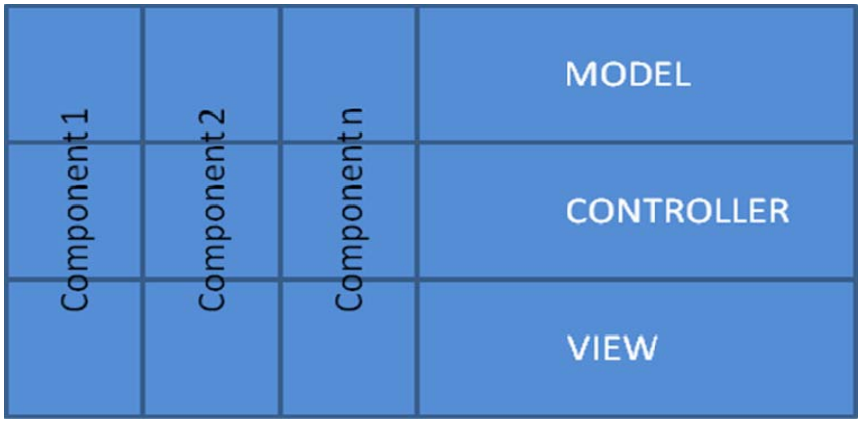

Figure 2-3: Grid Partitioning

The J2EE Platform

J2EE is a Java platform developed by Sun Microsystems to support the development and deployment of multitier, Web-based applications. As of the version 1.5 release of J2EE, its name was changed to Java EE; however, we continue to refer to it as "J2EE" as that is the name in common use.

In J2EE applications, components are layered by functionality, and can be installed on different servers according to their purposes. J2EE provides a "component-based approach to the design, development, assembly and deployment of enterprise applications." [4] The J2EE container structure is shown in Figure 2-4.

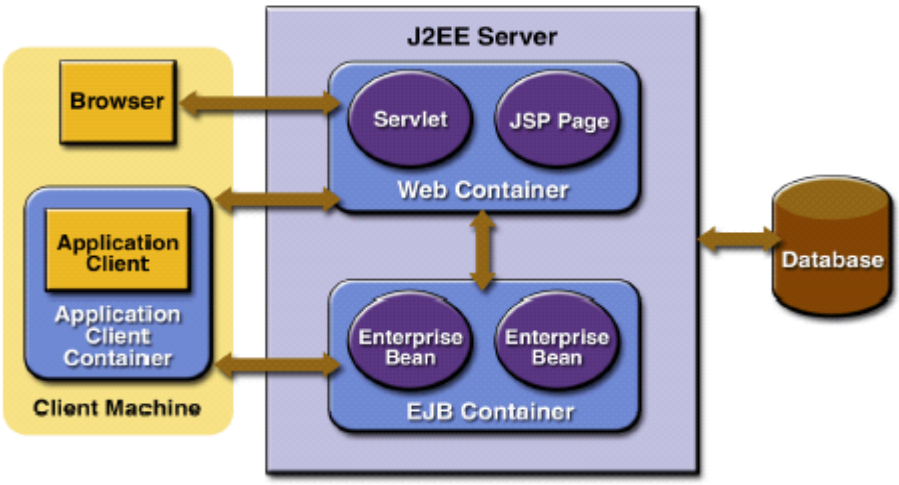

Figure 2-4: J2EE Server and Containers [4]

The client layer can consist of either a thin, browser-based client, or a thick application client. The web layer contains servlets and JSP pages. The JSPs form the View component of the MVC pattern. The Controller consists of servlets, which process user requests and handle system navigation. The Model component of the MVC pattern represents the business logic of the application and is responsible for maintaining application data. It is implemented using Enterprise JavaBeans (EJBs). The EJBs can be session beans (stateful or stateless), entity beans, or message driven beans.

Since J2EE supports distributed transactional applications in a robust and secure environment, it is a sound choice for implementation of the DISarm system. And although the use of MVC is natural with J2EE, the separation of application components into vertical partitions is not. As stated above, our goal is to show how the UP and UML can be used to support the design and development of J2EE applications that are partitioned in a grid-like pattern.

The Integrated Development Environment

In addition to the fundamental editing capabilities, integrated development environments offer many services to developers such as debuggers and built-in compilers. When working with J2EE applications, one of the more useful enhancements provided by some IDEs is automatic bean generation. One such IDE is the Rational Application Developer (RAD).

RAD can be used to generate the basic code for servlets and EJBs. It provides a mechanism for a bottom-up generation of the entity beans from tables in the database. The process constructs the entity bean and the create, get and set methods needed to handle database transactions, as well as the required interfaces to the entity bean. This can have its pitfalls, however.

Updates to the database require updates to the corresponding entity beans. If some of the entity beans have been modified by developers, automatic regeneration will overwrite the developers' code. We found through experience that it is easier and more reliable to regenerate all of the entity beans contained in one package, rather than trying to regenerate only some of the beans. This is because of the dependencies to the deployment descriptors and other modules that the IDE will automatically update on bean regeneration. So, if beans are not packaged efficiently, even a small change to the database can require regeneration of a large number of entity beans. In order to avoid these problems, careful consideration must be given in the design phase to how the beans will be packaged and deployed.

Chapter 3:Initial Requirements and Use Cases

We begin the development process for the DISarm system with the first step of the UP, requirements gathering. As stated in the introduction, the DISarm architecture was built in two iterations. In the first iteration, requirements were gathered and use cases were written for the first-phase module, the module concerned with gathering information on garbage and debris collection. The requirements gathering stage for the first-phase module is the subject of this chapter. Based on use case analysis, design elements of the model were produced following the MVC pattern, as set out in Chapter 4. Once the initial design was complete, we began the second iteration by updating requirements and adding new requirements. This is the subject of Chapter 5. Chapter 6 discusses the effects of the changes on the original design, and implementation and deployment issues introduced by the use of J2EE as the application framework. The architectural process is completed in Chapter 7, which shows the model as revised using the grid-partition methodology.

The first module of DISarm is the module that accumulates data on garbage and debris handling. Because garbage collection crosses both normal governmental services and disaster response, it was chosen as the base module of the system.

The above statements set the scope of the initial system, *e.g.* the customer wants a system that will provide for the input and storage of data relative to garbage and debris collection. However, these statements do not define what the system will do and how it will do it. This will be the focus of our requirements gathering. In order to determine what will be built, the customer was interviewed and initial requirements were gathered. These requirements are shown in Table 3-1:

Table 3-1: Initial Requirements

Number	Requirement Statement
1	The DISarm system shall allow a citizen to create his/her user profile.
2	The DISarm system shall allow citizens to report incident(s) concerning garbage/debris collection (on their own behalf or on other property).
3	The DISarm system shall allow citizens to enter the severity/urgency of an incident reported.
4	The DISarm system shall allow citizens to update reports(s) they have made.
5	The DISarm system shall allow citizens to view reports and/or maps showing incidents reported.
6	The DISarm system shall allow citizens to view report(s) showing information concerning responses by contractors to incidents.
7	The DISarm system shall allow contractors to review reports.
8	The DISarm system shall allow contractors to enter the status of their responses to incidents (active/closed).
9	The DISarm system shall allow contractors to view report(s) showing information on incidents entered and responses to incidents.
10	The DISarm system shall allow contractors to update the status field of utilities and other facilities.
11	The DISarm system shall allow government officials to verify reports entered by citizens.
12	The DISarm system shall allow government officials to create incidents.
13	The DISarm system shall allow government officials to associate citizen reports to incidents.
14	The DISarm system shall allow government officials to enter information concerning contractor companies, including company location, type of business, company contact.
15	The DISarm system shall allow government officials to enter information concerning contractor performance.
16	The DISarm system shall allow government officials to enter information concerning contracts.
17	The DISarm system shall allow government officials to enter information concerning facilities.
18	The DISarm system shall allow government officials to enter information concerning resources available for disaster response (including type of resource, location of resource, available transport).
19	The DISarm system shall allow government officials to view reports on incidents and responses.
20	The DISarm system shall allow an Emergency Manager to allocate resources to respond to an incident.
21	The DISarm system shall allow an Emergency Manager to enter and update information concerning contractors, contracts and contractor facilities.
22	The DISarm system shall allow an Emergency Manager to view all available reports.

It is apparent that the above requirements are not sufficient to allow coding of the DISarm system to begin. At best, these requirements are incomplete; at worst they may be vague

or misleading – and this is only a short list of requirements for a system that initially will be relatively small. As stated by Jacobson, Booch and Rumbauch in [1], it is "absurd to believe that the human mind can come up with a consistent and relevant list of requirements in the form 'The system shall . . . .'" A more intuitive way to capture requirements is through use cases. Use cases are written from the point of view of the users of the system, and thus facilitate communication with the customer as to what the system will do. But this is not the only function use cases serve. They also form a foundation from which the rest of development work can flow [1].

Discovering the use cases pertinent to a system can be challenging. The best way is to determine the actors (users) who participate in the system, and then examine how each actor will use the system [2]. To find the basic use cases for the DISarm system, the requirements listed in Table 3-1 were reviewed. The following actors were found: Citizens, Contractors, Government Officials and Emergency Managers. The requirements were then reviewed for information on how each of these actors will participate in the system. Finally, the use cases were inspected to find any missing actors and use cases. The only omission made is that an Application Manager will be needed to create user accounts for privileged users, such as the government officials and emergency managers. The actors and their roles are summarized in the DISarm actor catalog shown in Table 3-2:

Table 3-2: Actor Catalog

Actor	Description
Application Manager	Human actor responsible for • creating user accounts for government officials and emergency managers
Citizen	Human actor responsible for • creating his/her user profile • making reports(s) • update his/her own reports • viewing reports
Contractor	Human actor responsible for • updating incidents (*i.e.*, active/closed) • updating status field of utilities • viewing reports
Government Official	Human actor responsible for • verifying citizen reports • creating incidents • entering contractor information • entering contract information • entering facility information • updating contractor performance information • entering information on emergency resources • viewing reports
Emergency Manager	Human actor who: • has role of government official • managing resources • allocating resources

The requirements were then reviewed to determine how each of the above actors would participate in the DISarm system. Based on this review, an initial use case model was developed, as shown in Figure 3-1. The use case model consists of the graphical representations of the use case, and the textual specifications that form the backplane of the model [1].

Figure 3-1: Initial Use Case Model

The use case specifications for the initial DISarm system are set out in Appendix 1. Primary actors are the users who initiate a use case; participating actors join in or are the beneficiary of the actions of a primary actor. The steps set out for each use case are a first-level break out of how the actors will interact with the DISarm System, and for the most part only detail the primary flows of the use case.

Chapter 4: Initial System Design

Model Elements

Once the initial requirements of a system have been documented in use cases, the use cases can be studied to gather the information necessary to create analysis and design models. The goal is to produce "consistent models that are sufficiently complete to allow construction of a software system." [2] For the initial iteration of the DISarm design, it was determined that in addition to the use case model, a class diagram, ERD, and one or two activity diagrams would form an adequate blueprint for the system.

As stated above, the UP and UML provide a great deal of flexibility as to the level of detail that must be included in a model. For a class model, the only required element is the name compartment with the class name [2]. We have chosen to include key attributes and key operations in addition to the class name. Entity classes may be found by studying use cases, the information involved, and how the information will be manipulated [1]. This methodology was used for DISarm and the initial class diagram is shown in Figure 4-1.

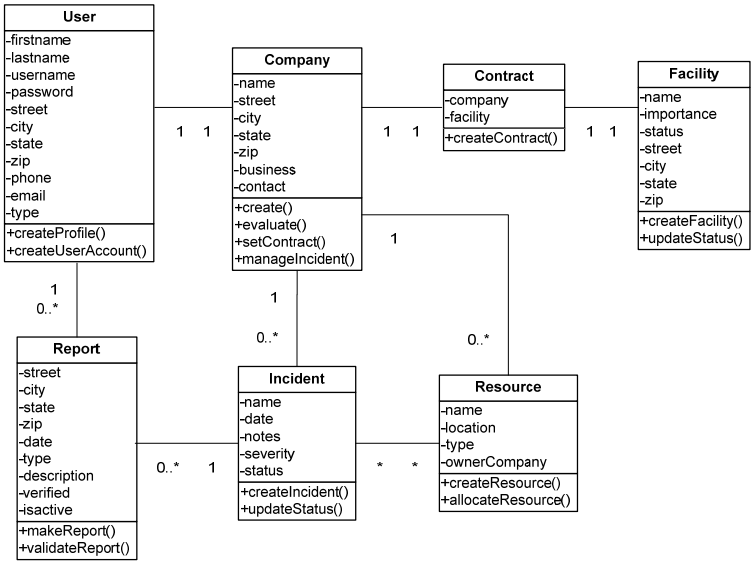

Figure 4-1: Initial Class Diagram

Most of the workflows of the DISarm system were straightforward and easily understood, so it was not felt that a large number of activity diagrams were needed to enhance understanding of the system. The activity diagram for the use cases Update Incident Status and Create Contract are included, as these use cases will be updated in the modified system described in future chapters.

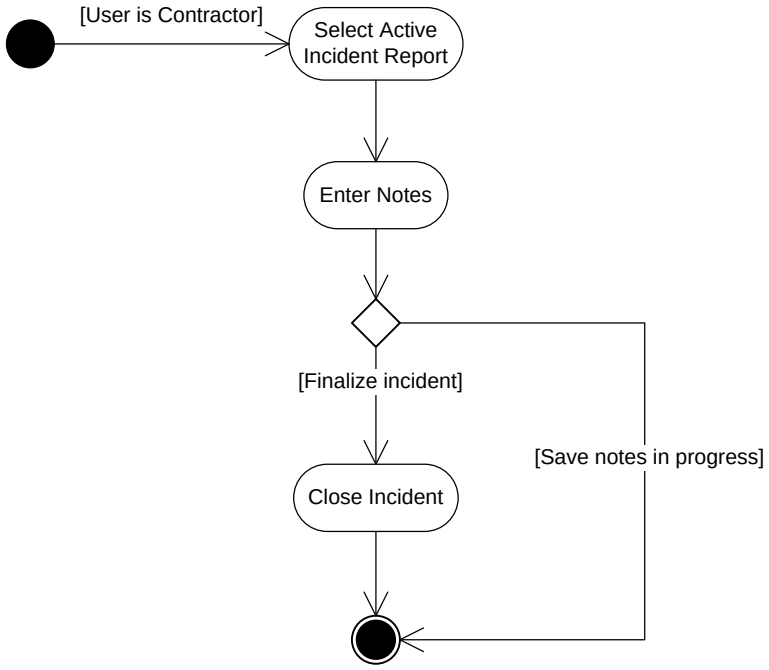

Figure 4-2: Update Incident Status Activity Diagram

As may be seen from Update Incident Status activity diagram in Figure 4-2, this activity is straightforward and simple – a contractor user is allowed to set the status of an incident to closed (inactive).

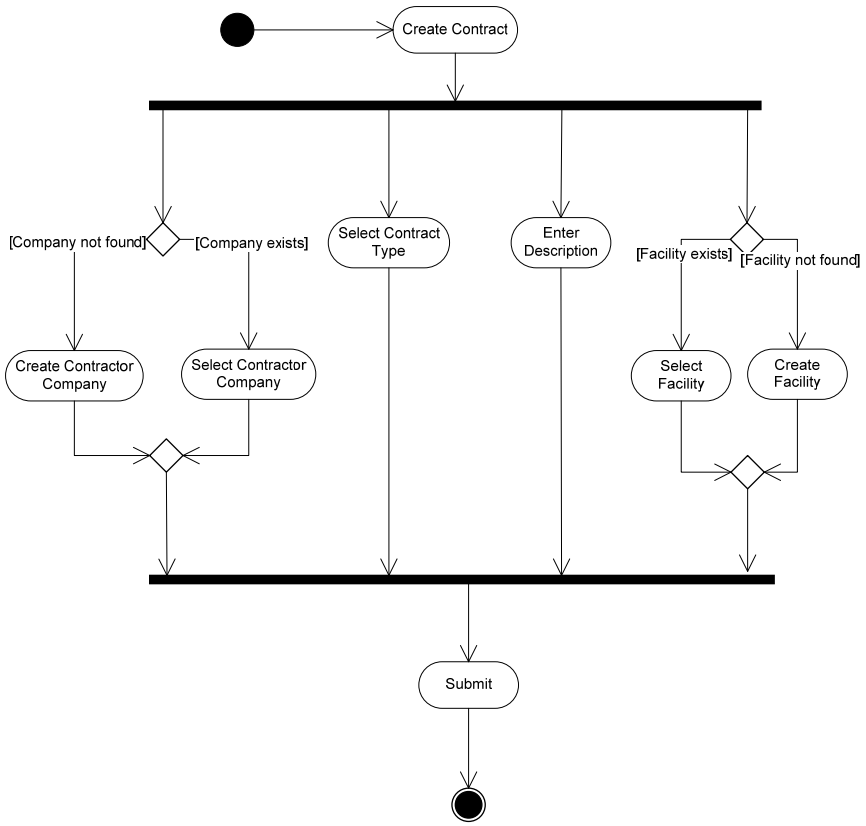

Figure 4-3: Create Contract Activity Diagram

The Create Contract activity diagram is somewhat more interesting. It shows that the activities of Create Contract take place in parallel, which informs the web designer that the web page for Create Contract should display these options on the same web page, and it also has branching flows, where Create Contractor Company and Create Facility are optional flows that execute if the company or facility to be associated to the contract do not exist.

The final diagram of the initial system model is the ERD. This was developed from the class diagram and use cases to contain the tables and the table attributes that will be required for persistent data storage. This model is depicted in Figure 4-4.

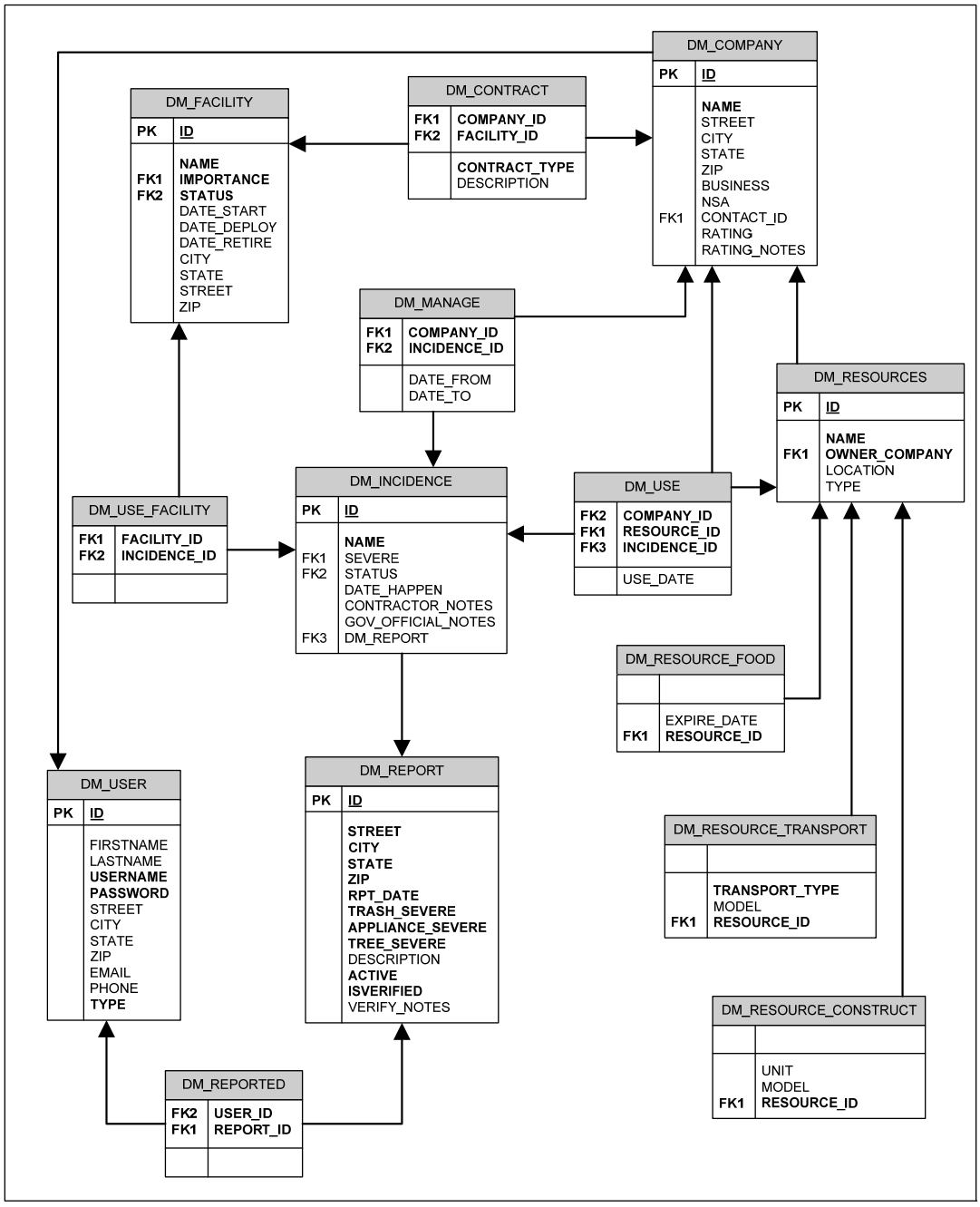

Figure 4-4: Data Model for Initial DISarm System

Implementation

The ERD completed our initial design, and development moved into the implementation phase. Based on the design, the J2EE components identified as being needed were JSPs, servlets, session beans and entity beans.

The JSPs needed were determined by reviewing the use cases, and are listed in Table 4-1.

Table 4-1: JSPs

JSPs		
Login.jsp	CreateUserProfile.jsp	CreateIncident.jsp
CitizenIncidentReport.jsp	UpdateUserProfile.jsp	IncidentStatusUpdate.jsp
CitizenReportUpdate.jsp	VerifyCitizenReport.jsp	CreateContractorCompany.jsp
CreateResource.jsp	AllocateResource.jsp	CreateUserAccount.jsp
CreateContract.jsp	CreateFacility.jsp	ViewCitizenReport.jsp
EvaluateCompany.jsp	UpdateFacilityStatus.jsp	ViewContractors.jsp
ViewIncident.jsp	ViewResources.jsp	ViewFacilities.jsp
ViewContracts.jsp	ViewUserAccounts.jsp	

The JSPs were placed in a WebContent folder, to be deployed to the web container of the J2EE Server.

The servlets process user requests and construct responses, and control navigation through the application. The servlets for the initial DISarm system, and a brief description of the user requests handled by the servlets, are shown in Table 4-2. The servlets were packaged in a JavaResource folder for deployment to the web container of the J2EE Server.

Table 4-2: Servlets

Servlet	Function	Related JSPs
IncidentServlet	Handles requests to make reports, create incidents, and updates to incidents and reports.	CitizenIncidentReport.jsp, CitizenReportUpdate.jsp, VerifyCitizenReport.jsp, CreateIncident.jsp, IncidentStatusUpdate.jsp
UserServlet	Handles user accounts	Login.jsp, CreateUserProfile.jsp, UpdateUserProfile.jsp, CreateUserAccount.jsp
ReportServlet	Handles all requests to view reports	ViewIncident.jsp, ViewContracts.jsp, ViewCitizenReport.jsp, ViewResources.jsp, ViewUserAccounts.jsp, ViewContractors.jsp, ViewFacilities.jsp
ResourceServlet	Handles requests to create companies, facilities, resources, and contracts and all requests for updates to resources	CreateContractorCompany.jsp, CreateResource.jsp, CreateFacility.jsp, UpdateFacilityStatus.jsp, CreateContract.jsp, EvaluateCompany.jsp, AllocateResource.jsp

Session beans are needed to hold the methods that implement the business logic of the system. Stateless session beans do not hold the client's conversational state. Stateless session

beans may be accessed by multiple clients, and thus offer better performance and scalability [4]. The stateless session beans for the DISarm system correspond to the servlets. At least one stateful session bean is needed to support a client's transient interaction with the system. In the DISarm system, this is the SessionManagerBean. Beans are accessed through interfaces. The home interface defines bean life cycle methods; business methods are defined in either the local home interface (for local access) or remote interface (for remote access) [7].

The automatic generation capabilities of the IDE were used to create the session beans and their interfaces; the beans were then modified as necessary to add methods to support the business logic of the system. The session beans and their interfaces were placed in an EJBSession package for deployment to the EJB container of the J2EE server. The DISarm session beans are listed in Table 4-3.

Table 4-3: Session Beans

Bean	Local Interfaces	Remote Interface
SessionManagerBean	SessionManagerHome	SessionManager
IncidentManagerBean	IncidentManagerLocal, IncidentMangerLocalHome	N/A
UserManagerBean	UserManagerLocal, UserManagerLocalHome	N/A
ReportManagerBean	ReportManagerLocal, ReportManagerLocalHome	N/A
ResourceManagerBean	ResourceManagerLocal, ReportManagerLocalHome	N/A

Finally, entity beans are needed to handle persistent data. Entity beans may be instantiated using bean-managed persistence (BMP) or container-managed persistence (CMP). BMP beans contain code to access the database; in CMP beans this function is handled by the container [7]. CMP beans were chosen for the DISarm system, and were generated using the built-in capabilities of the IDE to create CMPs from database tables. The DISarm entity beans

and related classes are listed in Table 4-4. They were deployed to the EJB Container of the J2EE Server.

Table 4-4: Entity Beans

Entity Bean	Interfaces	Key class
DM_Facility	DM_FacilityLocal, DM_FacilityLocalHome	DM_FacilityKey
DM_Contract	DM_ContractLocal, DM_ContractLocalHome	DM_ContractKey
DM_Company	DM_CompanyLocal, DM_CompanyLocalHome	DM_CompanyKey
DM_Manage	DM_ManageLocal, DM_ManageLocalHome	DM_ManageKey
DM_Incidence	DM_IncidenceLocal, DM_IncidenceLocalHome	DM_IncidenceKey
DM_UseFacility	DM_UseFacilityLocal, DM_UseFacilityLocalHome	DM_UseFacilityKey
DM_Use	DM_UseLocal, DM_UseLocalHome	DM_UseKey
DM_Resources	DM_ResourcesLocal, DM_ResourcesLocalHome	DM_ResourcesKey
DM_ResourceFood	DM_ResourceFoodLocal, DM_ResourceFoodLocalHome	DM_ResourceFoodKey
DM_Resource_Transport	DM_Resource_TransportLocal, DM_Resource_TransportLocalHome	DM_Resource_TransportKey
DM_Resource_Construct	DM_Resource_ Construct Local, DM_Resource_ Construct LocalHome	DM_Resource_ Construct Key
DM_User	DM_UserLocal, DM_UserLocalHome	DM_UserKey
DM_Report	DM_ReportLocal, DM_ReportLocalHome	DM_ReportKey
DM_Reported	DM_ReportedLocal, DM_ReportedLocalHome	DM_ReportedKey

Design Review

It is apparent that the DISarm components naturally fall into the MVC pattern when implemented in the J2EE framework. The relationship of the components of the initial DISarm model to MVC is shown in Figure 4-5.

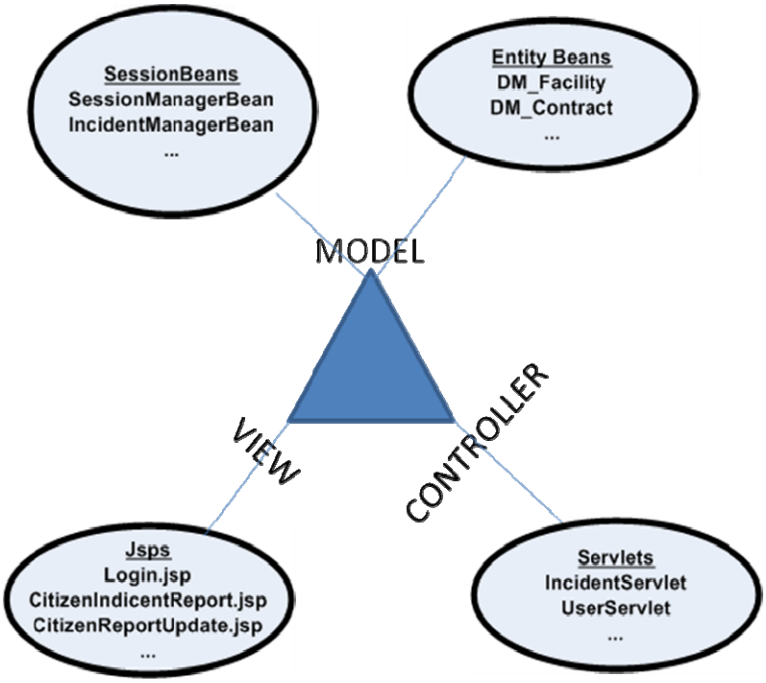

Figure 4-5: MVC – J2EE Relationship

The complexity of the J2EE implementation is substantiated by the number of classes required to instantiate the system. In order to make implementation manageable, it is crucial to take advantage of the automatic generation capabilities of the IDE.

Chapter 5: Additional System Requirements and Use Cases

As with most software systems, the needs of the users of the DISarm system will change over time. Additional system requirements came when the first module was completed. This formed a good case to experiment with the change impact to the original design.

Table 5-1: Additional System Requirements

Number	Requirement Statement
23	The DISarm system shall allow government officials to update the status of an incident (active/completed), that is, a government official can override a contractor response.
24	The DISarm system shall allow one contract to cover more than one facility.
25	The DISarm system shall allow citizens to register their current location during an evacuation (location/contact information/safety status).
26	The DISarm system shall allow citizens to update their location information.
27	The DISarm system shall have the capability to associate information a citizen enters for current location to the citizen's home address.
28	The DISarm system shall allow users to view a citizen's location information by searching on the citizen's name/home address.
29	The DISarm system shall allow government officials to enter location information on behalf of a displaced citizen (location/contact information/safety status).
30	The DISarm system shall allow government officials to update a citizen's location information.

The above requirement statements were analyzed to discover actors and use cases. There were no new actors added to the DISarm system by the additional requirements. However, it was determined that two use cases would require updating, and in addition several new use cases were added. The updated use case model is shown in Figure 5-1. Updated use cases are highlighted in yellow; additional use cases are highlighted in blue. The specifications for the additional use cases are detailed in Appendix 2.

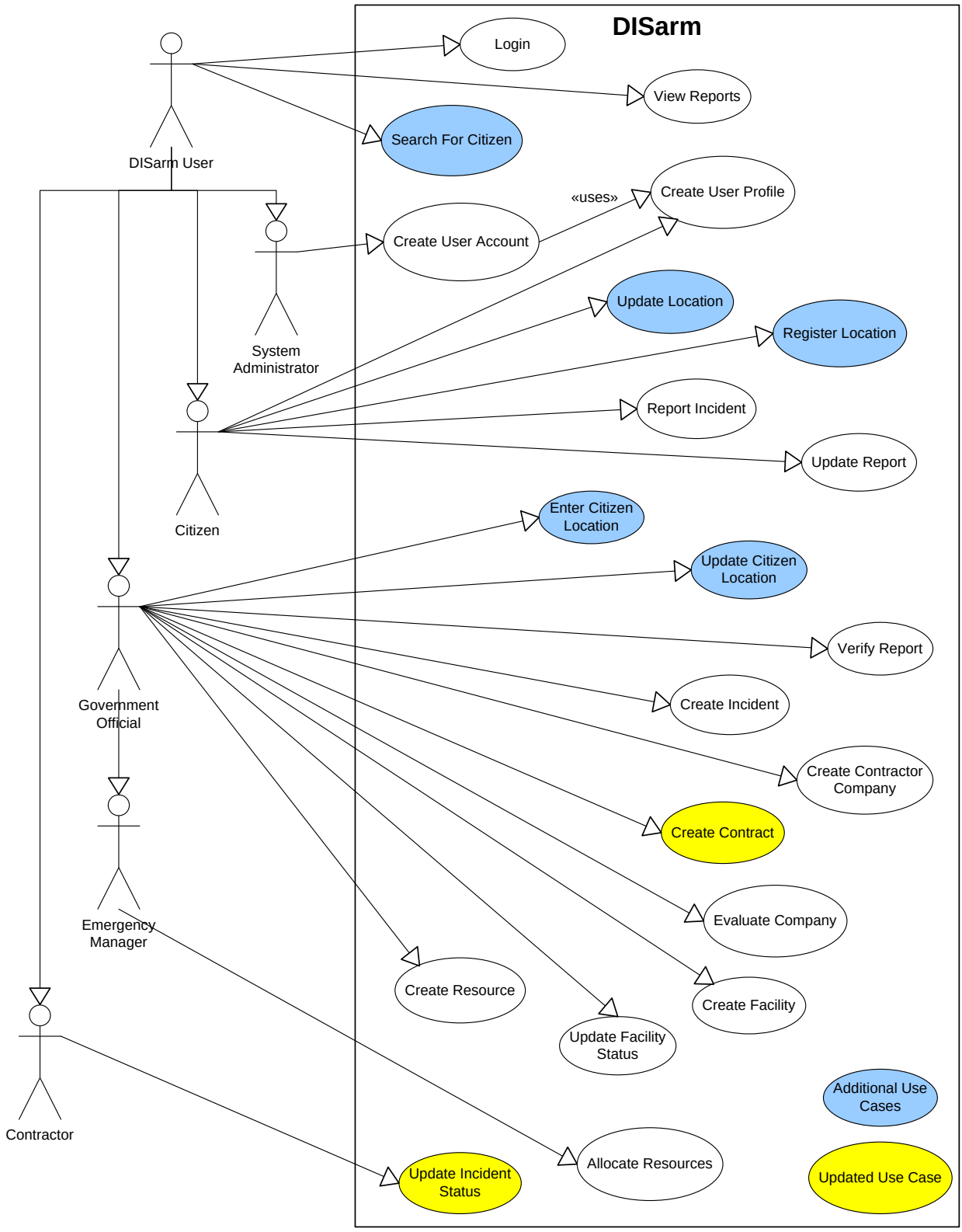

Figure 5-1: Updated Use Case Model

Chapter 6: Impact to Model of Updated and Additional Requirements

Three types of changes to the DISarm system were triggered by the additional requirements. The first type of change adds additional functionality within an existing module, but does not affect the data model. Requirement 23 is such a change. The next type of change updates existing functionality, but requires a change to underlying data structure to capture additional data. Finally, Requirements 25 through 30 encompass new system functionality, which require creation of new user interface components, the implementation of new methods, and additions to the data structure.

Use Case: Update Incident Status

The change to Requirement 23 adds functionality to allow a government official to override the status that a contractor has entered for an incident. The use case that traces to Requirement 23 is Update Incident Status, listed in the use case catalog in No. 8. The original and updated use cases are listed in Table 6-1, with the updates being highlighted in yellow.

Table 6-1: Original and Revised Use Cases -- Update Incident Status

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
8	Update Incident Status	Contractor	Citizen, Government Official, Emergency Manager	1. The Contractor selects an active incident report to review. 2. The DISarm System displays the selected report. 3. The Contractor enters notes about the incident. 4. The Contractor updates the status of the incident to closed. 5. <<alternative flow to #4>> If the Contractor chooses to save notes in progress, the status of the incident remains active. 6. Submit. 7. The DISarm System saves the updated information in the system.

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
8r	Update Incident Status	Contractor, Government Official	Citizen, Government Official, Emergency Manager, Contractor	1.The Contractor/Government Official selects an active incident report to review. 2. <<alternative flow to #1>> If the user is a Government Official, he/she may select a closed incident. 3. The DISarm System displays the selected report. 4. The Contractor/ Government Official enters notes about the incident (optional). 5. The Contractor/ Government Official updates the status of the incident to closed. 6. <<alternative flow to #4>> If the Contractor Government Official chooses to save notes in progress, the status of the incident remains active. 7. Submit. 8. If the Actor is the Contractor, the DISarm System saves the notes as Contractor notes, else the system saves the notes as Government Official notes. 9. The system saves the status of the incident

The design artifacts were reviewed to determine the effect of the changes to Update Incident Status. A review of the data model and other use cases showed that the data structure and methods needed to make this change were already in place in the system. This is because the use case Create Incident contains the provision that a government official may enter notes, so the design decision was made to simply append any new notes entered for Update Incident Status to existing notes (if any) using the data structure and methods already in place. It was possible to implement this change by updating the page currently in place for the contractor to set the incident status, namely IncidentStatusUpdate.jsp, and the underlying servlet, ResourceServlet, to allow a user with the role of government official to make updates. Therefore, the change to Update Incident Status can be classified as a minor change that has little impact, and it need not be considered further.

Use Case: Create Contract

The next change to the requirements had more effect. Additional Requirement 24 states that a contract may cover more than one facility. This requirement is covered by the Create Contract use case listed in the catalog as No. 10. The original and updated use cases are shown in Table 6-2, again with the differences being highlighted in yellow.

Table 6-2: Original and Revised Use Cases -- Create Contract

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
10	Create Contract	Government Official	Contractor, Emergency Manager	1. The Government Official selects to create a contract. 2. The Government Official selects the contract type. 3. The Government Official selects the company from a list of contractor companies in the DISarm System. 4. The Government Official selects the related facility from a list of facilities. 5. The Government Official enters a description of the contract. 6. Submit. 7. The contract information is stored in the system. 8. <<alternative flow to #3>> If the company is not in the system, the use case Create Contractor Company is performed and flow of control returns to step 3. 9. <<alternative flow to #4>> If the facility is not in the system, the use case Create Facility is performed and flow of control returns to step 4.

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
10r	Create Contract	Government Official	Contractor, Emergency Manager	1. The Government Official selects to create a contract. 2. The Government Official selects the contract type. 3. The Government Official selects the company from a list of contractor companies in the DISarm System. 4. The Government Official selects the related facility/facilities from a list of facilities. 5. The Government Official enters a description of the contract. 6. Submit. 7. The contract information is stored in the system. 8. <<alternative flow to #3>> If the company is not in the system, the use case Create Contractor Company is performed and flow of control returns to step 3. 9. <<alternative flow to #4>> If a required facility is not in the system, the use case Create Facility is performed and flow of control returns to step 4.

The update to this use case forced by new Requirement 24 has more effect that one might think, because it changes the relationship between a contract and a facility from a one-to-one relationship to a one-to-many relationship. The original data structure created to hold contract information is shown in Figure 6-1.

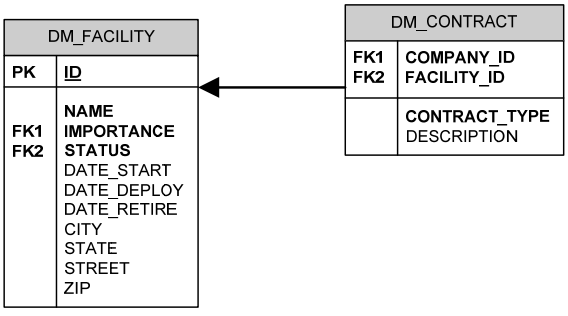

Figure 6-1: Contract-Facility Tables

If the data model is not changed, data normalization will be broken, since for every contract that controls more than one facility, the row in the database for that contract will be repeated in order to relate each facility to the contract. This will result in the contract type and description information for the same contract being stored multiple times in the database. This makes updating the data more difficult, and adds the risk that inconsistent data will be stored for a contract. To avoid this risk the data model will be updated to add a relationship between contract and facility, as shown in Figure 6-2.

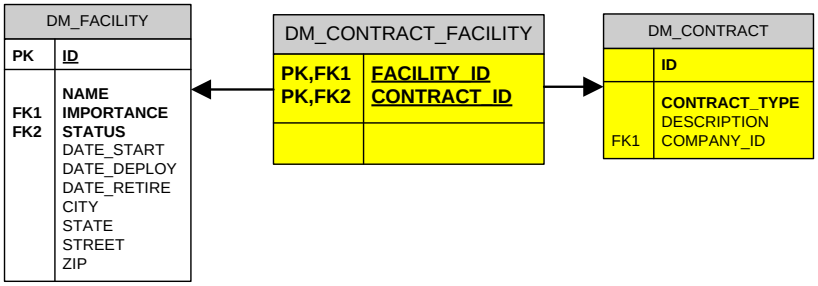

Figure 6-2: Revised Contract-Facility Tables

The updates to the data model will trigger revisions to the existing entity bean DM_Contract, its key class, DM_ContractKey, and its interfaces, DM_ContractLocal and DM_ContractLocalHome. It will also trigger the creation of a new entity bean and its related classes for DM_ContractFacility. In addition to the revision to the data model and corresponding entity beans, the web page CreateContract.jsp, the ResourceServlet, the ResourceManagerBean and SessionManager beans will also require modification. The web page will be modified to allow the selection of more than one facility, the resource servlet will be modified to accept a multi-dimensional array instead of a single value, the SessionManager will be modified to update the arguments passed by it to the ResourceManagerBean, and finally, the

ResourceManagerBean will be modified to loop through the facility array to create the relationship for each facility associated.

Additional Use Cases

Our third type of change is driven by additional requirements for the system. Additional Requirements 25 through 30, set out in Table 5-1, list the requirements for the location module of the DISarm system, and the new use cases corresponding to these requirements are detailed in Appendix 2 under numbers 17-21. The primary functionality added is set out in use case No. 17, Register Location, shown in Table 6-3 for the reader's convenience.

Table 6-3: Additional Use Case Register Location

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
17	Register Location	Citizen	DISarm Users	1. The citizen selects to Register Location. 2. The citizen enters information on current location: address, email, phone number. 3. The citizen enters notes on health/safety status. 4. Submit. 5. The DISarm system saves the citizen's current location information and associates it to this user.

The business rule inferred from Step 2 of the use case is that each citizen may register only one current location. Step 5 states that the information will be associated to the user, but it does not state how. Because of the one to one relationship between a user and current location, there is a choice of design for data storage: the current location can be saved as an attribute of the user; or, an additional data structure can be added to store the user's current location. Because the latter design offers more flexibility, *i.e.* it allows current location to be treated as a separate component, it was the design chosen. The new data structure is shown in Figure 6-3.

Note that there are two relationships between DM_USER and DM_USER_LOCATION. This is because the DM_USER_ID is the primary key of DM_USER_LOCATION, which creates the association of user to location, and secondly because the user id of the individual entering the information is stored to allow traceability to the government official who might enter this information on behalf of a citizen.

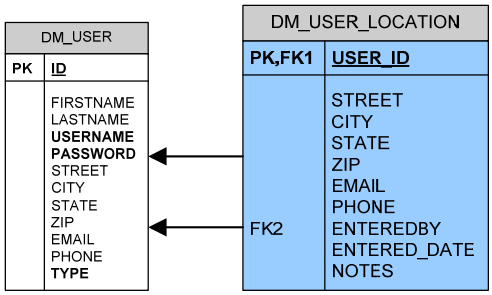

Figure 6-3: User Location Table

A new entity bean will be generated for DM_USER_LOCATION. A new JSP will be required, RegisterLocation.jsp, and based on our choice to maintain location as a separate module, a new servlet, UserLocationServlet, and a new session bean, UserLocationManagerBean, will be added.

The second interesting use case added is No. 19, Search For Citizen, which is shown in Table 6-4. This functionality will not touch the underlying data structure except to retrieve information, and thus no updates to the data model are necessary. It is treated as a separate component, however. This is because it is anticipated that other types of searches will be added in the future, and Search For Citizen will be a generic search that can be the foundation for other implementation. The additional classes needed for this component are: SearchForCitizen.jsp, SearchServlet, and SearchManagerBean.

Table 6-4: Search for Citizen

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
19	Search for Citizen	DISarm Users		1. The user selects to search for citizen. 2. The user enters search parameters (name, and/or home address). 3. Submit. 4. The DISarm system displays the location information associated to this citizen. 5. <<alternative flow to #4>> If no location information is associated to the citizen, the system displays the citizen's home address with a message that no updated information is available.

Review

From the preceding discussion, it can be seen that how much rework is needed when new requirements are discovered or existing requirements are updated is driven by the strength of the system design.

What should have been a relatively small change to the Create Contract use case required the regeneration of all of the entity beans, since we were using the automatic generation feature of our IDE to generate the beans and packed all the entity beans into one package. It could be argued that the better alternative would be to individually update the affected entity bean, generate a new bean for the new table, and thus avoid this issue. However, as stated in Chapter 2, we found it easier and more time efficient to allow the IDE to handle the chore of updating deployment descriptors and other references, rather than our updating the entity beans and then having to debug any problems that might ensue.

Although the remaining modifications triggered by the update to Create Contract do not seem to have a significant impact, the design review which was conducted to determine how to implement this change raised several issues.

One issue is how the functionality of the system was apportioned among the servlets and session beans. As stated above, the modifications to Create Contract touched the ResourceServlet and the ResourceManagerBean. But in modifying these classes to update contract functionality, we had the risk of inadvertently modifying unrelated code, since the code relating to resources and facilities was also included in this module. This uncovers a design flaw – our original choice to have only four servlets and their corresponding session beans results in classes that are too large and not cohesive. The original design of the ResourceServlet is reiterated in Table 6-5.

Table 6-5: ResourceServlet Functionality

Servlet	Function	Related JSPs
ResourceServlet	Handles requests to create companies, facilities, resources, and contracts and all requests for updates to resources	CreateContractorCompany.jsp, CreateResource.jsp, CreateFacility.jsp, UpdateFacilityStatus.jsp, CreateContract.jsp, EvaluateCompany.jsp, AllocateResource.jsp

At the time the original system was designed, it seemed appropriate to bundle all the functionality relating to companies, facilities, resources, and contracts together, since companies have contracts, companies use resources, and contracts control facilities. The better choice is to separate these components, since a change to a contract should not affect a resource or how a resource is allocated.

Another issue is that our original design does not readily permit multiple developers to work on the project concurrently. If, for example, an additional requirement had been included that necessitated an update to resources, the modification to the contracts piece and the

modification to the resource piece could not have been made concurrently, since the functionality is contained in the same component. Again, the better design is to separate these components, so that one developer may update the contract piece while the other updates resources, without having to worry about integrating updated code into the same component.

The new use cases for registering a location do not raise any additional issues. The lessons learned from a review of the create contract implementation guided us to the decision that location should be a separate component. To accomplish this goal, we will apply vertical partitioning.

The conclusion is that despite the partitioning of DISarm into the divisions of model, view, and controller, our initial design failed. Because of our reliance on the IDE to perform some of the tedious tasks of bean generation and deployment descriptor management, we fell into the trap of inadequately partitioning the system elements, and our design was not truly component based. It is apparent to us applying vertical partitioning to the MVC design becomes critical to control change. We set out the improvement of the model in Chapter 7.

Chapter 7: Improved Design

Updated Model Elements

As per the discussions in Chapter 6, we have concluded that our original design was flawed and inadequate. In this chapter, we update the design artifacts that were produced and documented in Chapter 5, using the process that will produce the grid pattern architecture. The first design artifact to be updated is the class diagram, shown in Figure 7-1.

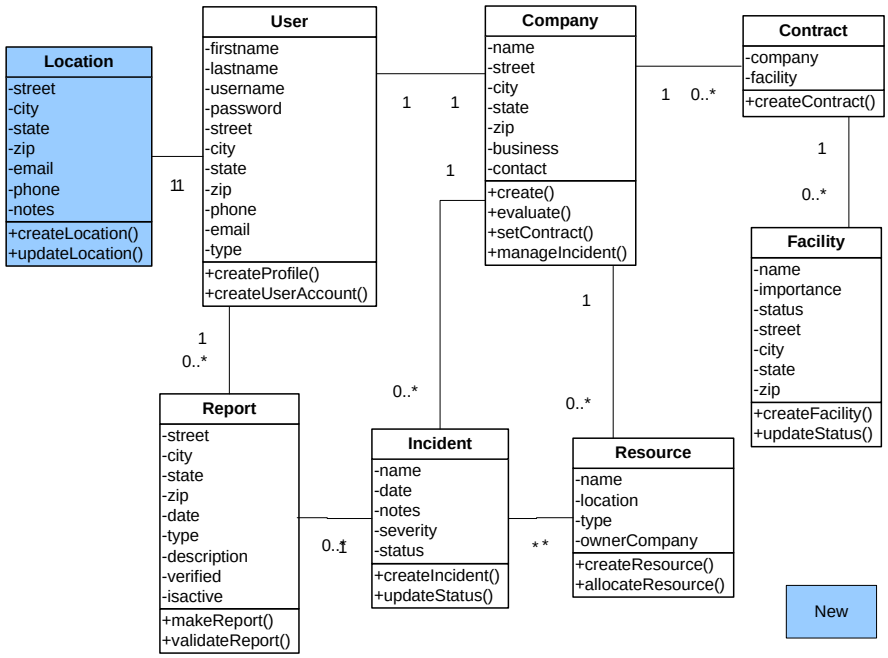

Figure 7-1: Updated Class Diagram

The class diagram now reflects the one to many relationship between a contract and a facility, and contains the new class needed to enable a user to register his or her location if forced to evacuate during a disaster. An argument could be made that since the relationship between user and location is a one to one relationship, the secondary location could be maintained as an attribute of a user. The design decision was to create the location as a separate class, as this

allows the location and the functionality required to instantiate it to be treated as a separate component.

The revised activity diagram for Update Incident Status is shown in Figure 7-2. The change needed to implementation can easily be seen in this diagram – we must provide functionality that allows a government official to reopen an inactive (closed) incident.

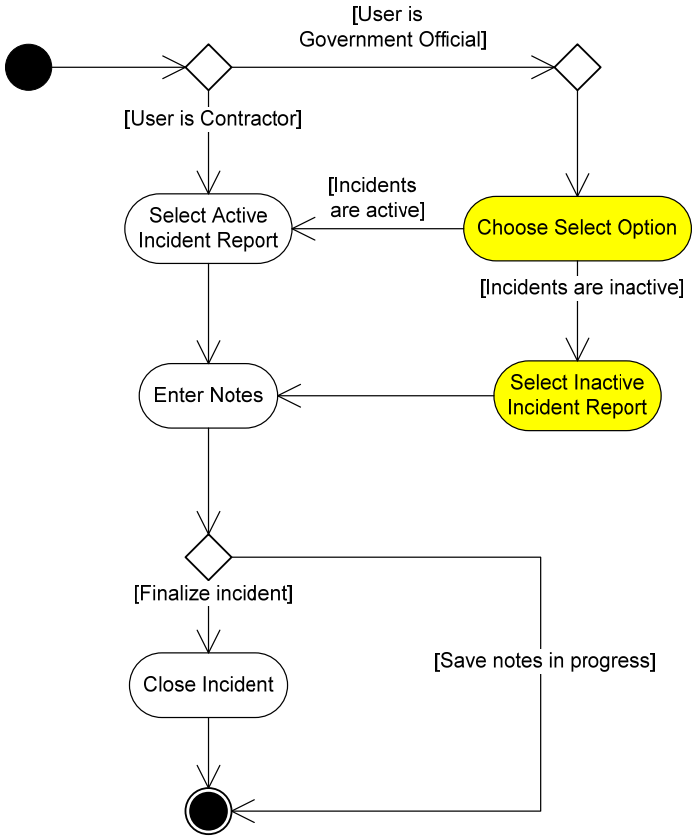

Figure 7-2: Revised Update Incident Status Activity Diagram

The next diagram, Figure 7-3, is the updated Create Contract Activity Diagram. The update to this diagram adds a flow of control that allows the user to continue to select or create facilities until the needed facilities have been chosen. The impact of this change will be discussed in the section on Design Review below.

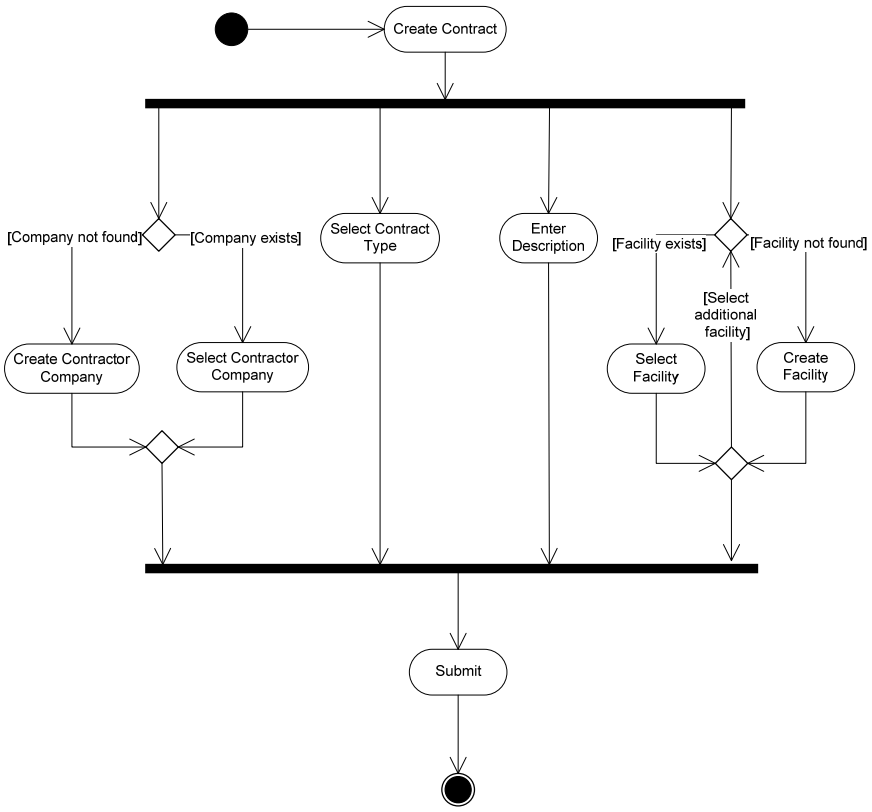

Figure 7-3: Revised Create Contract Activity Diagram

The last of our design artifacts updated from those created for the first iteration is the data model, shown in Figure 7-4. A new table for user location was added to store the relationship table between contract and facility. The contract table was also updated, to remove the previous primary key of company id and facility id, and to give it a primary key id instead. The company id became a foreign key relationship.

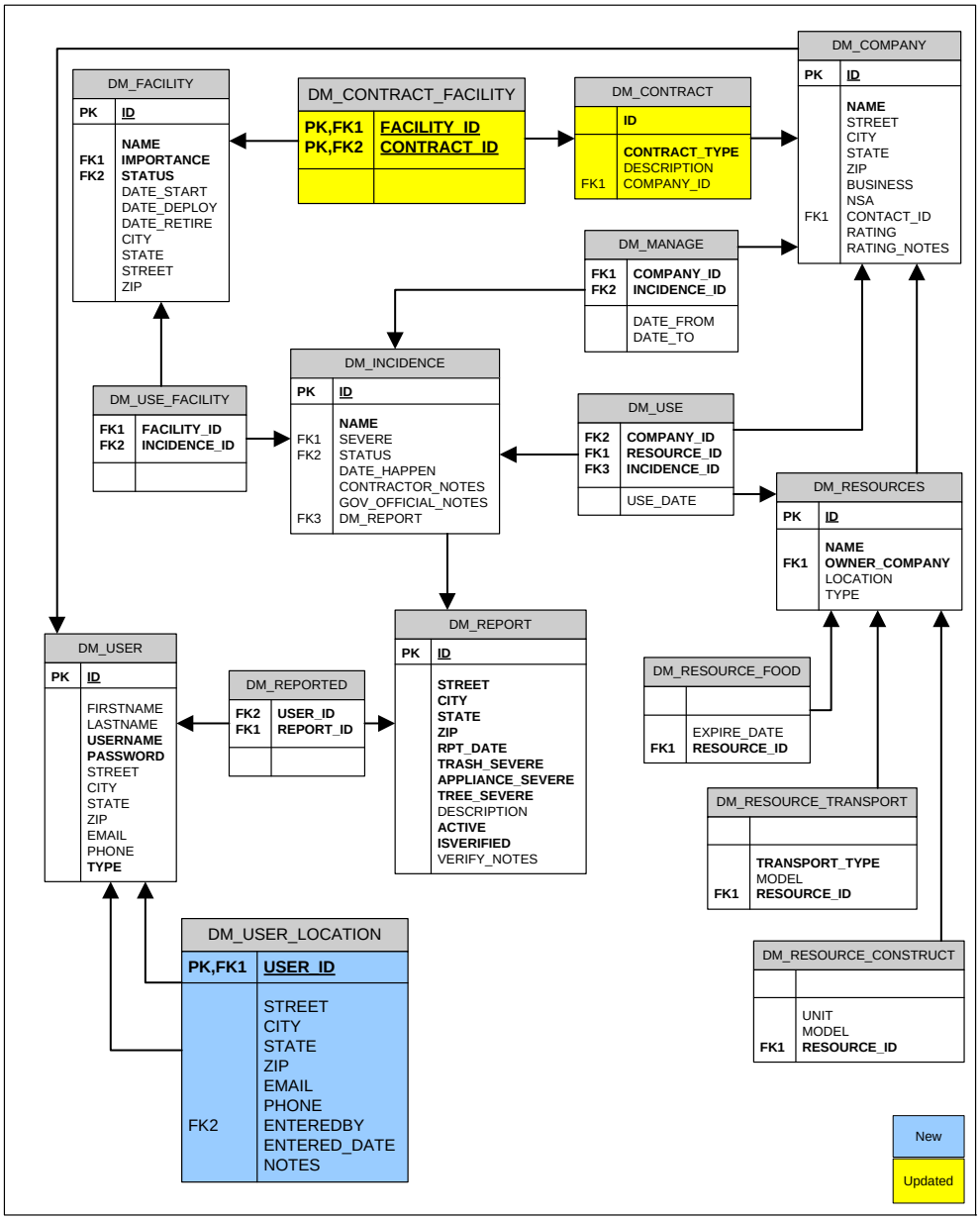

Figure 7-4: Updated Data Model

In addition to updating the existing design artifacts from iteration one, we had one more design chore. This was to review the use cases and group them into coherent packages. This use-case grouping formed the basis from which we created our grid partitions for implementation. The use case packages are show in Figure 7-5.

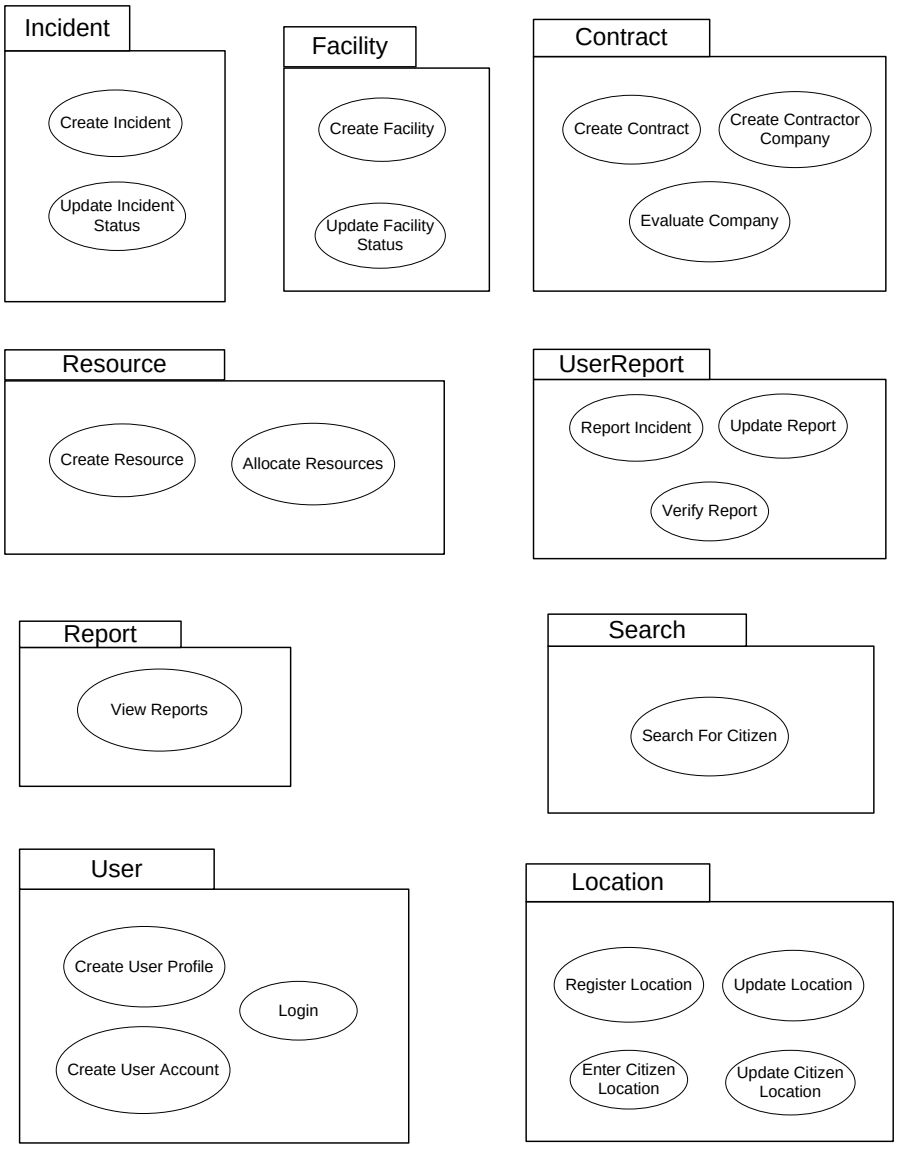

Figure 7-5: Use Case Packages

Updated Implementation

In the first iteration, the data model completed the design effort and implementation began when the JSPs, servlets, session beans, and entity beans needed to create the system were identified. Because the initial design was found to be inadequate, an additional design artifact was added to partition the use cases into packages. Once the updates to the JSPs, servlets and

beans are identified, we will create diagrams to show our partitioning scheme, which in our case will be a set of detailed package diagrams showing implementation classes.

There are five new JSPs identified for the updated system. They are added in Table 7-1, and highlighted in light blue.

Table 7-1: Updated JSPs

JSPs		
Login.jsp	CreateUserProfile.jsp	CreateIncident.jsp
CitizenIncidentReport.jsp	UpdateUserProfile.jsp	IncidentStatusUpdate.jsp
CitizenReportUpdate.jsp	VerifyCitizenReport.jsp	CreateContractorCompany.jsp
CreateResource.jsp	AllocateResource.jsp	CreateUserAccount.jsp
CreateContract.jsp	CreateFacility.jsp	ViewCitizenReport.jsp
EvaluateCompany.jsp	UpdateFacilityStatus.jsp	ViewContractors.jsp
ViewIncident.jsp	ViewResources.jsp	ViewFacilities.jsp
ViewContracts.jsp	ViewUserAccounts.jsp	
RegisterLocation.jsp	UpdateLocation.jsp	SearchForCitizen.jsp
AdminEnterLocation.jsp	AdminUpdateLocation.jsp	

In order to make the system more component-based, some functionality was moved to new servlets, namely, the UserReportServlet, ContractorServlet and FacilityServlet. Since these servlets would have been created in the original model if it had been properly partitioned, they are not highlighted in Table 7-2. After partitioning was applied, the servlets that must be updated due to changes in the use cases are the IncidentServlet and the ContractorServlet. In addition, two new servlets were needed to accommodate the additional functionality added by the updated use cases. These are the UserLocationServlet and the SearchServlet. The names of the new servlets are highlighted in light blue in Table 7-2; the updated servlets are highlighted in yellow.

Table 7-2: Updated Servlets

Servlet	Function	Related JSPs
UserServlet	Handles user accounts	Login.jsp CreateUserProfile.jsp, UpdateUserProfile.jsp, CreateUserAccount.jsp
UserLocationServlet	Handles requests to register displaced citizens' locations, and updates to locations	RegisterLocation.jsp, UpdateLocation.jsp, AdminEnterLocation.jsp, AdminUpdateLocation.jsp
SearchServlet	Handles requests to search for citizen locations. (Will handle other types of searches in future.)	SearchForCitizen.jsp
UserReportServlet	Handles citizen requests to make reports and updates to citizen reports.	CitizenIncidentReport.jsp, CitizenReportUpdate.jsp, VerifyCitizenReport.jsp
IncidentServlet	Handles requests to create incidents, and updates to incidents.	CreateIncident.jsp, IncidentStatusUpdate.jsp
ResourceServlet	Handles requests to create and allocate resources	CreateResource.jsp, AllocateResource.jsp
ContractorServlet	Handles all requests related to contractors and contracts	CreateContractorCompany.jsp, CreateContract.jsp, EvaluateCompany.jsp
FacilityServlet	Handles all requests related to facilities	CreateFacility.jsp, UpdateFacilityStatus.jsp
ReportServlet	Handles all requests to view reports	ViewIncident.jsp, ViewContracts.jsp, ViewCitizenReport.jsp, ViewResources.jsp, ViewUserAccounts.jsp, ViewContractors.jsp, ViewFacilities.jsp

The next artifact updated was the list of session beans needed. These were handled in a manner similar to the servlets – first the existing servlets were partitioned; then the servlets that would need updating were determined; finally, new servlets needed for functionality added by additional use cases were identified. These are shown in Table 7-3. New beans are highlighted in light blue; updated beans in yellow.

Table 7-3: Updated Session Beans

Bean	Local Interfaces	Remote Interface
SessionManagerBean	SessionManagerHome	SessionManager
UserManagerBean	UserManagerLocal, UserManagerLocalHome	
UserLocationManagerBean	UserLocationManagerLocal, UserLocationManagerLocalHome	
SearchManagerBean	SearchManagerLocal, SearchManagerLocalHome	
UserReportManagerBean	UserReportManagerLocal, UserReportManagerLocalHome	
IncidentManagerBean	IncidentManagerLocal, IncidentMangerLocalHome	
ResourceManagerBean	ResourceManagerLocal, ReportManagerLocalHome	
ContractorManagerBean	ContractorManagerLocal, ContractorManagerLocalHome	
FacilityManagerBean	FacilityManagerLocal, FacilityManagerLocalHome	
ReportManagerBean	ReportManagerLocal, ReportManagerLocalHome	

Finally, the list of entity beans, shown in Table 7-4, was updated to include the new entity beans. The same highlighting scheme was followed: new entity beans are highlighted in light blue; updated in yellow.

Table 7-4: Updated Entity Beans

Entity Bean	Interfaces	Key class
DM_Facility	DM_FacilityLocal, DM_FacilityLocalHome	DM_FacilityKey
DM_Contract	DM_ContractLocal, DM_ContractLocalHome	DM_ContractKey
DM_Contract_Facility	DM_Contract_FacilityLocal, DM_Contract_FacilityLocalHome	DM_Contract_Facility_Key
DM_Company	DM_CompanyLocal, DM_CompanyLocalHome	DM_CompanyKey
DM_Manage	DM_ManageLocal, DM_ManageLocalHome	DM_ManageKey
DM_Incidence	DM_IncidenceLocal, DM_IncidenceLocalHome	DM_IncidenceKey
DM_UseFacility	DM_UseFacilityLocal, DM_UseFacilityLocalHome	DM_UseFacilityKey
DM_Use	DM_UseLocal, DM_UseLocalHome	DM_UseKey
DM_Resources	DM_ResourcesLocal, DM_ResourcesLocalHome	DM_ResourcesKey
DM_ResourceFood	DM_ResourceFoodLocal, DM_ResourceFoodLocalHome	DM_ResourceFoodKey
DM_Resource_Transport	DM_Resource_TransportLocal, DM_Resource_TransportLocalHome	DM_Resource_TransportKey
DM_Resource_Construct	DM_Resource_ Construct Local, DM_Resource_ Construct LocalHome	DM_Resource_ Construct Key
DM_User	DM_UserLocal, DM_UserLocalHome	DM_UserKey
DM_Report	DM_ReportLocal, DM_ReportLocalHome	DM_ReportKey
DM_Reported	DM_ReportedLocal, DM_ReportedLocalHome	DM_ReportedKey
DM_User_Location	DM_User_LocationLocal, DM_User_LocationLocalHome	DM_User_LocationKey

This completes the updates to the artifacts generated in iteration one. We are now ready to partition JSPs, servlets, session beans and entity beans according to our grid plan. The resulting artifact will be a set of package diagrams for each layer of the MVC architecture. Rather than using a standard UML package diagram, we have chosen to list the contents of each package in a table format, as this structure is easier to review and understand. This contract packages are shown in Figure 7-3, the location packages are shown in Figure 7-4, and the report packages are shown in Figure 7-5. The remaining packages are listed in Appendix 3.

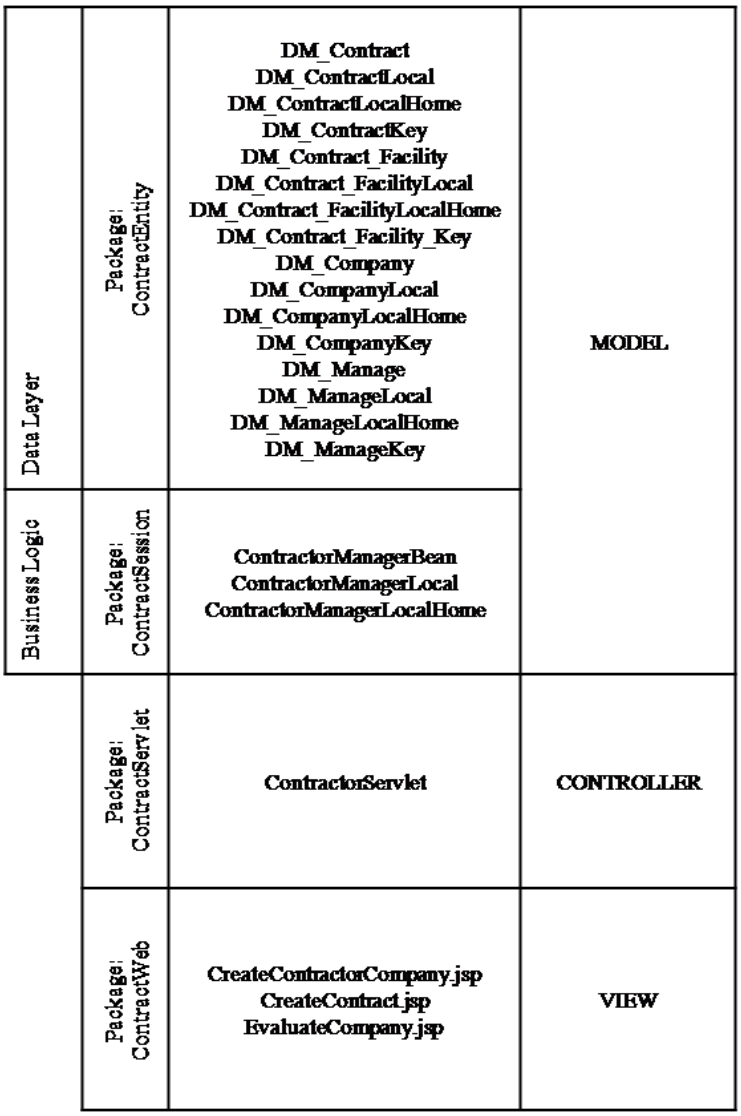

Figure 7-6: Contract Partitioning

The above grouping of the contract packages illustrates how few objects will be touched by the revisions stemming from the updated requirements.

Data Layer	Package: Location Entity	DM_User_Location DM_User_LocationLocal DM_User_LocationLocalHome DM_User_LocationKey	MODEL
Business Logic	Package: Location Session	UserLocationManagerBean UserLocationManagerLocal UserLocationManagerLocalHome	
	Package: Location Servlet	UserLocationServlet	CONTROLLER
	Package: Location Web	RegisterLocation.jsp UpdateLocation.jsp AdminEnterLocation.jsp AdminUpdateLocation.jsp	VIEW

Figure 7-7: Location

The figure showing the location package grid is included here to illustrate benefits of our plan. Since the entity beans to be generated relate to a new table in the data model, and since we have grouped these into their own package, no changes will be needed to existing beans. We can therefore use the capabilities of our IDE to handle the tedious chore of generating these beans. Furthermore, we can readily parcel out development tasks based on these partitions.

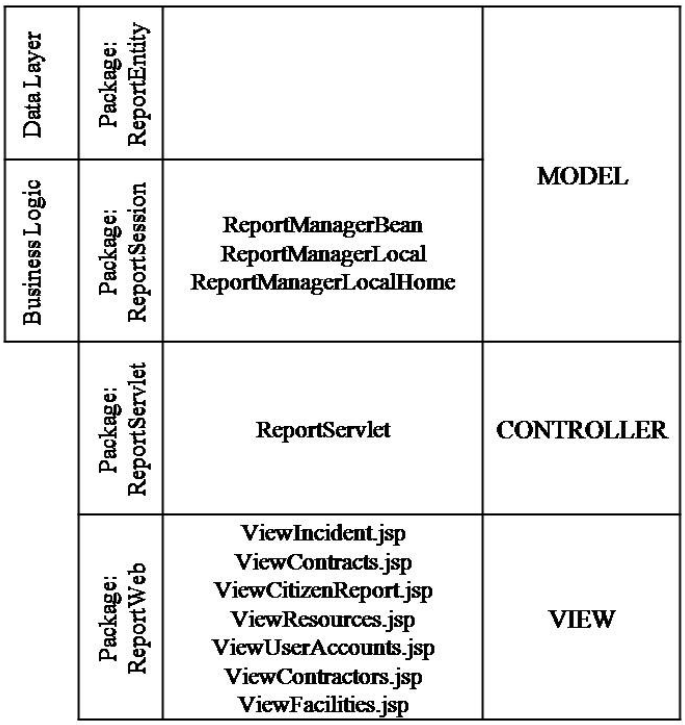

Figure 7-8: Report

There are no entity bean components shown in the Report diagram. This is because the only functionality of the report package is to retrieve and display data. Thus, it can be seen that in addition to the other benefits of grid partitioning, it forms an intuitive visual reference to system functionality.

Design Review of Updated Model

We now consider the effect the updates and additional use cases would have had if the original model had been grid-partitioned.

Use Case: Update Incident Status

The updates to this use case triggered only minimal changes in implementation. Therefore, there are no anticipated differences in effect between the two models.

Use Case: Create Contract

As stated in Chapter 6, revisions to this use case triggered updates to the data model, and corresponding revisions to the existing entity bean DM_Contract, its key class, DM_ContractKey, and its interfaces, DM_ContractLocal and DM_ContractLocalHome, as well as the new entity bean and its related classes for DM_ContractFacility. In addition, the web page CreateContract.jsp, the ResourceServlet, the ResourceManagerBean and SessionManager beans will be modified as set out above.

Under the partitioning scheme set out in this chapter, the effects of regeneration of the DM_Contract bean and adding the new bean would be localized to the one package, rather than affecting all fourteen of the original entity beans. This has major time-saving benefits. If the entity beans have not been modified from the generated code, the package can simply be dropped and the beans regenerated, allowing the IDE to handle all of the issues relating to updating references and deployment descriptors. On the other hand, if one or more of the beans have been modified, making automatic regeneration a less attractive choice, there are still benefits to be realized. There will be less code to review when updates are made, and changes will be isolated from the rest of the code, making debugging easier in the event errors are made.

The remaining changes caused by this update will likewise be easier to manage. Since the contract component now has its own manager and servlet, the benefits of less code to review, change being isolated, and easier debugging apply here as well as to the entity beans.

The final benefit to be realized by partitioning into components is a significant one, and that is that implementation can be logically and conveniently apportioned among developers. Since the components are relatively independent of each other, one developer can make all the necessary modifications to the contract package, and then integrate the changed classes into the

baseline system. Or, one developer can be assigned all of the modifications to entity beans, while others are assigning other portions of the development. This makes version control and software maintenance a much more manageable task.

Additional Use Cases

The additional use cases entirely relate to new functionality. The design decision to incorporate this new functionality as a separate component using grid-partitioning makes change management reasonable, and development easier. Most of the benefits are similar to those realized in for the Create Contract use case. Similar benefits are that development of the new component can take place in isolation from the remainder of the system; development responsibilities can be apportioned to multiple developers, for instance by assigning one developer the web interface and servlet, and another session bean and entity beans, and debugging code will be easier since problems will be isolated within the module. Additional benefits are that a new package of entity beans will be generated; no existing beans will be touched, and integration of the component will be a matter of importing the code and updating navigation.

Comparison Results

The final step in the design review is to measure the differences between the two modeling methods. In order to do this, it was necessary to determine the number of classes that comprised the original model, and the categories to which each was assigned. There were 23 JSPs, 4 servlets, 5 session beans, and 14 entity beans[1].

[1] For convenience, only the entity beans are counted, and not the corresponding interfaces and key classes.

The impact of the changes on the original design was then considered and the original classes affected by the updates were counted. The percentage of original classes updated and/or regenerated was computed. Finally, the new classes that were needed to implement the additional requirements were counted. The results are shown in Table 7-5.

Table 7-5: Affect of Change on Original Model

Implementation category	Number of classes in category	Number of classes affected by updates	Percent affected by updates	New classes added for additional requirements
JSPs	23	2	8%	5
Servlets	4	2	50%	2
Session Beans	5	3	60%	2
Entity Beans	14	14	100%	2

It is easy to see that the original design was flawed. 100% of the entity beans were regenerated in order to update one existing bean and add one new bean. In addition, 60% of the session beans and 50% of the servlets were affected because too much functionality was grouped together into one class.

The revised model was then reviewed to see if the results would be improved. The counts are shown in Table 7-6. The additional servlets (the UserReportServlet, ContractorServlet, and FacilityServlet) and session beans (the UserReportManagerBean, ContractorManagerBean, and FacilityManagerBean) created in the second iteration to properly partition the original model are counted in the "Number of classes in category" in Table 7-6, since these classes were not added as a result of updates to user requirements and use cases. Rather, these classes were added to correct design defects in the original model, and thus should be included in the base count when measuring the impact of change to the model.

Table 7-6: Affect of Change on Revised Model

Implementation category	Number of classes in category	Number of classes affected by updates	Percent affected by updates	New classes added for additional requirements
JSPs	23	2	8%	5
Servlets	7	2	28%	2
Session Beans	8	3	37%	2
Entity Beans	14	3	21%	2

In comparing the two sets of figures, we see that the new classes added for additional requirements did not change. This is to be expected since we treated the user location as a separate component in both models. When we look at the percentage of classes affected by updates, however, it is a different case. The number of servlets affected was reduced to 28% from 50%; the session beans to 37% from 60%, and only 21% of the entity beans were affected, rather than 100%. Based on these reductions, the benefits of grid-partitioning are obvious.

Chapter 8: Conclusion

This thesis has addressed the system architecture for enterprise level applications from two methodologies. In order to do so, we created a system architecture for the DISarm system, first using the horizontal multi-tier partitioning afforded by MVC. We then applied a process to impose vertical partitioning on top of our horizontal partitioning in order to achieve a grid partitioning of the system.

We found that we followed certain steps in order to achieve the improved model, which we set out herein as our guidelines. Enforcement of the grid partitioning should be a concern starting from Step 9. Facilitating the traceability of between the elements in different models is a key to accomplishing Steps 10 and 11. Therefore, the efforts to establish the model diagrams and documents that help trace the correspondence, such as tables or spreadsheets, are critical in supporting grid-partitioning.

1. Gather user requirements
2. Analyze requirements
3. Create a use case model and specifications for the use cases based on the requirements
4. Add any additional requirements found during this process to the use case model
5. Analyze the use cases
6. Create a class diagram based on the use cases and user requirements
7. Create activity diagrams, or other diagrams, for any interesting or complicated use cases.
8. Create a data model
9. Group the use cases in logical packages according to functionality
10. Identify implementation classes needed for JSPs, servlets, session beans and entity beans
11. Package the implementation classes in accordance with the use case packaging

When the design process was completed for the second iteration of the DISarm architecture, we compared the two models and found that significant benefits were realized when we used the grid partitioning approach:

- Increased ability to use the underlying tools afforded by the IDE, such as automatic bean generation
- Ability to apportion tasks among developers in a logical manner
- Less time spent debugging
- Improved change control management

For these reasons, we concluded that following the grid partitioning approach to be a sound practice, and critical to control change.

References

[1] Ivar Jacobson, Grady Booch, Jim Rumbaugh The Unified Software Development Process, ; 1st edition, Addison-Wesley Professional, February 4, 1999.

[2] Arlow, Jim and Neustadt, Ila, UML 2 and the Unified Process, 2nd edition, Pearson Education, Inc., 2005.

[3] Kruchten, Philippe, "The 4+1 View of Architecture", IEEE Software, 12(6) Nov. 1995.

[4] Bodoff, Stephanie, et al, The J2EE Tutorial, Second Edition, Sun Microsystems, 2004.

[5] Moore, Bill, et al, WebSphere Application Server – Express V6 Developers Guide and Development Examples, ibm.com/redbooks, October 2005.

[6] http://www.uml.org

[7]Anderson, Gail and Anderson, Paul, Enterprise JavaBeans Component Architecture, Prentice Hall, 2002.

[8] http://java.sun.com/blueprints/patterns/MVC-detailed.html

[9]Moore, Bill, et al, WebSphere Application Server-Express: A Development Example for New Developers, ibm.com/redbooks, November 2003.

Appendix 1: Use Case Catalog for Initial DISarm System

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
1	Login[2]	DISarm users	DISarm System	1. Login to DISarm using: a. New account b. Existing account
2	Report Incident	Citizen	Contractor, Government Official	1. The citizen selects Report Incident. 2. The citizen enters the address of incident. 3. The citizen selects the type of incident (trash, appliance, tree, garbage). 4. The citizen enters the severity of the incident. 5. The citizen enters a description. 6. Submit. 7. The DISarm system saves the incident report, sets the report to active, sets the report date to the current date, and associates the citizen to the report.
3	Update Report	Citizen	Contractor, Government Official	1. The citizen selects Update Report. 2. The citizen selects the address of the incident report to be updated. 3. The DISarm System validates the citizen as the user who created the incident report. 4. The System displays the report for editing. 5. <<alternative flow to #4>> If the incident report is closed, the report is shown in a non-editable format. 6. The citizen updates editable fields in the report (active/closed, description, severity). 7. Submit. 8. The System stores the updated report.

[2] All use cases, with the exception of View Reports, have as a precondition Login to the DISarm system.

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
4	Create User Profile	Citizen	DISarm System	1. The user selects to create a user profile. 2. The System prompts for user information: firstname, lastname, username, password, address, email (optional), phone number (optional). 3. Submit. 4. The DISarm System validates the username is unique, sets the user type to “Citizen”, and saves the user’s information in the system. 5. <<alternative flow to #4>> If the username is not unique, the System prompts for a unique username and flow of control returns to step 2.
5	View Reports	DISarm users		1. The DISarm user selects to View Reports. 2. The Disarm System displays available reports. 3. The user selects a report to view. 4. The System displays the report.
6	Verify Report	Government Official	Citizen, Contractor	1. The Government Official selects an incident report to review. 2. The DISarm System displays the selected report. 3. The Government Official sets the incident to “Verified” or “False Report”. 4. The Government Official enters notes (optional). 5. Submit. 6. The DISarm system saves the updated information in the system. 7. <<alternative flow to #5>> if the incident is set to “False” the DISarm system sets the incident to closed and saves the updated information in the system.

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
7	Create Incident	Government Official	Citizen, Contractor, Emergency Manager	1. The Government Official selects Create Incident. 2. The Government Official enters information on the incident: name, date of occurrence, notes. 3. The Government Official selects the severity of the incidence. 4. The Government Official selects the status of the incidence. 5. The Government Official selects the managing company from a list of contractor companies in the system. 6. Submit. 7. The DISarm system verifies that the name of the incident is unique, stores the incident in the System and associates the managing company to the incident. 8. <<alternative flow to #5>> If the company is not in the system, the use case Create Contractor Company is performed and flow of control returns to step 5.
8	Update Incident Status	Contractor	Citizen, Government Official, Emergency Manager	1. The Contractor selects an active incident report to review. 2. The DISarm System displays the selected report. 3. The Contractor enters notes about the incident. 4. The Contractor updates the status of the incident to closed. 5. <<alternative flow to #4>> If the Contractor chooses to save notes in progress, the status of the incident remains active. 6. Submit. 7. The DISarm System saves the updated information in the system.

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
9	Create Contractor Company	Government Official	Contractor, Emergency Manager	1. The Government Official selects to create a contractor company. 2. The System prompts for company information: name, address and type of business. 3. The Government Official selects the company contact from a list of Contractor users in the DISarm system. 4. Submit. 5. The DISarm System validates the company name is unique, associates the contact to the company, and saves the company information in the system. 6. <<alternative flow to #3>> If the company contact is not in the DISarm System, the System allows the Government Official to create a user account for the contact and flow of control returns to step 3. 7. <<alternative flow to #5>> If the company name is not unique, the DISarm system prompts the Government Official for a unique company name and flow of control returns to step 5.
10	Create Contract	Government Official	Contractor, Emergency Manager	1. The Government Official selects to create a contract. 2. The Government Official selects the contract type. 3. The Government Official selects the company from a list of contractor companies in the DISarm System. 4. The Government Official selects the related facility from a list of facilities. 5. The Government Official enters a description of the contract. 6. Submit. 7. The contract information is stored in the system. 8. <<alternative flow to #3>> If the company is not in the system, the use case Create Contractor Company is performed and flow of control returns to step 3. 9. <<alternative flow to #4>> If the facility is not in the system, the use case Create Facility is performed and flow of control returns to step 4.

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
11	Evaluate Company	Government Official	Contractor, Emergency Manager	1. The Government Official selects Evaluate Company. 2. The Government Official selects a rating for the company. 3. The Government Official enters notes. 4. Submit. 5. The rating information is updated in the system.
12	Create Facility	Government Official	Contractor, Emergency Manager	1. The Government Official selects to create a facility. 2. The DISarm System prompts for facility information: name, importance, status, start date, deploy date, address of facility. 3. Submit. 4. The DISarm System stores the facility information in the system.
13	Update Facility Status	Contractor	Government Official Emergency Manager	1. The Contractor selects to Update Facility Status. 2. The DISarm System displays a list of facilities under the contractor's control. 3. The contractor chooses a facility to update. 4. The contractor sets the facility status. 5. The contractor enters the deploy or retire date (optional). 6. Submit. 7. The DISarm System stores the updated information.
14	Create Resource	Government Official	Contractor, Emergency Manager	1. The Government Official selects to create a resource. 2. The Government Official selects the type of resource (transport, food, construct). 3. The Government Official enters the name and location of the resource. 4. The Government Official selects the owner company from a list of companies in the DISarm System. 5. Submit. 6. The DISarm System creates the resource in the system. 7. <<alternative flow to #4>> If the owner company is not in the DISarm System, the use case Create Contractor Company is performed and flow of control returns to step 4.

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
15	Allocate Resources	Emergency Manager	Citizen, Contractor, Government Official	1. The Emergency Manager selects to Allocate Resources. 2. The Emergency Manager selects an incidence to which resources will be allocated from a list of active incidents in the DISarm System. 3. The Emergency Manager selects resources to allocate to the incident from a list of resources in the DISarm System. 4. The Emergency Manager selects facilities to allocate to the incidence. 5. Submit. 6. The System associates the selected resources and facilities to the incidence.
16	Create User Account	Application Manager	Government Official, Emergency Manager	1. The Application Manager selects to create a user account. 2. The System prompts for type of user account: Government Official or Emergency Manager. 3. The System prompts for user information: firstname, lastname, username, password, address, email (optional), and phone number (optional). 4. Submit. 5. The DISarm System validates the username is unique, sets the user type to the account type selected, and saves the user's information in the system. 6. <<alternative flow to #5>> If the username is not unique, the System prompts for a unique username and flow of control returns to step 3.

Appendix 2: Updated Use Case Catalog

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
8r	Update Incident Status[3]	Contractor, Government Official	Citizen, Government Official, Emergency Manager, Contractor	1. The Contractor/Government Official selects an active incident report to review. 2. <<alternative flow to #1>> If the user is a Government Official, he/she may select a closed incident. 3. The DISarm System displays the selected report. 4. The Contractor/ Government Official enters notes about the incident (optional). 5. The Contractor/ Government Official updates the status of the incident to closed. 6. <<alternative flow to #4>> If the Contractor Government Official chooses to save notes in progress, the status of the incident remains active. 7. Submit. 8. If the Actor is the Contractor, the DISarm System saves the notes as Contractor notes, else the system saves the notes as Government Official notes. 9. The system saves the status of the incident.

[3] This use case is updated to add the Government Official as a primary actor.

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
10r	Create Contract	Government Official	Contractor, Emergency Manager	1. The Government Official selects to create a contract. 2. The Government Official selects the contract type. 3. The Government Official selects the company from a list of contractor companies in the DISarm System. 4. The Government Official selects the related facility/facilities from a list of facilities. 5. The Government Official enters a description of the contract. 6. Submit. 7. The contract information is stored in the system. 8. <<alternative flow to #3>> If the company is not in the system, the use case Create Contractor Company is performed and flow of control returns to step 3. 9. <<alternative flow to #4>> If a required facility is not in the system, the use case Create Facility is performed and flow of control returns to step 4.
17	Register Location	Citizen	DISarm Users	6. The citizen selects to Register Location. 7. The citizen enters information on current location: address, email, phone number. 8. The citizen enters notes on health/safety status. 9. Submit. 10. The DISarm system saves the citizen's current location information and associates it to this user.
18	Update Location	Citizen	DISarm Users	1. The citizen selects to Update Location. 2. The citizen updates address, email, phone number, notes. 3. Submit. 4. The DISarm system saves the updated information in the system.

ID	Use Case Name	Primary Actor	Participant/ Beneficiary Actors	Steps
19	Search for Citizen	DISarm Users		6. The user selects to search for citizen. 7. The user enters search parameters (name, and/or home address). 8. Submit. 9. The DISarm system displays the location information associated to this citizen. 10. <<alternative flow to #4>> If no location information is associated to the citizen, the system displays the citizen's home address with a message that no updated information is available.
20	Enter Citizen Location	Government Official	DISarm Users	1. The Government Official selects Enter Citizen Location. 2. The Government Official selects the home address of the citizen for whom the information will be entered. 3. The Government Official enters information on citizen's current location: address, email, phone number. 4. The Government Official enters notes on the citizen's health/safety status. 5. Submit. 6. The DISarm system saves the information in the system and stores the user id of the government official as the user entering the information.
21	Update Citizen Location	Government Official	DISarm Users	1. The Government Official selects to Update Citizen Location. 2. The Government Official selects the home address of the citizen for whom the information will be updated. 3. The Government Official updates the citizen's information. 4. The DISarm saves the updated information and stores the user id of the government official as the user entering the information.

Appendix 3: Grid Partitions

<table>
<tr><td rowspan="2">Data Layer</td><td>Package: UserEntity</td><td>DM_User
DM_UserLocal
DM_UserLocalHome
DM_UserKey</td><td rowspan="2">MODEL</td></tr>
<tr><td>Business Logic</td><td>Package: UserSession</td><td>UserManagerBean
UserManagerLocal
UserManagerLocalHome</td></tr>
<tr><td></td><td>Package: UserServlet</td><td>UserServlet</td><td>CONTROLLER</td></tr>
<tr><td></td><td>Package: UserWeb</td><td>Login.jsp
CreateUserProfile.jsp
UpdateUserProfile.jsp
CreateUserAccount.jsp</td><td>VIEW</td></tr>
</table>

Figure A- 1: User

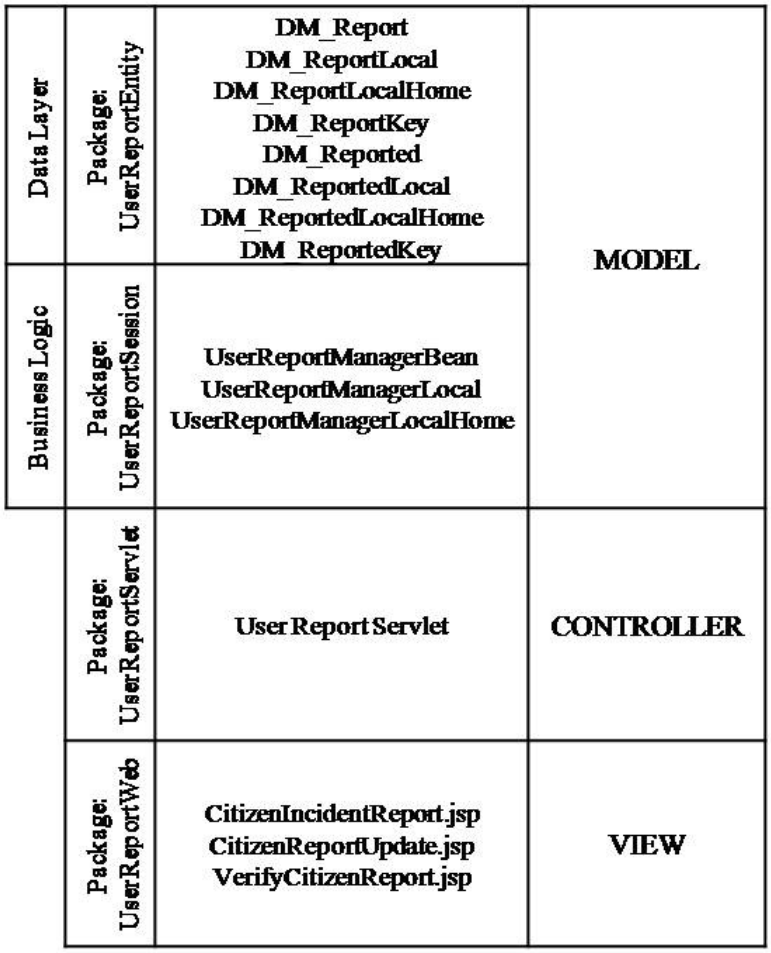

Figure A- 2: UserReport

Data Layer	Package: SearchEntity		MODEL
Business Logic	Package: SearchSession	SearchManagerBean SearchManagerLocal SearchManagerLocalHome	
	Package: SearchServlet	Search Servlet	CONTROLLER
	Package: SearchWeb	SearchForCitizen.jsp	VIEW

Figure A- 3: Search

Data Layer	Package: FacilityEntity	DM_Facility DM_FacilityLocal DM_FacilityLocalHome DM_FacilityKey	MODEL
Business Logic	Package: FacilitySession	FacilityManagerBean FacilityManagerLocal FacilityManagerLocalHome	
	Package: FacilityServlet	FacilityServlet	CONTROLLER
	Package: FacilityWeb	CreateFacility.jsp UpdateFacilityStatus.jsp	VIEW

Figure A- 4: Facility

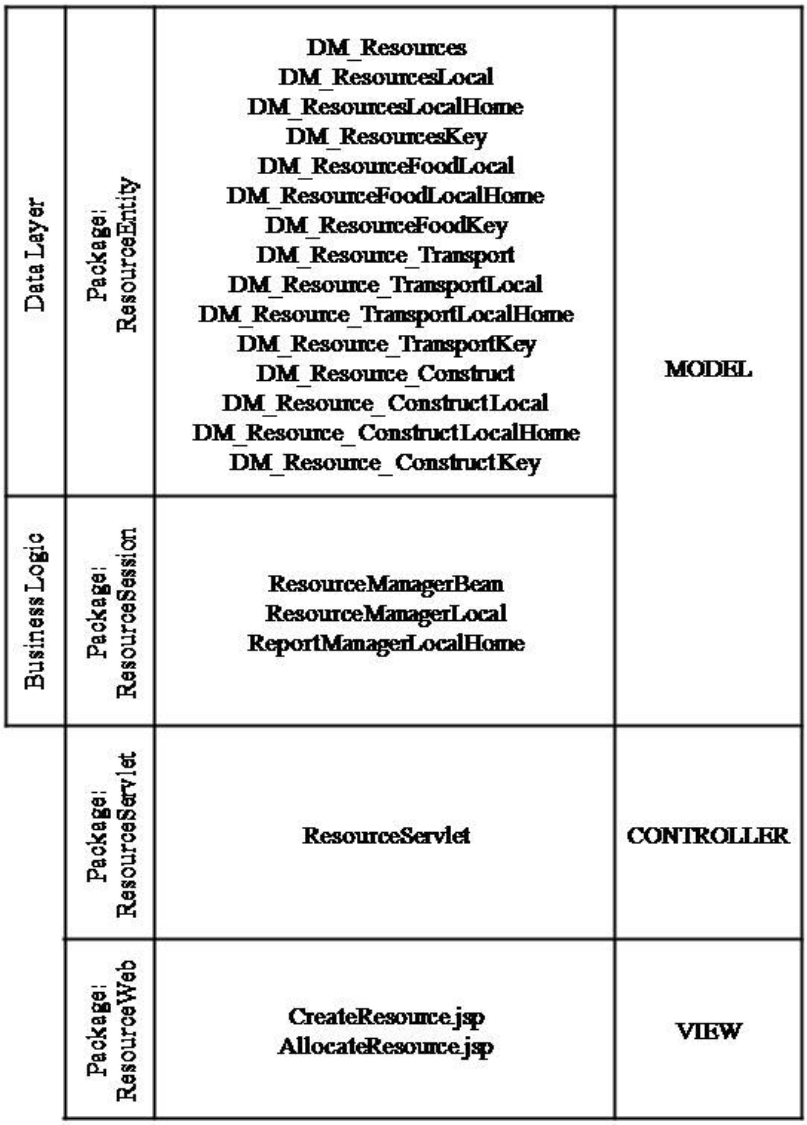

Figure A- 5: Resource

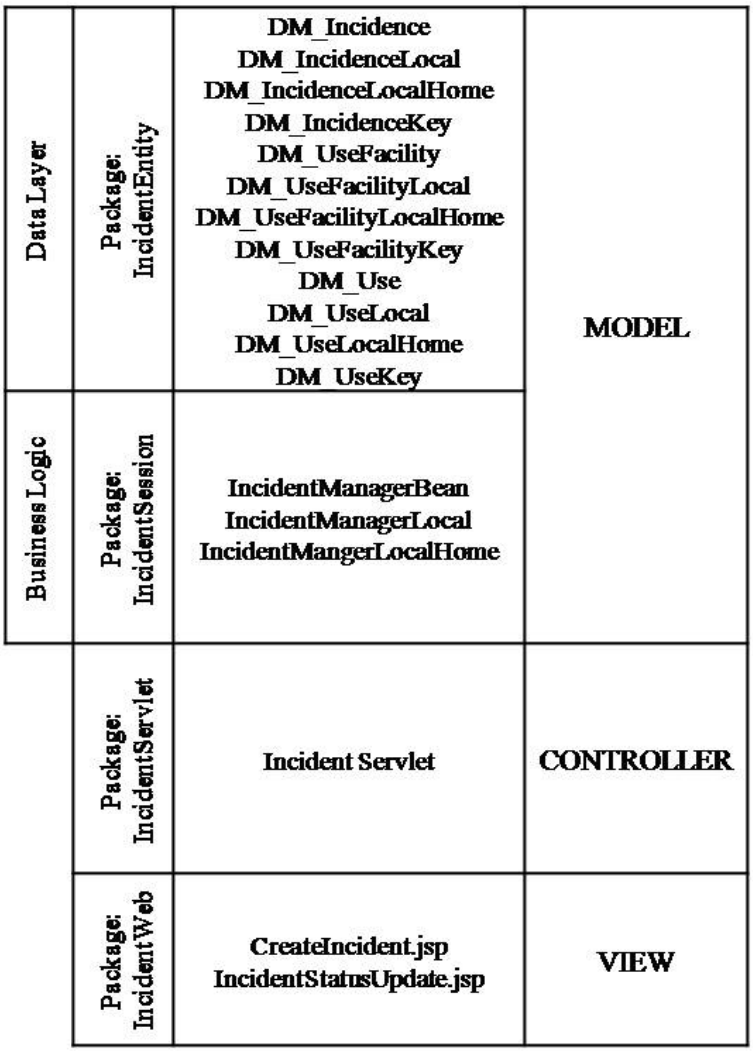

Figure A- 6: Incident

Vita

Aline Sewell Vogt was born in Bay St. Louis, Mississippi and received her B.S. from the University of New Orleans in December of 2003.