A SERVICE ORIENTED ARCHITECTURE FOR DATA-DRIVEN DECISION SUPPORT SYSTEMS

By

Ian James Cottingham

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Matthew Dwyer

Lincoln, Nebraska

July 2009

A SERVICE ORIENTED ARCHITECTURE FOR DATA-DRIVEN DECISION SUPPORT SYSTEMS

Ian James Cottingham, M.S.

University of Nebraska, 2009

Advisor: Matthew Dwyer

Decision support is a complex activity requiring high-availability of many, seemingly unrelated, data sources. Creating software systems to support decision making activities across multiple domains poses significant challenges in terms of high-availability of data and the interoperability of both data and software components. These data integration challenge are magnified when one considers that in many domains, data must be analyzed both spatially and temporally in order to provide effective decision support. The complexities of data integration in Decision Support Systems can be addressed by providing software operations exposed at data integration points. With data integration points exposed through a unified service framework, data of many formats, resolutions, and sources can be much more easily integrated through layering software component operations within the framework.

By developing to a unified framework specification for the integration of domain datasets, software engineers can create components to encapsulate data operations published as services within the framework. Common data type descriptors and operations provided by the framework can be utilized in the component development to ensure that, once published, framework components can interact in standard ways. The standard interaction of data-driven components results in the implicit integration of domain datasets; data integration becomes a function of component development. The framework architecture provides for highly cohesive and loosely couple components and preventing integration from coming at the expense of existing component functionality.

This thesis presents The Framework for Integrated Risk Management (FIRM), a unified framework to support data-driven decision support in agricultural domain through data integration and service publication.

ACKNOWLEDGEMENTS

This work was made possible through a cooperative agreement between The University of Nebraska - Lincoln and the United States Department of Agriculture Risk Management Agency[1]. Untold hours of software development went into the implementation of the architecture presented here. Without the combined efforts of one of the most dedicated software development teams that I have been privileged to lead, this implementation would not have been possible. Special thanks go to the members of that team: Shawn Baden, Jon Dokulil, Flora Jiang, Brian Knapp, Ben Kutsch, Laura Meerkatz, Debolina Ray, Derrick Stolee, Jesse Whidden, and Xueming Wu. Additional thanks go to Doctors Matthew Dwyer and Steve Goddard who advised on aspects of this research, as well as the faculty and staff of the National Drought Mitigation Center who provided domain expertise critical to understanding the complex processes encapsulated by this framework.

[1] Drought Risk, Impact and Mitigation Information System (USDA 05-IE-0831-0254)

This work is dedicated to my mentor and friend, Ed Smierciak, who, through many long hours of writing code together at IBM, inspired me to love the art of good software design.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

MOTIVATION

The growth of Internet applications used for everyday activity has resulted in an increase in demand for high-availability data systems in research and decision support communities. This demand is being met by the development of vast data repositories and applications that provide users with information from current stock quotes to what friends are planning over the weekend. These data are increasingly being used to make decisions and take actions. Whether recognized as such, many of the web applications that people are interacting with every day could be considered to be decision support systems. The increased awareness of data-driven web applications combined with their integration into new decision workflows results in the greater availability of data to support decision-making activities. Additionally, a new class of information worker emerges who is proficient in isolating data necessary to more effectively make decisions.

A traditional decision support system is defined as an information system that "draws on transaction processing systems and interacts with other parts of the overall information system to support the decision-making activities of managers and knowledge workers in [an] organization" [1]. The workflows encapsulated by traditional decision support systems typically revolve around a user feeding parameters into the system, the system computing some result based on the parameters, and the results being presented in a usable format. In this type of decision support system, workflows encapsulate domain processes based on the user classification. One of the more frequent uses of this approach is user partitioning in software wizards; screens for Novice, Intermediate, and Advanced users are separated into unique workflows from the same set source of data [1,2,3,4]. In this approach, each wizard incorporates differing levels of functionality leading to a similar result.

With the rise of web-based applications and the broader availability of data through the Internet, decision support began to incorporate aspects of knowledge management. Static workflows gave way to dynamic information discovery based on many sources of data. In this model, data are partitioned based on their relationship to other data and are frequently fed through domain-centric models or workflows to

create information. In these systems data and information can also be combined to create higher-order data that are termed knowledge [4]. The data segmentation that has grown out of the introduction of knowledge management principles has resulted in a significant focus being placed on domain models and workflows as well as the acquisition of domain specific data sets. Where decision support systems were previously driven by user controlled models and workflows, they are now largely being driven by availability of domain-relevant datasets. User interactions with these systems require less focus to be placed on the type of user and more on the kind of information that is needed. Domain models and workflows become transparent to the overall tasks of data, information, and knowledge discovery. The architectural constraints that are placed on what one can term a Data-Driven decision support systems pose a number of new challenges to system development. These challenges arise principally from the role of data integration in system development.

With a focus on the availability of data to form information and knowledge, a Data-Driven decision support system must be able to quickly integrate high volumes of unrelated data in order to remain relevant. The availability of remote datasets accessed through platform independent protocols must be easily leveraged by data-driven systems to provide detailed views and assessments within the considered domain. This availability requirement poses a significant data integration challenge to software developers creating such data-driven systems. Datasets can be as varied as the data themselves, coming from static data repositories, network aware devices and sensors, or other applications through web services. Web applications and services are becoming increasingly location-aware, further complicating the data integration process by amassing large quantities of GIS data. This creates an orthogonal data format to be integrated in the system. These GIS data repositories can easily grow to multiple terabytes in size and are frequently delivered through geo-spatial web services external to the system. With much of the data being external to the system, highly adaptive software must be created to facilitate the integration and availability of data.

A second challenge arises when one considers the domain specific models and workflows encapsulated by decision support systems. While generally transparent to the decision support activity, these models and workflows are required to create information and knowledge. The results of one

process can become an internal dataset required by another process to create information or build knowledge and thereby portray an accurate view of some domain state. External data can be integrated to form an internal dataset supporting processes or published as a dataset to be consumed by other systems. This requirement results in the abstract view of a decision support system being composed of many connected paths passing through data integration points. At any given point, another dataset can intersect the current path, requiring that the data be normalized in some way to allow integration to take place. In the context of the software design, this problem can be easily addressed by creating a normalized central dataset to support system processes. However, when considered in relation to the evolving data landscape of the Internet, size of data sets, and the need to have current results, static normalization of the data is often not an option.

Data normalization can, however, be accomplished in software through the application of Component-Oriented (CO) design principles [5,6,9,22]. By encapsulating dataset operations into independent software components, processes interact with the dataset through software components rather than directly. These software components give system developers a finer level of control overdatasets that are disparately consumed and uniformly reproduced within the system. System operations are partitioned into groups based on the data operations they wrap. This highly cohesive and loosely coupled implementation pattern is a hallmark of Object-Oriented (OO) system design, and can be very effectively extended to support data-driven software development [6,10,23].

The primary goals of CO architecture are to address data flow between components and create an infrastructure upon which to build use case conformant software [5,6,22,23]. In the context of decision support systems, data integration requirements result in a loose view of date flow between components. The architecture is less concerned with which components data may pass through and is more concerned with how data passes in and out of the component. In order to realize this black box view of component data flow, the architecture must enforce a strict data model. Logic for accessing an individual dataset or executing some domain specific process or workflow is isolated in unique components. Data integration is accomplished through runtime composition of component calls.

APPROACH

While this architectural view of a data-driven decision support system aligns with the unique challenges presented by data integration, it fails to address the resulting component integration problems that would arise from its implementation. Distribution and configuration of components around a unified application model is needed to support application development in this paradigm. To address this problem, this thesis proposes the implementation of a Service Oriented Architecture (SOA),which integrates middleware services with the component model described above. The SOA is realized through the implementation of a set of core foundation services consisting of the following elements:

1. *A centralized management framework.*

   In a SOA, components can be distributed across multiple applications and systems. In order to coordinate communication between components and determine the availability of a component to process a request, a centralized controller is required. The centralized management framework is introduced to act as a controller for all component invocations. The centralized management framework is based on a design pattern for partitioning component management operations and lookup services. Through the use of the Manager/Accessor design pattern, component interactions are loosely coupled with centralized management framework operations. The centralized management framework creates abstract groupings of components around common configuration needs such as shared data source or domain workflows.

2. *Core integration services API*

   Supporting the high-level configuration and discovery services provided by the centralized management framework is a set of service APIs for managing component interaction with the framework. Common crosscutting services like access control, logging, notification, data source connectivity, and domain data model operations are made available to components through the core integration services. The core integration services support the black box composition of components within the framework and represents a bridge between framework functionality and middleware services like connection pooling and load balancing. The core integration

services also facilitate data sharing through the publication of services backed by component implementations.

3. *An integration component runtime.*

   An implementation of private core integration services API classes is made available to applications through the integrated component runtime. The integrated component runtime manages communication between framework components or consumer applications and the centralized management framework. The integrated component runtime allows these framework communication classes to remain private while at the same time being available to callers to manage centralized management framework connections and component call marshalling. The integrated component runtime provides the runtime context for all components both internal and external to the framework.

The proposed framework implementation is built using several key Java technologies including: Java Enterprise Edition (Java EE) 1.5 [8], Java Management Extensions [7], and Java Persistence Architecture [11]. The framework takes advantage of many of the service and component oriented constructs made available by these technologies. Additionally, the Java EE specification provides a number of middleware services that are available to the framework to support required deployment and runtime operations. While Java technology provides a platform upon which to implement the proposed architecture, the key concepts presented here are not unique to Java; this architecture could be implemented with other component-based development platforms.

OUTLINE

The remainder of this thesis is structured as follows. Chapter 2 presents background and related work. Chapter 3 presents the formal architectural specification for a service-oriented decision support framework, discussing framework configuration, service providers, and application middleware and deployment considerations. The Framework for Integrated Risk Management is introduced as a reference implementation framework to support application development in the agricultural decision support domain; Chapters 4 and 5 focus on elements of this framework. Chapter 4 introduces the domain model

for The Framework for Integrated Risk Management, discussing various aspects of the model and their support function for a component model. Chapter 5 presents the component model and discusses how it is used to support development of application software. Chapter 6 presents a case study of two decision support systems that used the framework, and Chapter 7 concludes with contributions and a discussion of future work.

CHAPTER 2: BACKGROUND AND RELATED WORK

This chapter discusses the framework that is a predecessor to this work and drove to the approach described by this thesis. In addition to the legacy framework, several key technologies that emerged at the outset of this work are discussed. Some of these technologies were used in the development described later and others were used as the basis for conceptualizing several architectural patterns and methodologies that were later adopted by the framework presented by this thesis.

THE 3CO FRAMEWORK

The 3Co framework was a framework developed to support the design and implementation of components and component connectors to facilitate data integration. The framework focused on the design of the components and connectors, as well as the coordination of components by runtime callers [4]. These aspects, the three 'cos' of the framework, were used as the basis for the development of decision support systems in the agricultural domain. The resulting decision support, The National Agricultural Decision Support System, provided a basis for evaluation of the 3Co framework and had significant influence on the work presented here. 3Co and NADSS and the issues that arose with integration of new datasets during development of the systems were the driving factors in the design of the architecture presented by this thesis.

The 3Co architecture was a layered architecture, grouping components into one of four classifications based on the data they would provide.

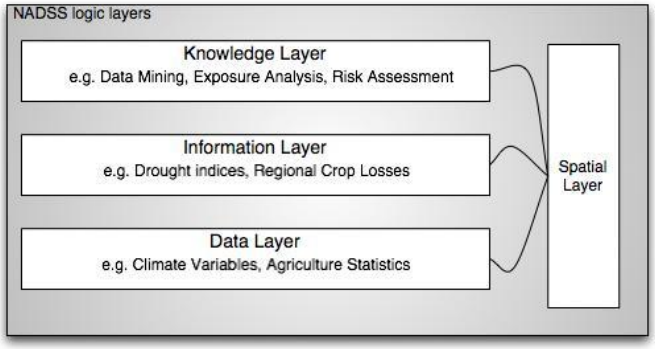

Figure 1: The 3Co framework architecture

The high-level architectural view of 3Co is depicted in Figure 1. This hierarchical layered view of the framework was realized by the classification of components as data, information or knowledge components. Lower level components were used by higher-level components to generate data as defined by the layer. Information layer components, for example, computed data values based on the application of models to data layer component data. Knowledge components, in turn, used information and data for computing values. The framework included an orthogonal spatial layer [12] that, through connector definitions, allowed values from each of the three other layers to be rendered spatially.

3Co provided a basis for the development of decision support tools and supported the development of tools around connectors. Discussed extensively in Chapter 6, 3Co was achieved success at developing decision support systems in a distributed environment. It was, however, the static connector definitions that presented the largest obstacle to data integration within the framework. The challenges to integrate data using the framework led to the consideration of a more agile and dynamic architectural methodology to overcome data integration obstacles. The use of SOA as a basis for that design was motivated by the more monolithic layered approach taken by 3Co. The proposed architecture builds off many 3Co concepts of component orientation, generation of data through component connection, and use of connectors for application modeling. These concepts are enhanced in several ways by the availability of new technology and design styles discussed throughout this chapter.

SERVICE ORIENTED ARCHITECTURE

Service Oriented Architecture (SOA) is a way of composing an application through the consumption of disparate component functionality delivered through a common network-aware interface, a service. These applications aggregate many different data sources in what is commonly termed a mash-up in order to provide their functionality. SOA builds on distributed system concepts that have existed for many years [5,6,10]. The main difference is that SOA requires design parameters aimed at simplifying many application integration challenges commonly associated with distributed system implementation [10]. A SOA is composed of many services exposing operations mapping to application use cases. A service has the following key properties [10]:

- Platform Independence – The interface definition exposed by a service does not contain any platform specific assertions or parameterization. This property gives rise to the use of XML-based messaging over HTTP-based protocols for operation invocation. This property can be considered one of the key components to the tacit linking of web services and SOA.
- Dynamic Service Discovery and Linking – Services register themselves in some way at runtime allowing consumers of the service to bind to and invoke operations at runtime. This dynamic binding is generally facilitated through a remote protocol such as IIOP or HTTP. This attribute requires that services can be invoked.
- Service Autonomy – Services exist independent of each other, internalizing all operations and state. Other services can interact with the service through the well-defined contract only. All other information about a service is hidden.

These properties allow SOA applications to easily integrate application functionality provided by other components or remote applications. The ease of integration, paired with the increased use of the World Wide Web for application delivery, is resulting in the emergence of SOA as the de-facto standard for Internet application development.

From the standpoint of data integration, the concept of application integration poses several potential uses. The elements of platform independence, linking, and autonomy can all be used to support abstraction of elements in a component architecture for the integration of data as a service. The architectural model also supports significant decoupling between its components, allowing for flexibility in the design and implementation of architectural support elements. These aspects of SOA make it appealing as a basis for a framework for data integration.

JAVA EE 5

The Java Platform Enterprise Edition (Java EE) defines a standard architecture for developing applications that "combine existing enterprise information systems (EISs) with new business functions that deliver services to users" [8]. The Java EE specification, developed as JSR 244 in 2005 and 2006, introduced significant improvements over J2EE. These improvements dealt primarily with the abstraction

of the underlying framework from components being developed in it. The goal of the these improvements centered around the use of plain old Java objects (POJOs) to be able to act as enterprise Java beans (EJBs) with no more than annotations. EJBs, or the Java EE definition of a component, could be easily constructed to incorporate only elements of the computational functionality required to implement the needed business logic. Architecture services, like transaction and persistence, were also incorporated into the component implementation through the use of annotations and deployment injection. Functionality was incorporated into a component as it was deployed to a middleware runtime, not as it was developed or packaged during development.

The updates made with the Java EE specification not only make the platform an appealing choice for the development of a framework for data integration, many of the elements of abstraction found in the specification motivate the implementation strategy for such a framework. Component isolation through middleware service injection can be used to provide significant decoupling between components and framework services while maintaining service functionality. This design consideration, a hallmark of object and component oriented design, is an important aspect of the proposed method for data integration. Java and Java EE can also be used as the overall runtime platform for the proposed framework, allowing the framework to be designed and implemented independently while at the same time incorporating several robust services in deployment.

JBOSS APPLICATION SERVER AND FRAMEWORKS

The use of JSR 244 elements in the design and implementation of the architecture presented in this work were heavily influenced by the JBoss implementation of the specification. JBoss approached the concept of service injection in EJB components through the application of aspect oriented programming (AOP) [25], developing a weaving compiler to introduce service functionality to components. The use of AOP as a basis for the separation of concerns within the framework motivated the development of a set of integration services discussed later. The use of AOP was initially considered and then abandoned due to overhead associated with development. Many of the AOP concepts, however, can be seen in the use of proxy services and general structure of services discussed later.

JBoss also introduced the concept of the use of the Java Management Extension framework to create deployable service components to support runtime configuration as well as EJB extensions to create component connections [24]. The extended deployment structure was adapted to form the basis for the later discussed centralized management framework. The deployment structure of the service components was altered slightly to allow deployment external to the framework application, however the annotation set used mimicked the JBoss implementation with only minor additions needed throughout development.

The design patterns used by JBoss in early implementations of a JSR 244 compliant application server had the greatest influence on the design of the framework for data integration. While 3Co motivated the solution to the problem, JBoss motivated the implementation of the solution in its use of existing Java technologies to accomplish a number of service abstractions in support of deployed application component models. Both of these factors, combined with the rich framework development environment provided by Java EE were all incorporated in the development of a service oriented architecture for data integration.

CHAPTER 3: FRAMEWORK ARCHITECTURE

Service Oriented Architectures (SOAs) are component-oriented systems that publish component business logic for consumption by other systems [10]. The central building blocks in these systems are the components, which encapsulate some domain-specific functionality. Components are designed to be black box computational units, which work together to comprise the complete application functionality. In the Java EE specification this black box structure is accomplished by providing a component interface and implementation. An EJB container in application middleware manages the component instances, called Enterprise Java Beans (EJBs); generally this is done using the Java Naming Directory Interface (JNDI). Callers use the component interface to interact with component instances through proxies created by the JNDI context. These proxies can be remote or local. Proxies to remote components use protocols like IIOP, CORBA, or JNP to invoke methods on the component instances and can be deployed across JVMs; local components restrict calls to the same JVM instance in which the proxy is running. The underlying object instance created by the JNDI context is hidden from the callers through the component interface, allowing transparent component co-location and facilitating system distribution.

With this view of the component model, one can conceive of a component framework tied together with services to support unification of data flows between components. By implementing a framework architecture, component implementations are easily integrated by building them against the common service structure and services can be easily created against common component elements. The framework architecture presented here is made up of three abstract elements, the centralized management framework, The core integration services and the integrated component runtime. These elements together form the core foundation services. Each core foundation services service works separately to collectively constitute an environment in which components can be deployed and interaction with other components facilitated. This high-level view of the framework architecture represents an abstract specification for the framework itself, omitting the component and data model implementations. Implementations of this architecture provide data and component models, integrated

with each of the Core Foundation Services. This abstraction is chosen primarily to allow application of the architecture within multiple decision domains. Decision support systems span many varying domains making the abstraction of an efficacious cross-domain application or component model difficult. By providing architectural elements to facilitate loose communication between components through a well-defined, domain-centric data model, wider adoption of the architecture for decision support system development is supported.

The goal of the framework architecture is to isolate domain processes, models, and data repositories by encapsulating interactions with these elements in component invocations. This component-oriented architectural organization of framework logic facilitates the inclusion of new data and domain methods with little impact to the overall system. Only those components making use of new additions need to be modified, and then only if the inclusion impacts the data model. In well-defined application domains a data model will infrequently require change, providing near transparent inclusion of components in the implementing system. A secondary goal of the architecture is to create an environment where consumers of component data, information or knowledge can transparently interact with the framework. Component bindings are externalized to support both internal system operations and external systems. Due to the isolation of component and data access logic, the bindings can be integrated with systems maintaining orthogonal data models. Model definitions are encapsulated in a static object model, allowing component binding to take place without impact to the consuming component or system.

These architectural goals are accomplished by integrating the implementing component and data models with the services offered by the core foundation services. The framework is designed to create an environment in which supporting services are abstracted from the component and data models to support greater integration flexibility. This abstraction allows the implementation to focus on the domain modeling rather than framework development; service delivery and integration support are accomplished through interaction with the core foundation services. The architecture is implemented by integrating domain-centric component and data models with the services provided by the core foundation services, resulting in the following view of an implementing architecture.

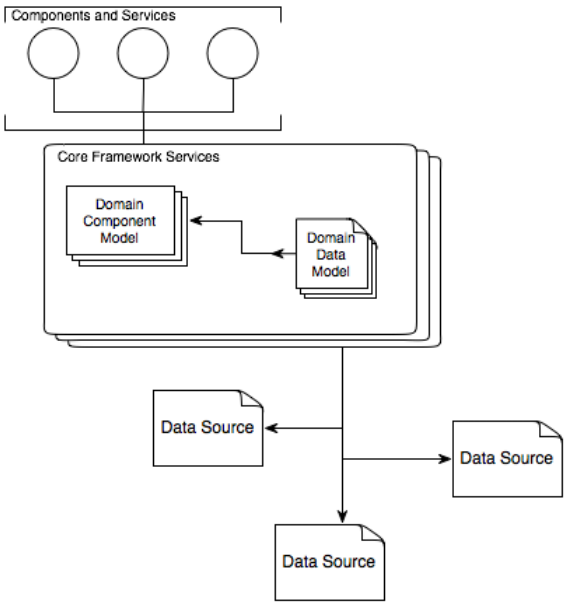

Figure 2: An Implementing architecture

In this view, the component and data models are wrapped by the core foundation services to create component implementations and facilitate access to external data repositories. The following issues are addressed by the Core Foundation Services.

1. Shared configuration. Components require configuration parameters at runtime to determine data source and service connection information, internal state, and runtime behavior. In many cases these configuration elements can be stored as property files or in databases. Traditional configuration mechanisms do not address the need for shared configuration between common components that are not deployed together. In a distributed component deployment, where multiple components may be abstractly linked through a common configuration, it is imperative that all components participating in a data workflow share configuration state. This can be accomplished in the framework by providing a common configuration factory using Java Management Extensions (JMX). Distributed components can be notified through Managed Bean bindings of parameter changes at runtime, preventing synchronization issues across components. The framework can also utilize shared configuration services in order to determine the availability of components to fulfill requests.

2. Middleware integration. The Java Enterprise Edition specification provides a number a number of middleware services for components to take advantage of at runtime. Middleware services can be used to provide common data source and service connectivity, framework monitoring, security, transaction management, and persistence [8,11]. While application servers provide entry points to service APIs, it is important that framework components consume these services in a uniform way to ensure common component operation. Shared configuration can also be thought of as a middleware service and may be utilized by components to determine how services are consumed. For this reason it is necessary for the framework to provide a component bridge for accessing shared functionality.
3. Runtime integration. In order to achieve a loosely couple component architecture it is necessary for abstract component bindings to be realized dynamically at runtime. A component runtime can be provided that will supply dynamic component bindings at the point of invocation. This allows for a greater level of distribution in the architecture. Runtime integration is also required to provide service bindings to shared configuration and middleware services, isolating component implementations from framework logic. The runtime is responsible for determining component availability and provides a bridge to clients communicating with framework components.

The core foundation services addresses each of these issues by providing managed services to framework components. The following is a discussion of each of these services and their impact on the component and data models.

CENTRALIZED MANAGEMENT FRAMEWORK

The centralized management framework provides shared configuration for framework components. The centralized management framework utilizes Java Management Extensions (JMX) to deliver configuration and management functionality to components and to the framework itself. Examples of management functionality can include the path to JNDI instances, external data source bindings, log verbosity, or runtime instrumentation. The centralized management framework is

composed of a Managed Bean (MBean) server wrapper and a set of objects conforming to a framework design pattern called the Manager / Accessor Pattern. The MBean server can be created as a standalone server or can utilize middleware MBean servers as discussed below. The centralized management framework wraps the MBean server by providing a deployment interface for the runtime creation of MBeans from properly annotated Java objects. This allows framework developers to express configuration elements as part of the data model rather than constraining elements to a JMX-centric view of the architecture. MBeans, known to the centralized management framework as services, are used as service objects because they are decoupled from the configuration architecture and can include crosscutting framework functionality.

The Java Management Extensions (JMX) specification defines an architecture for monitoring and configuring Java objects at runtime [7]. The JMX architecture provides several levels for deploying, managing, and connecting to instrumented resources. The centralized management framework utilizes the Instrumentation Level of the JMX architecture to publish services as managed resources. Resources are made manageable (i.e. services are created) through the JMX Agent Level by implementing MBeans, which are deployed to an MBean server. This Instrumentation Level pattern decouples the MBean from the JMX implementation and allows any system class to be easily refactored into an MBean. Decoupling is important in that it allows the inclusion of framework specific functionality like service persistence. The Instrumentation Level also provides a notification framework to facilitate the propagation of notification events between MBeans.

The JMX specification defines two classes of MBean, standard and dynamic. Standard MBeans are Java objects having an explicitly defined management interface. The management interface defines the methods made available for invoking operations on the bean or reading and writing its properties, and must conform to the Java Bean design pattern [13]. Standard MBeans expose managed attributes and operations statically through the management interface and cannot be modified to expose additional operations at runtime. The second type of MBean, the Dynamic MBean, exposes attributes and operations dynamically at runtime by implementing a method which returns all operation and attribute signatures. The JMX specification includes the*DynamicMBean* interface that a Java object can implement

to be considered a Dynamic MBean. The interface specifies abstract methods for obtaining attribute values or invoking operations. The following class diagram depicts a Dynamic MBean class.

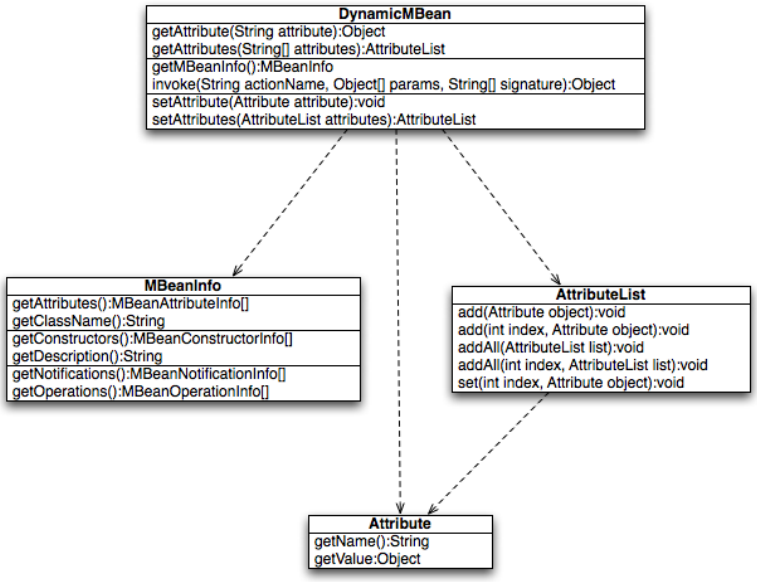

Figure 3: Dynamic MBean class structure

Classes implementing the *DynamicMBean* interface must implement the logic to determine MBean attributes, values, and operations at runtime based on parameter values. Attributes and their values can come from XML documents, databases or other classes and the sources for attributes or their values can be changed at runtime. Likewise, operation logic can be determined at runtime from multiple sources or can be configured to different behaviors under specific system states. Dynamic MBeans offer greater flexibility as management information can be updated, after the MBean has been deployed, in response to state changes within the framework.

The centralized management framework uses the JMX specification to create instrumented resources for management of components. These resources are deployed by the centralized management framework as services to be consumed by framework components. Because the centralized management framework provides its own service persistence logic and access control, services are realized through the implementation of Dynamic MBeans; the ability to create an abstract service infrastructure built using JMX was a key driver in this decision. Services are identified by the centralized management framework through the use of annotated classes following the Manager / Accessor design

pattern. The Manager / Accessor design pattern was created to allow some configuration elements to be made available to caller components while making others available to only the binding component. The pattern is so named as it specifies two access objects for the component configuration, the Manager and Accessor. The following diagram shows the class relationship that is required by the design pattern.

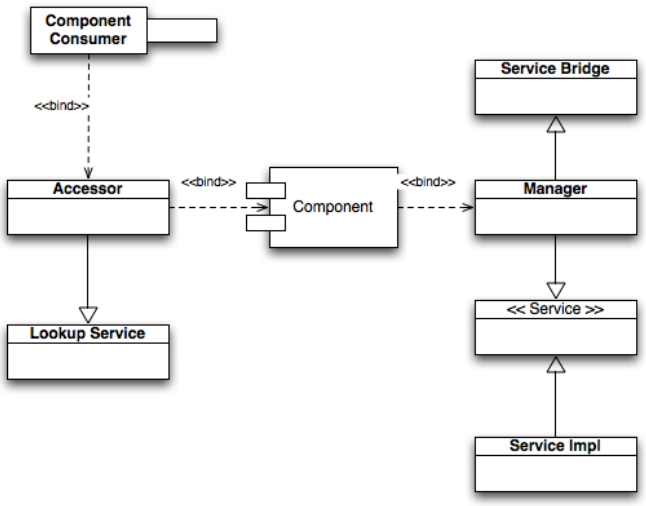

Figure 4: Manager / Assessor design pattern

In this design pattern, configuration is provided to a component through the Manager. The Manager allows access to properties and operations that are to be made available to a component. Generally these would include externalized properties and crosscutting operations. The Accessor in turn allows access to operations that a component caller would utilize in order to access the component and exposes no configuration properties.

The Manager and Accessor are supported by two centralized management framework delegates by extending an abstract base class providing delegate functionality. Each delegate provides the abstract functionality needed by the extending object to service component requests. In the case of the Accessor, the *LookupService* delivers service location functionality following the Service Locator design pattern [6]. The *LookupService* is used to locate components by class name from the appropriate JNDI instance based on connection information provided by centralized management framework Services. This delegate utilizes core integration services and integrated component runtime elements, discussed later, to provide component bindings and determine connection parameters. Each Accessor extends the *LookupService*,

creating a new delegate instance for each Accessor instance. The lookup logic is abstracted and encapsulated in the *LookupService* and each component Accessor must pass the fully qualified component interface class name to obtain a reference to a specific component. The *LookupService* maintains a reference to a JNDI context defined in a configuration service and injected by the integrated component runtime that is used to obtain remote component instances. This abstraction creates a single point of lookup within the framework, which supports many component accessors and creates a common integration point for binding to other CSF services. The Manager participates in a considerably more complex set of interactions than the Accessor in that it provides proxy access to services through Dynamic MBeans rather than relying on a single delegate implementation.

A centralized management framework service is essentially a registered Dynamic MBean wrapped by persistence logic provided by the centralized management framework and created from an annotated class known as a *ServiceImpl*. A *ServiceImpl* is a Java object annotated as a *Service* whose property reader and operation methods are annotated as a *ServicePoint.* The *Service* annotation requires that an object name be provided for use by the centralized management framework to identify the MBean that is created in the JXM Agent Level. The*ServicePoint* annotation allows for a default value to be specified in the case of a property access method. Each annotation also allows a meta-data description to be provided. Method parameters can optionally be annotated as *ServicePointParameters* in order to provide additional meta-data to the method by describing parameter function; the meta-data are encapsulated into an MBeanInfo object at the time the MBean is created. These annotations are used by the centralized management framework in the deployment of services to determine which operations and annotations to expose in the MBean.The following is an example of an annotated *ServiceImpl* class.

```
@Service(objectName="edu.unl.sample:type=SampleService",
providerInterface=ExampleService.class)
publicclassExampleServiceImpl implements ExampleService {

        private String myProperty;

        @ServicePoint(defaultValue="hello world")
        public String getMyProperty() {
                returnmyProperty;
        }

        publicvoidsetMyProperty(String myProperty) {
                this.myProperty = myProperty;
        }

        @ServicePoint()
        publicvoidtranslateProperty(@ServicePointParameter(name="Language",
                        description="The language to translate the property to")
                String language) {
                ...
        }
}
```

Figure 5: Example ServiceImpl class

In the example, the *ExampleServiceImpl* is a Java class which implements the *ExampleService* interface. This interface represents the management interface through which other components will access the service functionality. The class is annotated as a *Service* and both an object name and the service interface are specified. Despite implementing the *ExampleService* interface, the deployer will use the actual annotation to determine how the MBean will be created, so the interface must be specified there; this is known as the *providerInterface*. The use of interface inheritance is only a function of good OO-style and could be omitted entirely as the methods' invocations will be preformed dynamically at runtime with the MBean implementation. The example class provides three methods, two of expose the *myProperty* property. These follow the Java Bean convention for property access, and the getter is annotated to expose the property in the MBean. The setter could optionally be omitted and would result in *myProperty* being read-only in the MBean instance. The third method is also annotated as a *ServicePoint* and would map to an MBean operation. The parameters of this method are documented with *ServicePointParameter* annotations.

The centralized management framework contains a deployment framework responsible for the management of services through the creation of MBeans from *ServiceImpl* classes or instances. The *ServiceDeployer* is the centralized management framework object responsible for creating and managing the services. The *ServiceDeployer* maintains a reference to an MBean Server where the created MBeans will be deployed. At the startup of the framework the *ServiceDeployer* will either create an MBean Server

associated with its instance or bind to a container managed server. An environment property is set to determine how the MBean Server is obtained. Generally, if the framework is deployed in an existing application server, the container provided MBean Server is preferred; this is the default behavior. The *ServiceDeployer* creates a new service from a provided *ServiceImpl* by instantiating a new *ServiceBase* object with an instance of the *ServiceImp* and deploying it to the MBean Server.

A *ServiceBase* is a Java object implementing the *DynamicMBean* interface as required by the JMX implementation. The *ServiceBase* maintains internal lists of exposed getter, setter and operation methods by evaluating the annotations on the provided *ServiceImpl* class through reflection. Each access or invocation type is maintained in a separate list, one for getter methods, one for setter methods, and one for operations. Methods without *ServicePoint* annotations are omitted from the method lists. When the *ServiceDeployer* encounters an annotated getter method, it looks for a corresponding JavaBean setter method for that property. If one is found, that method is added to the setter list and the property is considered read/write. If not found, only the getter is added to the method list and invocations attempting to set the value of the property will fail, resulting in read only access to the property. The internal method lists are used by the *DynamicMBean* methods to provide attribute values and operation invocations and allows the single type of *ServiceBase* to provide functionality for any number of services, each differentiated by the underlying *ServiceImpl* instance. This structure enforces decoupling of the centralized management framework services from the JMX implementation in that the JXM implementation is only directly referenced by the *ServiceBase* and could be replaced at a later time by another management architecture with no change to the centralized management framework service implementations.

Service functionality is exposed to framework components through a Manager associated with the service through the provider interface. Much as the Accessor provides component lookup functionality, the Manager exposes service operation and attribute values statically by implementing the service provider interface. The actual MBean attributes and values that the *Manager* wraps are invoked using standard MBean access logic encapsulated in the abstract class *ServiceBridge*, which the *Manager* extends. When constructed, the *Manager* invokes its super class constructor, passing in the MBean

object name used when the *ServiceBase* was registered with the MBean Server. The *ServiceBridge* then binds to the MBean and maintains an abstract MBean object reference for *Manager* operations to be invoked against. In this way, a *Manager* class method implementations call super methods on the *ServiceBridge* to access the backing MBean to return values to the caller. Caller classes could bind to the MBean directly, removing the need for the *Manager*, however this would decrease type safety and expose the underlying JMX implementation to callers.

The implementation of the *Manager* has the potential to create a significant security flaw in the framework. In order for the *ServiceBridge* to invoke *Manager* operations it must have access to the underlying MBean server. MBean server invocations need not be limited to only those named MBeans that the framework manages; when using the container MBean server, the *ServiceBridge* has access to all MBeans, including those which manage the container itself. To address the potential security exploits that could be created by call interception, the pattern expressly requires that server-side classes only extend the *ServiceBridge*. This prevents client code from having direct access to the MBean server. A further guard is introduced by using the JMX security model to ensure that only authenticated callers are allowed to invoke MBean operations. Authentication functionality is injected into the *ServiceBridge* by the integrated component runtime runtime context discussed below. The following sequence diagram illustrates an attribute access operation through the manager.

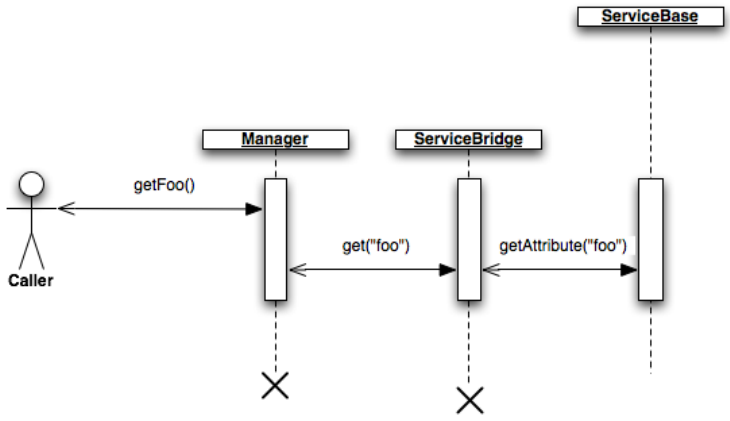

Figure 6: Property access sequence

By using JMX as the underlying configuration infrastructure, the centralized management framework can expose configuration attributes and operations to external management applications; most notably the middleware management console for the application server hosting the framework. This results in greater application deployment flexibility as the framework can easily integrate with existing deployment management solutions so long as they support JMX. This implementation allows the component configuration to be modified at any time in response to events like loss of connectivity or by a deployment manager for maintenance or redeployment. The design of the centralized management framework is such that is allows the framework to be easily adapted to changing deployment conditions with minimized loss of functionality of the framework components.

CORE INTEGRATION SERVICES

Where the centralized management framework encapsulates configuration logic and distributes it to collections of components, the core integration services are framework specific components and classes designed to support the implementation component model. Core integration services components and classes provide component model implementations with framework functionality that crosscut the domain model. The core integration services are designed to avoid redundant implementation and normalize framework operations. Core integration services components and classes support the component model implementation and integrate with other core integration services, data repositories, classes and centralized management framework functionality; it can be thought of as a base component model. The logic encapsulated by these components and classes are designed to cross cut domain specific implementations and are provided to facilitate greater reuse and standardization among component model implementations. The core integration services act to provide middleware functionality to the framework components, exposing functionality of the deployment containers to components in a framework specific way. This isolation of components from direct container interaction results in a more portable framework. As long as the core integration services are implemented to specification in a Java EE compliant container, framework components can be deployed into that environment.

The core integration services are a collection of components and classes, interacting with the framework in the same way as component model implementations. Foundation services are provided to component model implementations through the same service model used to integrate components. This allows the extension of the framework with new service functionality to take place without modification to existing component implementations; it achieves the black-box goal of the framework internally. Service implementations in the core integration services can take two forms: direct invocation or proxy. Direct invocation services are those that are statically bound to component implementations (i.e. the component class maintains a reference to an object instance providing the service). Proxy services wrap component invocations to extend the functionality of the component in a uniform way and can alter the flow control of the component invocation.

Direct invocation services are generally responsible for delivering integration points to middleware services, wrapping that functionality in a framework specific way. The core integration services include a direct invocation service called the *DataSourceInjector*. This service encapsulates the logic to perform a JNDI lookup on a JDBC data source created by the middleware container. The *DataSourceInjector* provides a uniform way to bind components to JDBC data sources. The static class is supported by a *DataSourceType* enumeration that contains a list of system data sources. The use of an enumeration is designed to make refactoring application code easier as a single update can change all references to a specific data source type. The data source types can be bound to JNDI data sources in the centralized management framework, allowing data sources to be dynamically swapped out at runtime. The *DataSourceInjector* can optionally use a string argument representing the JND name, bypassing the static enumeration.

A second direct invocation service commonly used across component implementations is the *Logger.* The *Logger* provides uniform logging services to components and classes and can integrate with a JDBC resource, external file, or container managed logging services to log statements are runtime. The *Logger* is bound to centralized management framework services to allow control of statement logging at runtime. Statements passed to the *Logger* are bound to the classes from which they were issued, allowing for better processing of log information. The *Logger* provides four loggers, partitioned around

the functionality encapsulated by the object logging statements. These loggers include a component, service, system, and application log to control how logging information is stored and allowing different framework functionality to be logged at different levels. Log levels are used to indicate the type of statement being logged and are used when setting logger verbosity. Statements are logged as either fatal, error, warn, info, or debug. These levels can be set separately for each logger, allowing, for example, all services to log fatal errors only, while components log debug information. The design goal of the *Logger* is to allow the system to be adjusted at runtime as needed to isolate faults or determine infrastructure needs. The following is an example of a component using the *DateSourceInjector* and *Logger* services.

```
publicclassSampleComponent {

        privateDataSourcesource                                                    =
DataSourceInjector.injectDataSource(DataSourceTypes.SYSTEM);

        privatestatic      Logger      LOG      =      Logger.getLogger(Loggers.COMPONENT_LOG,
SampleComponent.class);

        publicvoidclearUsageData() {
                Connection conn = null;
                try {
                        conn = source.getConnection();
                } catch ( Exception e ) {
                        LOG.error("could not obtain a connection", e);
                        LOG.debug("Total memory: "+Runtime.getRuntime().totalMemory());
                }

                ...
        }
}
```

Figure 7: Example use of DataSourceInjector and Logger services

In addition to middleware service wrappers, the core integration services provide several components that wrap external functionality in a framework-centric way. One such service included in the core integration services is a search engine that integrates with the data model implementation and acts as a direct invocation service. The *SearchService* provides full text fuzzy search logic by indexing objects implemented as part of the data model, building on object persistence logic provided by the Java Persistence Architecture (JPA). Objects that are to be indexed and queried through the *SearchService* must be JPA annotated Java objects. The *SearchService* requires that, in addition to JPA annotations, a set of search annotations be included as well. These annotations are used by the *SearchService* when

generating indices to identify searchable objects, properties exposed for search, search result description, search result identifier, search result name, and search result meta data. The latter five annotations are used by the *SearchService* to build search result objects that will be returned when a search operation is invoked. These search result objects are proxies for the actual Java objects, exposing only those annotated values to maintain smaller search result sets. The actual object can be loaded from each search result to provide all data encapsulated by the object. Objects are loaded and indexed by the *SearchService* using reflection to determine searchable properties and result values. The *SearchService* integrates with the centralized management framework to expose management operations for building a new search index.

Apache Lucene[2] provides actual search logic; the *SearchService* is a black-box wrapper for this functionality, which could be replaced without impact by another search provider. The main goal of the *SearchService* and its annotations is to wrap the functionality of the implementation search engine to abstract it away from caller components. This would allow for other search engine implementations to be substituted after implementation with no impact to the component or domain models. The operations and annotations provided by the *SearchService* are generic enough to facilitate search operations without need for implementation specific data to be incorporated into the domain model, decoupling the search implementation from the data model. This decoupling is further reinforced through the use of a generic search result object. The *SearchResult* object includes methods for generically obtaining search result information based on meta-data included in data model objects. The basic elements of a search result are the result title, text or description, and any additional named meta-data. Each *SearchResult* maintains the lookup parameters of the data model object mapping to the result, allowing for object to be loaded as needed. The primary driver for the encapsulated search data is to increase performance of result rendering by clients.

The core integration services also include a bridge to GIS services for accessing, manipulating, and publishing spatial data. The GIS service bridge is included to give framework components direct

[2] http://lucene.apache.org/java/docs/

access to spatial data. Unlike standard relational databases, which can be referenced through middleware services provided by the application container, spatial operations require either custom implementation or integration with external services. The spatial operation components included with the core integration services are backed by commercial GIS packages from ESRI and include ArcGIS Server, ArcObjects, and ArcSDE. The integration of these products as components in the framework is based off work in [12]. The premise behind the GIS services is that framework components may need to consume or produce spatial data and that production or consumption of the data should happen in a framework-centric way. Due to the complex nature of GIS service components, transparent integration with external services is far more practical than development of internal components to manage GIS operations. By utilizing an external service, GIS data produced by the framework and can be easily published and shared with other systems in a standard way. The abstraction also allows for the external GIS service to be replaced as needed with little impact on the framework components utilizing its functionality.

Framework components access GIS functionality through two components, the *MapQuery* and *MapArchive*. Several classes designed to implement an abstract spatial data model support these spatial components. Generally, the implementation of a component and data model as seen here would be left to framework implementations, however due to the cross cutting nature of GIS operations, the models are included in the core foundation services through the core integration services. The following class diagram shows the relationship between GIS components and data objects.

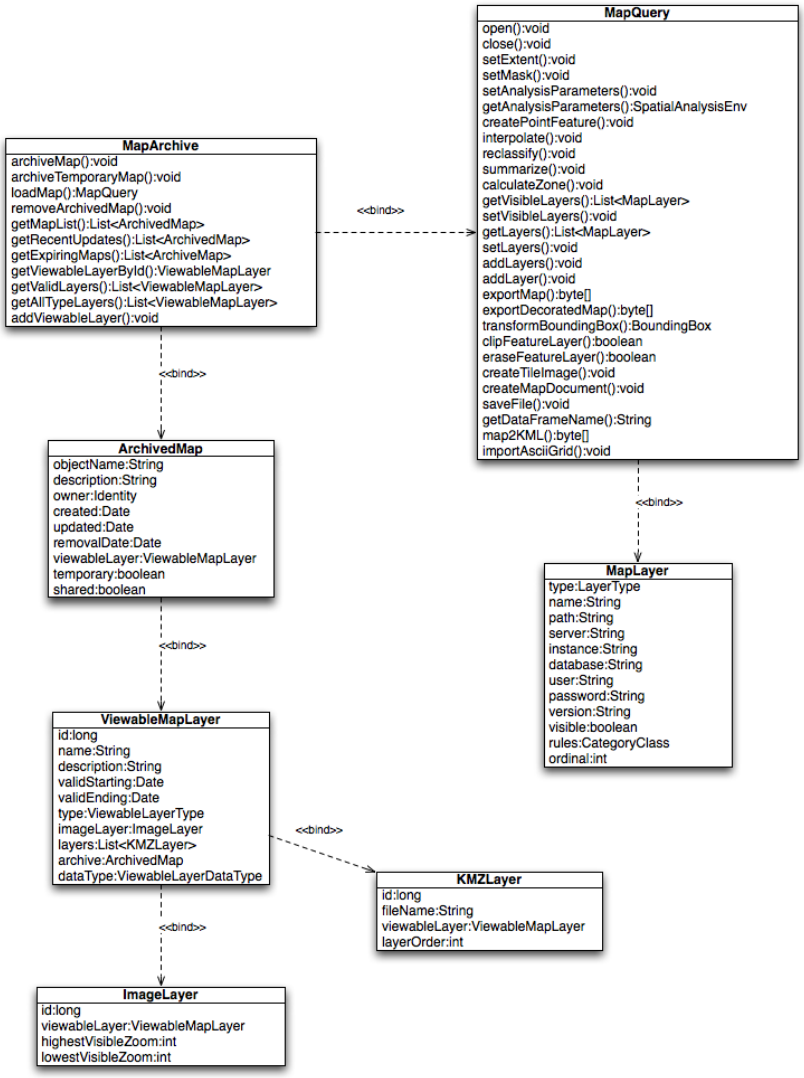

Figure 8: GIS component class relationships

The *MapQuery* is a stateful component that maps GIS operations to a corresponding GIS server object. Before any operations can be invoked on an instance of a *MapQuery* component, it must attach to the server object; this is done by calling *open()*. Once attached, the *MapQuery* can create, manipulate, export, or query data from the GIS server based on data encapsulated by the server object. The set of operations is designed to allow the framework to easily produce new or modify existing GIS server objects. Operations are specifically generalized to allow application across multiple domains, representing GIS authoring functionality as opposed to full GIS. For example, network analysis functionality is not present in the *MapQuery* as it is a more specialized kind of analysis.

Server-side GIS objects are made known to the framework through the *MapArchive.* The *MapArchive* provides JPA bindings, with framework specific attributes, to server-side GIS objects accessed through the *MapQuery*. The purpose of the *MapArchive* is to give framework components access to persistent GIS data in the framework backed by GIS server objects. References to server-side map layers are maintained in the framework by the *ArchivedMap* entity. The object model mapping allows the framework to access and manipulate persisted GIS data stored in the server repository through the *MapQuery*. While ArcGIS Server does provide visualization functionality, this is not incorporated into the framework due to its specificity with the ArcGIS Server implementation. Map visualizations are accomplished in the framework by exporting a view state associated with the server object as a collection of map layers on an *ArchivedMap* or as a static image. The *ArchivedMap*, weather or not it is being used for visualizations, links to a server side GIS object. This allows for static visualizations to incorporate aspects of analysis through the *MapQuery*. Map layers are generated in the framework to be displayed with Google Maps as either Keyhole Markup Language (KML) documents or as a collection of cached map tiles in Portable Network Graphics (PNG) format. The implementation of the framework GIS components can be seen to map the *MapQuery* to ArcGIS Server and the *MapArchive* directly to ArcSDE. Respectively, the components provide GIS authoring and storage functionality, and isolate the GIS implementation from consuming framework components.

The framework provides a user profile component as part of the core integration services. The *UserProfile* is a component containing a number of methods for creating, managing, accessing information about, and authenticating a user. The *UserProfile* is implemented as a component to allow binding with other components transparently. Information in the *UserProfile* is encapsulated in an entity called the *Identity*. The basic properties associated with the profile are persisted in the system database using JPA through the *Identity* object. The *Identity* object provides a link between the abstract concept of a user profile and other persisted objects like archived maps (discussed above) and folders (discussed below). Components interact with the *Identity* through the *UserProfile* by invoking operations that wrap the *Identity* data. A component obtains a reference to *UserProfile* through the standard framework lookup services. The initial reference is in an unauthenticated state, meaning that it is not associated with

any specific *Identity*. Any operations invoked while in this state result in an *InvalidStateExcception*. The only operation that can be invoked in this state is *authenticate()*, which takes as arguments a user id and password. If the credentials provided match an *Identity* in the system, the *UserProfile* is then bound to that *Identity* and its state represents that of the user. A single *UserProfile* object cannot be "re-authenticated" to represent another user, a new component reference must be obtained.

The *UserProfile* core integration services component by itself provides little useful functionality; its benefit to the system is realized when paired with two additional core integration services elements. The first of these, a folder management service, allows component model data to be bound to a specific user and persisted in the system database. A *Folder* is a persistent entity that is tied to the *Identity* and exposed to the system through the *UserProfile.* Using a *Folder*, the results of component operations can be stored in the system, allowing them to be quickly retrieved at a later time. This functionality can be used for archival of results or as a caching system for component invocations. The *Folder* is a standard Java object containing a name, a mapped *Identity*, and a list of *FolderEntry* objects. A *FolderEntry* object in turn maintains a description of its contents, a reference to the *Folder* to which it belongs and a reference to the binary contents of the entry stored on the file system. The *FolderEntry* has operations for reading and writing the referenced contents file. *FolderEntry* contents are managed through the *UserProfile* by the component that will persist the result.*Folder* access is provided to components through the *UserProfile*.

In addition to providing component result storage, the folder management service also provides operation scheduling to allow component invocations to be made at scheduled intervals and the results stored for later retrieval. The *FolderAutomator* is the persistent object responsible for invoking component operations and storing them in an associated *Folder.* Scheduling information is maintained by the *FolderAutomator* allowing the system to determine if the automation task should be invoked; invocations are performed by an instance of *Executor*. The *Executor* is an abstract class that provides Threaded execution logic and *FolderEntry* management. Operations are scheduled by creating a *FolderAutomator* and specifying a specific *Execution* class to generate the result that will be written to a *FolderEntry*. The *Executor* class will lookup components and invoke the necessary methods to generate a

result and return it as a byte array to the *FolderAutomator*. The *FolderAutomator* then creates a new *FolderEntry* for the result, which is then written to the file system. The system maintains a task that will query *FolderAutomator* objects form the system database based on scheduling information and execute the automation task as needed.

The second core integration services element that is paired to the *UserProfile* is an access control service. This service represents a core integration services proxy service, as its functionality wraps component invocations to determine if the invocation should take place. The access control service works by introducing the concept of a *Role* to the *Identity* object. Rules can be created for controlling which *Role* objects are allowed to invoke operations on specific components; these rules are encapsulated in *ACLRule* objects. The *ACLRule* is a persistent object that stores a regular expression representing the component name and operation that the rule governs, the role or user ID the rule is applied to, and the precedence ordering of the rule. The precedence is used to determine the order in which an *ACLRule* is applied to determine if an operation can be invoked. This is useful for allowing specific user overrides to a role restriction on an operation.

The *AccessController,* which manages *ACLRule*objects, is a component used by the system to create, query, and remove *ACLRule* objects from the system. The *ACLController* is not responsible for enforcement of an *ACLRule*, the object itself is used by the *UserProfile* to determine the Boolean state of the rule. Access control functionality is delivered by the *UserProfile*through the *allowCall()* method. This method takes a string representation of the component method signature and evaluates it against all of the rules applying to the specific *Identity* and roles associated with the *UserProfile*. A list of *ACLRule* objects is obtained from the *AccessController* for the unique user ID and any roles associated with the *UserProfile*. These objects are then evaluated against the method signature in order of precedence and the final Boolean result is returned. If the *ACLRule* does not apply to the method signature, it is skipped, preventing non-applicable rules from setting the call state to false.

INTEGRATION COMPONENT RUNTIME

Proxy services included in the core integration services give component developers a way to inject service functionality into components they create, however they are not always capable of delivering functionality at runtime. In the case of access control functionality, the abstract elements for controlling access to components is made available through the core integration services and rules for access can be described using the core management framework. But, these two foundation services cannot enforce component call checks at runtime without being explicitly invoked at the outset of each method call. This process would be complex and require significant crosscutting logic to be intertwined with each component implementation, defeating the purpose of both the centralized management framework and core integration services. To address the service/component runtime integration issue, the framework architecture provides an integrated component runtime.

The integrated component runtime maintains the foundation service context at runtime, allowing framework callers access to the runtime services provided by the core foundation services. Framework callers can be other framework components or external systems. The integrated component runtime is a component runtime in that component access cannot take place outside of the service context, which is initialized and maintained by the integrated component runtime. The deployment architecture of the framework is based on a client server model; each component or calling system becomes a client to any other component that it makes calls from. The “server”, or component instance being called, is identified to the caller, or client, by the centralized management framework through the service context. In this way, a unified framework can be transparently created in a distributed environment though a single service context. This single service context translates into a single runtime instance of the centralized management framework with a distributed component deployment. For this reason, the MBean server used to support the centralized management framework is generally instantiated and maintained by the integrated component runtime running in server mode. The following diagram illustrates this deployment architecture.

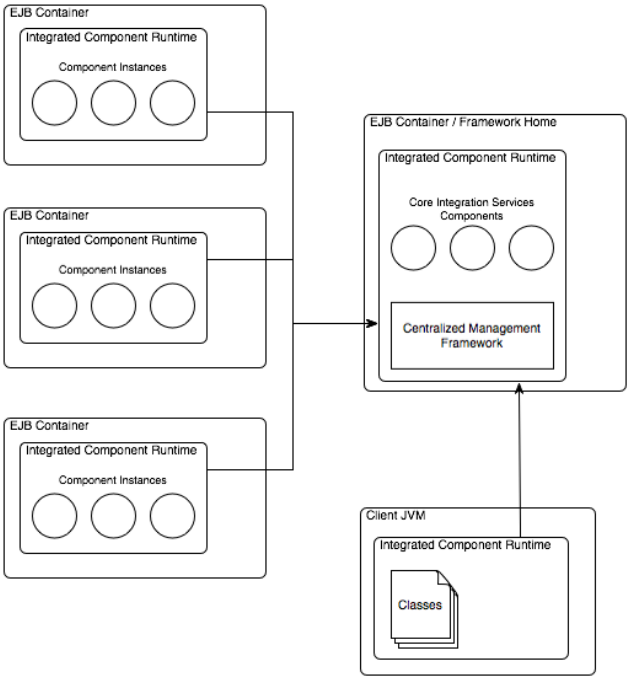

Figure 9: NCC deployment architecture

When bootstrapped, the integrated component runtime can run as either a client or a server. The singular difference between these modes is that, when in server mode, the integrated component runtime will instantiate two services. The first is the deployment service supporting the centralized management framework. The deployment service will scan a local file system for Java Archive (JAR) files containing configuration classes annotated with centralized management framework annotations. From these, MBeans will be created and deployed to the MBean server also created by the deployer. In server mode, the integrated component runtime will listen on a preconfigured port for management requests for the centralized management framework, the network URL to this port is known as the framework home. The second service is a native component container. Framework components are designed to rely on Java EE application server EJB containers for deployment; this, however, limits components to pure Java objects. In order to support legacy components, the system also provides a mechanism by which native code (compiled binary libraries) can be used to support components. Due to constraints placed on the EJB container, these components cannot be deployed with the pure Java component implementations. The integrated component runtime supports native component deployment by creating

a Remote Method Invocation (RMI) object registry into which native component instances, implemented as RMI components, can be bound.

The so-called Native Component Container (NCC) itself can be run in two modes, master or slave. This bimodal operation of the NCC is designed to facilitate load balancing by distribution of computational tasks preformed by the components. When the NCC is instantiated it broadcasts a UDP handshake request across the network. If no response is received, the NCC operates in master mode, utilizing its own deployed components to fulfill complete component operation requests. If a response is issued back to the instance that broadcast the handshake request, the NCC will execute in slave mode, utilizing its components to perform partial computations of component requests. The partial request is partitioned by the master NCC and sent to slaves as single component operation requests. Results are returned to the master NCC, the component caller, which will then merge results from all slaves into a single object and return the merged result to the original component caller. Native component callers receive the result transparently from a single component instance served by the master NCC but partitioned and delegated to the slave NCCs.

Apart from runtime service creation, the integrated component runtime functions no differently between client and server implementations and supports the centralized management framework and core integration services by providing execution contexts for runtime operations. The integrated component runtime maintains a management context to deliver attributes and operations to components by creating a binding between the centralized management framework *ServiceBridge* and the MBean service URL (framework home). The centralized management framework classes use this context to perform its MBean operations. A component runtime context is maintained by the integrated component runtime building on the component implementation model by creating a proxy based on the component interface. Components are added to the runtime context by the *LookupService*, which binds component references to the runtime context through a proxy. The integrated component runtime proxy class, called a *ComponentInvocationHandler,* contains an invocation method that wraps the component invocation and allows additional operations to take place before the method invocation or after the invocation before the return. Component callers interact with an instance of the *ComponentInvocationHandler*

rather than the component reference returned by the JNDI. The *ComponentInvocationHandler* instance is created with the component interface and the component reference returned from the JNDI; the component reference is known as the backing bean. The *LookupService* returns the *ComponentInvocationHandler* instance as an instance of the component interface providing transparency to the caller; the instance is known as the contextual component.

As methods are invoked on the contextual component, the *ComponentInvocationHandler* invoke() method is actually being called. This method takes three arguments, the instance of the object being called, the method being called and an array of method arguments; calling invoke and passing these arguments is handled by the Java proxy class, *java.lang.reflect.InvocationHandler*. The invoke() method contains the core integration services proxy service bindings that will interact with the component invocation. Depending on the result of the core integration services calls, the *invoke()* method will call the argument method on the backing bean held by the *ComponentInvocationHandler*. The result of the call is the standard component result. Additional core integration services methods will be called prior to returning the component result and may result in modification to the result data. The integrated component runtime can be extended to add component context logic, such as result caching, through provided annotations, with two proxy services explicitly defined by the integrated component runtime: access control and error logging. Error logging functionality is delivered by performing a checked call to the component and, in the event of an exception, logging it prior to the return. Component error logging is done on the client error console rather than the framework log system to ensure that fatal errors are reported even if the client cannot communicate with foundation services.

Access control logic is supported by the integrated component runtime bootstrapping process. This process ties the integrated component runtime instance to the framework through the *UserProfile*; this enables the access control functionality by identifying a runtime user (the developer) to the framework. In order to bootstrap the integrated component runtime a developer token must be present on the integrated component runtime (or framework) classpath. The developer token (firm.token) is an ASCII text file containing a UUEncoded encrypted string with the user ID for which the token is associated as well as the MD5 hash of the user password. At runtime, an integrated component runtime class will

load the token, decrypt the contents with the public key provided by the framework API, and authenticate the user, placing the profile in the runtime context. If the token cannot be found, or the user is not authenticated, the runtime bootstrap will fail and terminate the Java runtime with an error message. The integrated component runtime bootstrap process takes place at the first component lookup, not at the start of the parent process, preventing the need to include bootstrapping logic in framework client software. When a method call is made on a contextual component the *UserProfile* is loaded from the runtime context and an access control check is made as discussed above using the arguments to the *invoke()* method. If the result of the ACL check is false, the invocation will throw a security exception; if true, the invocation will continue as normal, performing any additional proxy service operations around the actual component invocation. The ACL check is the first action preformed by the *ComponentInvocationHandler* preventing unnecessary logic from executing in the event of an access control failure.

The integrated component runtime allows FIRM to deliver service functionality transparently thought standard Java APIs. Clients of the framework running in service context can interact with framework components as if the objects were local to the JVM in which the code is being executed. The communication protocols and component bindings are handled by the integrated component runtime, requiring minimal awareness of the remote nature of the components at runtime. This concept, protocol-unaware component binding was a key motivator in the design of the integrated component runtime and is one of the benefits of the core foundation services.

FRAMEWORK INSTANCE

Thus far the discussion of framework architecture has focused on the encapsulation of orthogonal component support logic through the development of foundation services. With an understanding of the core foundation services that the framework architecture provides to facilitate component interaction the remainder of this thesis will focus on the development process within the model presented. The primary focus of development will be on the creation of a domain data and component models. These two elements (Chapters 4 and 5) are required to form a complete framework

implementation and will be directly impacted by the core foundation services. Later (Chapter 6) a discussion of application implementation based on the framework will be provided, demonstrating how the core foundation services can support the use of component-oriented design in the development of end-user software systems. The implementation discussion will revolve around the instance of the framework architecture named The Framework for Integrated Risk Management[3] (FIRM). FIRM is a framework designed to support the development of decision support tools in the agricultural risk management domain.

The agricultural domain was selected as the domain for the framework development to allow comparison with the 3Co framework [4]. This domain is beneficial for several reasons. First, and foremost, it allows a comparison with 3Co and its implementing end-user application. Second, it provides a rich environment with many spatial and temporal data sets; both of which frequently change with the addition of new data. Lastly, the processes driving decision support in this domain are made up of many staged analysis. Data from one analysis method often serves as the input for another and therefore map well into the desired component structure supported by this framework architecture.

[3] At the time of writing several FIRM codebases exist. The version 1.0 (Thresher) code base was released in the fall of 2007 with several minor updates subsequently made. A branch of the Thresher code base was forked for a minor (1.1) release (Wildwoods) in the spring of 2008. The branch removed several unused elements of the framework. A third code base (2.0 also branched from Thresher) is scheduled for release (Mozart) in the fall of 2009 and further refines elements of the architecture, adjusting the communication model to better support the emerging Cloud Computing paradigm. As this work has been written over several years and started with the development of the Thresher codebase, the discussion will focus solely on models found in that code base, restricting the analysis discussion to the Thresher release. A light discussion of the rationale for the adjustments made to FIRM over its 2-year deployment lifecycle can be found in the Chapter 6. Detailed discussion of the later versions of the code base is to be provided by later works.

CHAPTER 4: FRAMEWORK DATA MODEL

The FIRM data model describes the many data sets that are accessed by the higher-level framework components. The purpose of this model is to allow normalized interaction between components and data sources as well as components with other components. The model is made up of two core elements, the persistent data model and the transient data model. The persistent data model represents a datum in a persistent state (i.e. in storage), made available to components through interaction with external systems in a specific way; an example of this is accessing data stored in a MySQL database using SQL queries. Data in the persistent data model are typically encapsulated by a single component. The transient data model represents data in a transient or runtime object encapsulated state, made available to other framework components and systems by component bindings.

One of the primary goals of the transient data model is to incorporate data access methodology in a naturally semantic way. The APIs that are provided by the transient data model should map to real-world data actions. When implemented properly, code utilizing the transient data model should read as if the process to read data were being described in natural language; or as close to this as possible with and Object Oriented language. The transient data model is realized by collections of abstract data types (ADT), which encapsulate the data they describe at runtime. In this view of the data model, data in the persistent data model serves as input to base-line framework components (stove pipes), and data in the transient data model serves to allow bindings to be formed between components and systems.

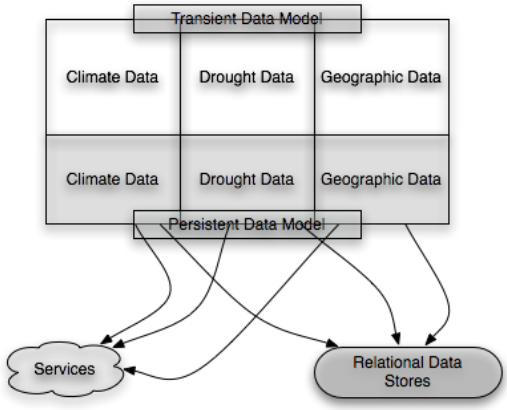

Figure 10: The FIRM data model

The diagram above illustrates the data model concept and shows the 3 main data types described by the model: climate, drought, geographic. The model is partitioned based on these data types in order to represent the unique properties of each. The persistent and transient aspects of the data types together constitute the complete domain data model upon which the component model will be built.

CLIMATE DATA MODEL

As a framework designed to support decision making in the agriculture domain, FIRM data objects are modeled primarily on climate data and their relationship to other aspects of the system. Climate data values form the basis of much of the logic encapsulated in the framework, from historical trend analysis to drought severity quantification, temperature and precipitation values serve as input to nearly every included model. Climatic data is time sequence data organized by spatial locality, the smallest recognized unit of which is referred to as a weather station. Weather stations can be generally thought of as collections of data for a fixed point in space; they most often map to a physical piece of hardware (the station) that collects data. FIRM is capable of housing data for stations belonging to multiple networks or for common IDs in a single network.

Persistent Data Model

FIRM represents weather stations and their data sequences with a relational data structure. This differs from the temporal structure utilized by the data source provider[4]. The data specification used by FIRM differs from that of the domain in order to facilitate both spatial and temporal operations against the data. The nature of weather stations is such that they may be "offline" for time intervals, reporting no data, and then come back online in a slightly different physical location. Because of this, the model separates out basic station information with its locale and network association. Additionally, FIRM can manage data for common stations across multiple data sources. A data source is a service from which data is collected. The following diagram illustrates the table relationship defined by the persistent climate data model.

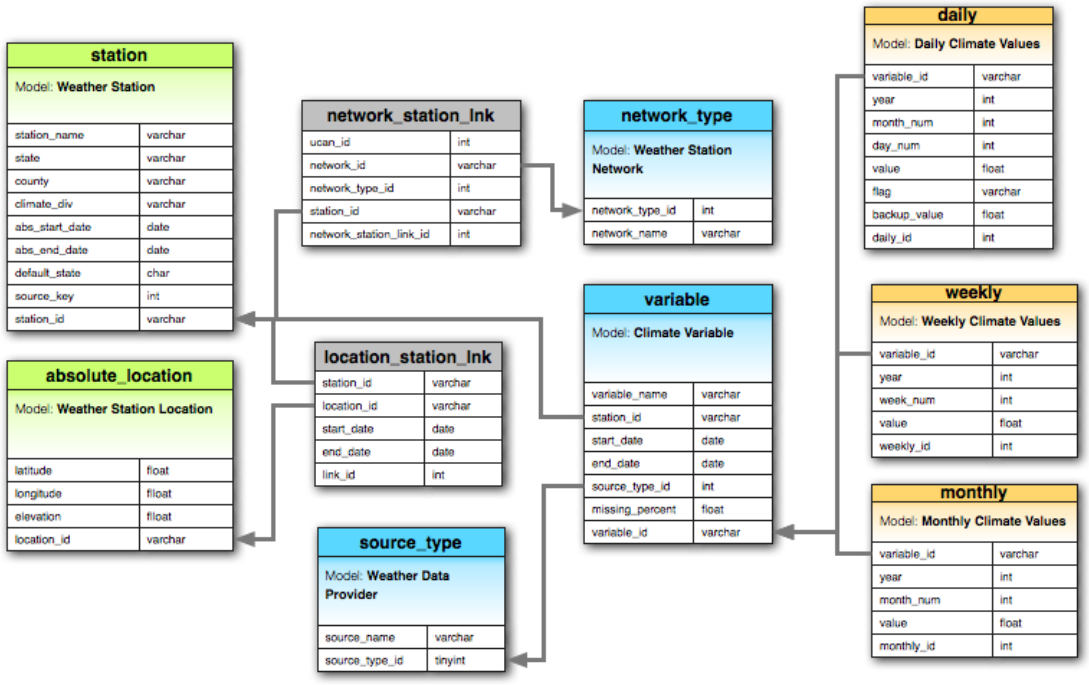

Figure 11. Persistent climate data model relationships

Each station reports data for a number of variables, which can differ across station networks and data sources. Variable linkage with a station is therefore both source and type dependent. Data are generally

[4] Climate data for FIRM are provided through a partnership with the High Plains Regional Climate Center, a part of the National Climatic Data Centers (NOAA NCDC) Applied Climate Information System (ACIS) network.

reported, and therefore available, at daily time intervals. FIRM summarizes data, where feasible, to weekly and monthly time periods for independent storage. The purpose of summarization is to increase performance of queries by limiting the calls to the daily data table.

The following tables describe the relational organization of data contained in the persistent climate date model. In addition to the climate variable values, the persistent climate data model defines information about the data collection. These data provide useful meta-information to associate with many of the higher-level domain specific processes and are encapsulated by the component model.

Field	**Description**
station_id (k)	The model generated primary key.
station_name	The name provided by the source / network for the station.
state	The state in which the station is located.
county	The county in which the station is located.
climate_div	The climate division in which the station is located.
abs_start_date	The earliest starting date for any variable reported by the station.
abs_end_date	The latest ending date for any variable reported by the station.
default state	Legacy value, deprecated when CORBA components were removed.
source_key	The primary key of the record describing the source of data for the station.

Table 1. Description of the station table

The *station* table provides an entry point for requests in the persistent climate date model. All data is organized, at the highest level, by station. The table contains the base metadata for stations, describing both the temporal window of data collected for the station, as well as information about the source from which the station is populated. Locale information for stations is split between the *station* table and the *absolute_location* table. Basic geopolitical metadata is stored with the primary entry in the *station* table; geospatial metadata is linked to the station through the *location_station_link* table. The need to partition geospatial and geopolitical metadata for stations arises from the fact that, due to the nature of weather stations, a station may be relocated over time with or without interruption of data reporting. In order to maintain a single record per station in the system, the change in reporting or location must map to the unique ID of the station. In this way a singular record of data, which may contain periods of no reporting, can be queried.

Field	**Description**
location_id (k)	The model generated primary key.
latitude	The latitude of the station in degrees.
longitude	The longitude of the station in degrees.
elevation	The elevation of the station in ft. above sea level.

Table 2. Description of the absolute_location table.

Field	**Description**
link_id (k)	The model generated primary key.
location_id	The ID of an entry in the location table.
station_id	The ID of an entry in the station table.
start_date	The date on which the station begins reporting data.
end_date	The date at which the station stops reporting, a value of 9999-99-99 indicates that the station data record is current.

Table 3. Description of the station_location_link table.

Data in the persistent climate data model are compiled from multiple sources that provide reported data for weather stations operated within a specific network[5]. Of the various networks that are represented by the persistent climate data model, there exists variance between stations reported and available data variables. Due to this variance, stations in the persistent climate data model must maintain metadata describing the network from which the station is queried as well as the source of any queried variable. Networks are differentiated from sources based on uniqueness of ID. Within a single network there exists no overlap between stations IDs; an ID maps to a single station. A source for a station may provide data from the same network as another source but apply quality assurance algorithms to the data before storage in order to generate a more serially complete record. Because of this, the persistent climate data model may contain two instances of the same network station, each form a differing source, reporting differing "completeness" of results. Because the source of the data within a network

[5] FIRM collects data from two primary networks, the National Weather Service Cooperative Weather Station Network (NWS COOP) and the Automated Weather Data Network (AWDN). The NWS COOP network is made up of observational stations and generally has a longer reporting window for station values. Stations in the AWDN report values automatically and are available daily.

differentiates the value of a data variable, the source becomes variable specific metadata, while the network to which the station belongs is station level metadata.

Field	**Description**
network_type_id (k)	The model generated primary key.
network_name	The name of the weather station network, examples include: AWDN, COOP, SNOWTEL.

Table 4. Description of the network_type table.

Field	**Description**
source_type_id (k)	The model generated primary key.
source_name	The name of the data source, examples include: ASIS, NOAA.

Table 5. Description of the source_type table.

Because a given station may exist in multiple networks, the metadata describing the network is linked to the *station* table in order to eliminate redundancy in the *station* table, providing a schema in third normal form (3NF).

Field	**Description**
network_station_link_id (k)	The model generated primary key.
network_id	The ID of an entry in the network table.
network_type_id	The ID of an antry in the network type table.
station_id	The ID of a weather station in the SDM.

Table 6. Description of the network_station_link table.

The purpose of the schemas discussed to this point is to provide a structure for organizing the data represented by the persistent climate data model. The remaining schemas link to this organizational structure and describe stored climatic variables and the time sequenced values for those variables.

Field	**Description**
variable_id (k)	The model generated primary key.
variable_name	The name of a variable that can be accessed from the SDM, values include: PRECIP, HIGH_TEMP, LOW_TEMP, AVG_TEMP .
station_id	The SDM generated ID of a station reporting the variable.
start_date	The first date on which a record of the variable exists.
end_date	The last date on which a record of the variable exists, a value of 9999-99-99 indicates a current record.
source_type_id	The source of the station from which the variable was queried .

Table 7. Description of the variable table.

The *variable* table is used to store information pertaining to each of the climatic data types that are represented in the persistent climate data model. The source specific name for the variable is paired with a station id and bound with a start and an end date. This indicates that a given station reports the variable under consideration for period from start date to end date. For each variable type and unique station ID, there is a single record in the *variable* table. If a station contains a gap in the data record for a specific variable, the absolute date range is used; that is to say the period of missing data will be represented by the system missing data flag. The persistent climate data model generated ID for the source of the station is also provided and allows for a mapping of source specific variables to stations. The *variable* table constitutes a mapping of raw data to source typed stations. The data values map to a specific variable as described by the following schemas.

Field	**Description**
daily_id (k)	The model generated primary key.
variable_id	The type of variable that this value represents.
year	The year in which the value was reported.
month_num	The 1 – 12 month in which the value was reported.
day_num	The 1 – 31 day on which the value was reported.
value	The numeric value.
flag	A flag indicating a synthetic value generated by a quality control algorithm.
backup_value	The original value if a quality control algorithm updated the value.

Table 8. Description of the daily table.

Field	**Description**
weekly_id (k)	The model generated primary key.
variable_id	The type of variable that this value represents.
year	The year in which the value was reported.
week_num	The 1 – 52 week to which daily values have been summarized.
value	The summarized value of a week of daily reported values.

Table 9. Description of the weekly table.

Field	**Description**
monthly_id (k)	The model generated primary key.

variable_id	The type of variable that this value represents.
year	The year in which the value was reported.
month_num	The 1 – 12 month to which daily values have been summarized.
value	The summarized value of a month of daily reported values.

Table 10.Description of the monthly table.

Variable values (data) exist as time series data in the persistent climate data model and are stored in 3 time units: daily, weekly and monthly. The primary format for storage of all variable values is the daily format. For the purposes of increasing performance of many higher-level processing functions, the persistent climate data model defines schemas for weekly and monthly summaries of all daily variable values. Such summaries are generated for each period (weekly or monthly) in a year and are based on the stored daily values; which is to say that the summaries are the results of summarization on quality controlled data. Each value is stored with a reference to the variable that it represents and the necessary temporal mapping field(s) (i.e. day, week, month, or year). Each value is assigned a persistent climate data model generated ID field for indexing purposes. In the case of daily values, two additional fields are provided for the quality control functionality. A data flag is provided to indicate if a value is synthetic, meaning that it was generated as a result of execution of the algorithm rather than being directly reported for the station. In the case of synthetic data, the original reported value is also provided.

Transient Data Model

The general structure of climatic data discussed above and the results they are used to produce are largely temporal in nature; therefore, the transient climate data model in FIRM is largely structured around the mapping of numerical values to time units. In order to effectively allow the generalized use of this data as both input to and output from models in the system, a generalized set of operations for traversing temporal sequences is paired with the mappings to form the basis of a majority of framework operations.

Access to the persistent climate data model is accomplished by components through SQL. Components must bind directly to a data source containing persistent climate data model schemas and build queries, extracting the data from SQL data types provided by the Java Database Connectivity (JDBC)

driver. This activity is supported by framework core foundation services; however it does represent a lower level access model than will be discussed below for the drought and geographic data models. For this reason, the transient climate data model does not maintain a 1:1 mapping with the schemas discussed above. The object specification provided by the transient climate data model describes data containers with access operations based on common data usage activities.

The resulting data object, the *CalendarDataCollection*, combines the time unit / value mappings with operations for traversing the dataset in a uniform way based on calendar arithmetic and structured around a common unit of data, the weather station. This concept is somewhat abstracted in the *CalendarDataCollection* with the data values encapsulated by the object being partitioned by some unique identifier. In most cases the unique identifier will map to a physical weather station, allowing computed data results to be easily related to a location and its static data. However, data sets can be created independent of the persistent climate data model, this generalization being desirable to support external climatic data sets.

The *CalendarDataCollection* is an immutable object and is used primarily to pass large quantities of temporal data between components in the system. An instance contains a data structure to map value collections to unique locations (weather stations), a starting and ending date for the time sequence, and a time unit. The time unit is represented by an enumeration in the transient climate data model called a *CalendarPeriod*. The enumerable types for this object are DAILY, WEEKLY, MONTHLY, and YEARLY. These are the only time units recognized by FIRM. The *CalendarDataCollection* implements *java.lang.Iterable,* by implementing this interface its contents are easily traversed in a time-wise linear fashion. The *CalendarDataCollection* wraps an internal data object of type *java.util.Map<String, Float[][]>*, extending the map to provide several domain-specific operations. One of the main drivers for the immutability of the *CalendarDataCollection* is the time unit mappings. Changes to the time unit definition at runtime could invalidate the state of the object by creating an inconsistency between the defined data array and the internal time unit. An instance of the object is created for a single set of time sequenced data. The data are sequenced based on one of the transient climate data model time units; this is used to inform the state of the *CalendarDataCollection* when it is instantiated, being used to compute the expected array

length for the internal data container. The lengths are statically defined in the framework to allow normalized parsing of the data and conform to the Gregorian calendar. The following example demonstrates how data can be read from a *CalendarDataCollection* instance and output to the console.

```
publicclassCalendarDataRead {

        publicstaticvoid main(String[] args) {
                try {
                        ...

                        CalendarDataCollectioncdc = ...

                        for ( String station : cdc ) {
                                System.out.printf("\n--- %s ---\n", station);
                                intcurrent_year = cdc.getBegin().getYear();
                                for ( float[] year : cdc.getStationData(station) ) {
                                        System.out.printf("%d: ", current_year++);
                                        for ( float value : year ) {
                                                System.out.printf("%f\t\t", value);
                                        }
                                        System.out.printf("\n");
                                }
                        }

                } catch ( Exception e ) {
                        e.printStackTrace(System.err);
                        System.err.println("Execution halted");
                }
        }
}
```

Figure 12: A CalendarDataCollection iteration example

For a single station, the above example would generate the following console output.

```
Container created for token: FIRM_BASE
[FIRM (codename Thresher) version 1.0.20080618.1-R]

--- 254739 ---
2000:  -9999.000000  -9999.000000  -9999.000000  -9999.000000  -9999.000000  -9999.000000  -
9999.000000 -9999.000000 -9999.000000 -9999.000000 0.530000 0.430000
2001:  0.520000  1.440000  0.990000  2.340000  9.600000  2.860000  1.220000  1.410000  5.760000
1.200000 1.680000 0.120000
2002:  0.390000  0.380000  1.070000  2.440000  4.870000  0.160000  1.130000  7.730000  1.900000
5.050000 0.140000 0.000000
2003:  0.250000  1.310000  0.710000  1.780000  3.440000  5.870000  1.220000  1.170000  3.540000
1.250000 2.180000 0.400000
2004:  0.960000  0.950000  2.780000  0.860000  3.010000  3.240000  2.660000  1.750000  2.910000
0.350000 2.510000 0.410000
2005:  0.700000  2.130000  0.560000  2.010000  2.030000  2.050000  4.420000  2.470000  0.260000
2.390000 1.710000 0.150000
```

Figure 13: Example FIRM output

The *CalendarDataCollection*provides methods for converting the data values as they are iterated over. Two common methods allow conversion between English and metric units: *convertToEnglish()* and *convertoMetric().* There are also methods that allow a classifier to be added that will convert values between domains. If an instance of the *DataClass* interface is provided, the *CalendarDataCollection* will return the result of the single interface method *DataClass.classifyValue()* on each call to obtain a data

value. This allows for type mappings to be provided at runtime to modify the values contained by the *CalendarDataCollection*. Common uses of this functionality could include classifications for spatial mappings. Rules like 'precipitation from 0.0 – 1.5 in. => 0' can be added so that data ranges can be represented uniformly across space (e.g. as a color range on a map). Different classifiers can be applied at runtime over multiple iterations, allowing the same data to be represented in multiple ways. The underlying data state of the object is not changed by applying a classification, preserving the immutability of the object.

The *CalendarDataCollection* includes several methods for data extraction beyond those provided by the iterator. The entire data matrix for any given station can be obtained by calling the *getDataMatrix()* method. This method wraps the *get()* method of the internal map, taking the key value as an argument and returning the mapped result. By using this method a caller may modify[6] the data contained in the *CalendarDataCollection* for use in other processes. The *CalendarDataCollection* also provides a method to extract a fixed data point from each station and augment it with metadata. This method is useful for generating spatial representations of data points, providing a convenient way to generate a *SiteList*. A *SiteList* contains a single data value for a point (station) and several pieces of metadata for a given location, with latitude and longitude being mandatory. This object can be passed to the spatial components of the framework to generate visualizations of data contained in the *CalendarDataCollection* for a fixed point in time over space. These various operations support the generalized nature of the *CalendarDataCollection* by allowing it to be easily integrated across the component model of the framework.

The*CalendarDataCollection* implements a protected constructor and can only be created legally in the framework by a *PeriodOrderedDataBuilder*. The *PeriodOrderedDataBuilder* object encapsulates the calendar operations for the creation of collections of values matching the temporal unit definitions implicit in the domain. These calendar rules allow data collections to be stored in normalized ways across

---

[6] As an immutable object, the *CalendarDataCollection* returns a copy of the internal data array on calls to getDataMatrix() so that modifications to the array do not change the state of the object itself.

the framework. The *PeriodOrderedDataBuilder* object is created when building a data set to be read by framework components. The object holds state for a single weather station at a time, allowing individual or collections of values to be added. When instantiated, the *PeriodOrderedDataBuilder* requires a temporal period over which the data will be stored as well as the temporal unit to be used. The object will then compute the expected state (i.e. the number of total expected values for each station). This computation is done based on the number of temporal units contained in the argument period. FIRM calendar data are organized by years and must contain a fixed number of values for each year. For this reason, regardless of the dates used in the temporal period, the *PeriodOrderedDataBuilder* will structure its internal storage objects to hold data values from the minimum period of the beginning year to the maximum period of the ending year. For example, if a *PeriodOrderedDataBuilder* were instantiated for daily data from November 22, 1978 to April 17, 2080, the storage range would be from January 1, 1978 to December 31, 2080. When created, the object will automatically fill data outside of the requested temporal range with flag values *OUT_OF_REQUESTED_RANGE* this allows the initial state of the object to represent the data request. The numeric representation of this flag (-9999) can be seen in figure 12 for months in the starting year which fall outside of the request range.

In addition to the internal storage objects, the *PeriodOrderedDataBuilder* also maintains a *DateTime* object whose state corresponds to the data at which the next value will be added. All FIRM data and component models utilize the Joda Time date API.[7] The issues with the Java *Date* and *Calendar* objects are well understood and an early design decision was to facilitate greater accuracy and functionality in the logic described here by using a 3rd party date API. In the previous example, after adding 5 values to the object, the internal *DateTime* value would be November 27, 1978. The minimum and maximum values of this state object are informed by the time unit argument used at the instantiation of the *PeriodOrderedDataBuilder;* the time unit is provided by an enumeration instance of *CalendarPeriod*. This state allows the *PeriodOrderedDataBuilde* to determine where in the data array a newly added value should be placed, moving to the next row in the array at the end of a calendar year.

[7] http://joda-time.sourceforge.net/

Because all values that are passed to the *PeriodOrderedDataBuilder* map to a date, a date must always map to the same value in an array for a fixed unit of time. This presents a potential problem in that all FIRM calendar logic conforms to the Gregorian calendar.

There are several nuances that occur in the Gregorian calendar. The most commonly recognized is that, in the Gregorian calendar, every 4th year adds an additional day to the year. These so called leap years place the extra day not at the end of the year but after the 58th day. As a result, the occurrence of March 1st in a leap year maps to the 60th day of the year and to the 59th day of a non-leap year. To preclude this date shift that would require that any logic traversing the *CalendarDataCollection* account for the leap year, FIRM treats all years as leap years containing 366 days. As data are added to the *PeriodOrderedDataBuilder* in a non-leap year, the 59th value added (representing February 29) for each non-leap year is adjusted to the 60th position, with the 59th automatically set with the flag value *NON_EXISTANT*. The flag value is added in place to the data sequence rather than at the end of the year to allow each date to always fall on the same day of the year.

A second problem is introduced when considering the Gregorian calendar in the context of a fix unit mapping. The Gregorian calendar view of a week holds that a week is a 7-day period beginning on Monday. This results in the first week of any year not always beginning on January 1. The first week of the year is defined in the Gregorian calendar as the first week in which a majority of the days of that week fall in the next year. A side effect of this system is that a given day of the year will not always fall in the same week of the year preventing the uniform mapping of a week number to a set of 7 dates. Because FIRM does not support conversion of values between time units, the inability to uniformly map a week to a set of dates does not present a problem. The sliding day upon which the first week of the year beings does, however, result in variability of the number of weeks occurring in a given year.

Every 400 Gregorian calendar years consist of a total of 20,871 weeks. This number is not evenly divisible by 52, rather it consists of 329 52-week years and 71 53-week years. These 71 so called leap weeks occur every 5 to 7 years when the last day of the 52nd week of the year falls on December 27th or earlier. When this occurs, the week after the 52nd still contains a majority of days in the current year; therefore this week is not the 1st of the next year but the 53rd of the current. In order to normalize the

number of weeks in a year for uniform data array length, FIRM treats every week as having 52 weeks.[8] While this decision does not impact the addition of weekly values to a *PeriodOrderedDataBuilder*[9] it does pose a problem when summarizing daily values to weeks in that a week of values is excluded 71 times out of 400.

The *PeriodOrderedDataBuilder* is supported by the *CalendarDataParser* to summarize sequences of one time unit to another. The *CalendarDataParser* partitions daily or weekly data into monthly or weekly blocks, allowing individual period values, summations or averages to be returned. This class can be used as an input filter to the *PeriodOrderedDataBuilder* to create various data summaries used by the component model of the framework. An instance is created with a period starting date, period ending date, and a 1-dimensional array of data values, one value for each time unit in the period. Like the *PeriodOrderedDataBuilder*, the *CalendarDataParser* maintains an internal *DateTime* to indicate the current calendar position of the data. The caller uses a set of methods to extract composite values from the initial data array. These methods map to the time units defined by the *CalendarPeriod* enumeration. For example, *getNextWeekAverage()* would return the average of the next calendar week from the current date position. For daily values, the *CalendarDataParser* would sum or average *k* entered values where *k* represents the number of days in the month. The *CalendarDataParser* will automatically adjust for flag values like *NON_EXISTANT* based on its *DateTime* state. It is in this object that the fixed weeks per year utilized by FIRM poses a problem.

To account for leap weeks, the *CalendarDataParser* will treat the 52nd week of a year containing a leap week as a 14-day week, combining the 52nd and 53rd weeks. The values for each unique day of the

---

[8] In retrospect as this thesis was being prepared this decision was determined to be a mistake. The implementation of FIRM should treat all years as having 53 weeks with the NON_EXISTANT flag being used in years which do not contain a leap week. This approach would have allowed for greater consistency in the model and would prevent the obfuscation of summarized data values that will be presented below. The main driver for the fixed 52 weeks per year was to address the expectations of researchers verifying data summaries provided by FIRM. The standard expectation in the agricultural community is that of a 52-week year. This expectation could have, however, been easily overcome with and explanation of the data model. Future versions of FIRM should be updated to introduce greater consistency within the domain data model.

[9] This statement assumes that the data values being added conform to the FIRM view of a fixed 52-week year (i.e. only 52 values will ever be added for a given year). If they do not, in those years that would otherwise contain a 53rd week, a data error will occur as the expected values state of the *CalendarDataCollecion* is computed with 52 weeks for each year. While this does not introduce any data errors if the assumption holds (as it does in the framework implementation), it is further support of the inaccuracy of this approach as described in footnote 7.

week are then averaged to create a composite 7-day 52nd week. This approach was taken to allow the inclusion of all data in a fixed 52-week year. Due to the general lack of variability for precipitation and temperature during this period, this approach was deemed to have the least impact on the generation of weekly time series data while allowing the data to conform to the fixed week year.[10] In addition to the concatenation of leap weeks, the *CalendarDataParser* also creates summaries based on the week number calculated from a date. For example, consider the date Tuesday August 19, 2003, which fell on the 231st day of the year in the 34th week of the year. A summary of weekly data including this day would include data from beginning on the previous Monday (day 230) and ending on day 236. Using this method to calculate weekly summaries for the entire year of 2003 would require 2 values provided from the previous year; this is because January 1, 2003 occurred on a Wednesday. Additional methods[11] were considered for weekly summaries and rejected as they did not allow for sequentially processing data based solely on a data stream and a *DateTime* object. The following is an example of building a *CalendarDataCollection* using the *CalendarDataParser* to generate weekly summations.

---

[10] This approach was not quantified statistically when originally made in the implementation of the data model. It is, however, known that the assumption of minimum variability does not hold for data from the Southern Hemisphere. FIRM contains data for only the Northern Hemisphere.

[11] The three methods considered where the 'Natural' approach (chosen), the 'Fixed Year' Approach, and the 'Sliding Scale Approach'. The 'Natural' Approach for summarization of weekly data will take a day, and provide a summation of the daily values from the first day of the week in which the value falls to the last (i.e. +7 days). Considering August 19, 2003, a weekly summary will be the sum of values from day 230 - 236. In the 'Fixed Year' Approach the calendar will be 'normalized' such that the first day of the first week of the year will fall on January 1, resulting in a given date always falling in the same 7-day period. Again considering August 19, 2003, in a 'fixed' year the date occurs in week 33. A weekly summary would include values from day 225 – 231. In 'Sliding Scale' Approach the concept of a week is shifted based on the selection data from a given dataset. In the August 19, 2003 example using the 'Sliding' Approach would result in the date being the first (or last) day of the period and taking the following (or previous) 7 values for the summary. A weekly summary would include the values from 231 – 237 (or 225 – 231). Each result yields a different summary set, and each method can be used to produce 'correct' results provided it is consistently applied across all calculations in the framework.

```
DateTime start = newDateTime(2000,1,1,0,0,0,0,GregorianChronology.getInstance());
DateTime today = newDateTime(System.currentTimeMillis());

// create a data builder to compute the number of days in the period.  NOTE:
// this number of days must be computed according to FIRM logic and not just
// the standard calendar.
YearDataBuilderdaily_builder   =   newYearDataBuilder(start,   today,   CalendarPeriod.DAILY,
                                                    DataType.UNKNOWN);
intlen = daily_builder.getExpectedValues();
System.out.printf("This builder expects %d total days of data\n", len);

// create a flat array of daily values
float[] daily_data = newfloat[len];
for ( inti=0; i<len; i++ ) {
        daily_data[i] = rand.nextFloat();
}

// create a data builder to hold weekly summarized data from the array above
YearDataBuilderweekly_builder   =   newYearDataBuilder(start,   today,   CalendarPeriod.WEEKLY,
                                                    DataType.UNKNOWN);
System.out.printf("This builder expects %d total weeks of data\n",
                                                    weekly_builder.getExpectedValues());

CalendarDataParser parser = newCalendarDataParser(daily_data, start.getYear(),
                                                    start.getMonthOfYear(),
                                                    start.getDayOfMonth());

weekly_builder.openStation("Example Station");
while ( parser.hasNextWeek() ) {
        weekly_builder.add(parser.nextWeekSum());
}
weekly_builder.writeStation();

returnweekly_builder.returnCollection();
```

Figure 14: Example data summation code

Working in concert, the *CalendarDataCollection*, *PeriodOrderedDataBuilder*, and the *CalendarDataParser* allow for data sequences to be mapped to daily, weekly, or monthly time units conforming to Gregorian calendar rules. The encapsulation of logic for manipulating data sequences based on a calendar system facilitates normalized component interaction and comparable data results. The rules encapsulated by these three objects represent the basis for nearly all calculations in FIRM and were heavily scrutinized during development.

The persistent climate data model describes both time sequence data and the location data around which the time sequences are group. In the persistent climate data model these location data are treated as metadata for locations (weather stations) and are paired with the time sequence data through the locations. The transient climate data model encapsulates these data through a generic object called the *MetaDataCollection<T>*. Like the *CalendarDataCollection*, the *MetaDataCollection<T>*maintains a mapping of data to stations by unique identifier. A key difference in this object can be reused to describe metadata organized around collection points. In order to enable this reuse, the *MetaDataCollection<T>* object is a generic. In the transient climate data model the *MetaDataCollection<T>*is typed to

MetaDataType, an enumeration of possible metadata types defined by the persistent climate data model. The *MetaDataCollection<T>*maintains data with two internal data structures to support its operations. The first is a mapping of metadata to unique identifier; this object is of type *java.util.HashMap<String, Map<T, Object>>*. The internal sub map, a collection of the metadata for a given unique ID, is itself a mapping of the data value to a typed enumeration, in this case a *MetaDataType*.

The *MetaDataCollection<T>* object implements *java.lang.Iterable<String>* to allow traversal in much the same way as the *CalendarDataCollection.* The object will iterate over the unique keys in its internal map, setting the object state to the current ID. The *getMetaData()* method will then return the metadata value of the argument type for the current station. The metadata map for an entire station can be extracted (while iterating or not) by calling *getStationMetaData()* and passing the unique ID of the station as an argument. The following is an example code snippet showing how to iterate over a*MetaDataCollection<MetaDataType>*.

```
publicstaticvoid main(String[] args) {
        try {
                ...
                MetaDataCollection<MetaDataType>metadata ...

                DateTimeFormatter formatter = DateTimeFormat.forPattern("MMM dd, yyyy");

                for ( String station : metadata ) {
                        System.out.printf("\n--- %s ---\n", station);
                        System.out.printf("Station Name: %s\n",
                                metadata.getMetaData(MetaDataType.STATION_NAME));

                        System.out.printf("Runs from: %s to %s\n",
                formatter.print((DateTime)metadata.getMetaData(MetaDataType.START_DATE)),
                formatter.print((DateTime)metadata.getMetaData(MetaDataType.END_DATE)));

                        System.out.printf("Located at: lat=%f lon=%f\n",
                                metadata.getMetaData(MetaDataType.LATITUDE),
                                metadata.getMetaData(MetaDataType.LONGITUDE));
                        }

        } catch ( Exception e ) {
                e.printStackTrace(System.err);
                System.err.println("Execution halted");
        }
}
```

Figure 15: Example metadata iteration code

The following output would be displayed from running the code in Figure 16.

```
Container created for token: FIRM_BASE
[FIRM (codename Thresher) version 1.0.20080618.1-R]

--- 375215 ---
Station Name: NEWPORT ROSE
Runs from: Nov 23, 1957 to Dec 31, 2003
Located at: lat=41.500000 lon=-71.349998

--- 379327 ---
Station Name: WOOD RIVER JUNCTION
Runs from: Apr 01, 1938 to Sep 30, 1948
Located at: lat=41.432999 lon=-71.699997
```

Figure 16: Example FIRM output

The *MetaDataCollection<T>* object also provides methods for building the collection; these are not separated out as they were for the *CalendarDataCollection*[12]. Like the *PeriodOrderedDataBuilder*, data is added to the collection by first creating (opening) a new station. Metadata is then added by calling the *add()* method; metadata are added a single *<T>*type at a time. Internal state is maintained by the object throughout the add process, a station is completed by calling *writeStation()*. Operations will throw an *InvalidStateException* if the state of the object is not correct for adding data. Data can optionally be imported as an entire collection for a given type with the *importMap()* method. This method will add the typed value for each station in the argument *Map<String, Object>* for an argument type *<T>.* Stations not contained by the collection at the time are added. Finally, a single metadata type can be extracted across all stations using the *extractType()* method.

The *MetaDataCollection<T>* object also provides operations to modify the contents of the collection on display. The order over which the stations are iterated can be modified by attaching a sorted list of station IDs with the *attachedSortedStationList()*. Calling this method updates the object state forcing all calls to *iterator()* to use the sorted list rather than the keyset. This functionality allows for runtime control of station display logic, providing lists sorted by state, or county in which the station is located. The transient climate data model provides an object for performing simple boolean operations on collections of metadata. The *MetaDataCollectionEditor<T>* provides basic and, not, or operations. The object maintains a typed reference to a *MetaDataCollection<T>,* which is updated by the operations

[12] This represents an inconsistency in the model that is scheduled to be addressed in a future update to the framework.

provided on the editor. This functionality is included as a convenience object to support search operations in application models.

The transient climate data model also defines several classes for describing climate variables. The *VariableMetaData* class is one such object and is used to encapsulate the specific metadata associated with a variable rather than a station. Variable metadata generally includes the type of data (defined globally in the framework by the *DataType* enumeration), and the dates for which values of the type are reported. Weather station reporting ranges for a given data type often contain gaps in reporting, these gaps are represented in the *VariableMetaData* as the percent of data missing over the data range described by the metadata. The second variable related class that is defined for the model is the *VariableFilter* class. This class is used by component callers to define parameters describing reporting ranges. The class has fields for a period over which values should be reported, the *DataType* of value that is reported and the percent of missing data that can be tolerated. The natural language description of data encapsulated by a *VariableFilter* could be 'all stations reporting precipitation from November 8, 1920 thru December 15, 1935 with at least 75% accuracy'.

DROUGHT DATA MODEL

The primary risk factor analyzed by FIRM components is drought. The climate data accessible through the framework are most often utilized by framework components to compute drought indices. A drought index is a model for computing some quantification of drought based on climate indicators [14,15,16,17,18]. These data themselves are generally structured as time sequenced data matching the time unit from which they were computed. Researchers, decision makers, and agricultural producers use these data to assess the severity of drought historically in a region, providing an augmented picture of the climate; these data represent information in the decision support system and are computed by components rather than archived in the system.

FIRM also provides access to anthropogenic drought data sets. These are structured data describing events that can be associated with drought. These so-called drought reports are used in conjunction with drought index data to provide a more complete view of the effects of climate on a

landscape and can serve as an invaluable resource to further quantify a drought index value or as an early indicator of drought.

Persistent Data Model

As information in the framework, FIRM does not represent drought index data in a persistent model, for this reason the persistent drought data model describes only drought index data and relationships. Drought index data are organized into individual units called reports. Reports take the form of user, media or impact. A user report describes an event or occurrence that can be attributed to drought submitted by a user of the system[13]. A media report contains the same basic information found in a user report but is sourced to a media outlet; generally these data come from regional newspapers. The third report, the impact report, represents a collection of media and user reports and represents a specific impact that can be attributed to drought. An impact is generally a single event or occurrence that may be described by several reports. The following diagram depicts the relationships between data elements in the model.

[13] User submitted drought impact reports (indeed all drought impact data) do not originate from applications built on FIRM. Although FIRM APIs do exist for the creation of drought report data, application models do (and should) not expose the functionality to users. All drought impact report data comes from the Drought Impact Reporter (http://droughtreporter.unl.edu), a drought data tool developed and provided by the National Drought Mitigation Center (http://drought.unl.edu). Through a collaborative partnership with the NDMC, FIRM is able to access the data for inclusion in framework components.

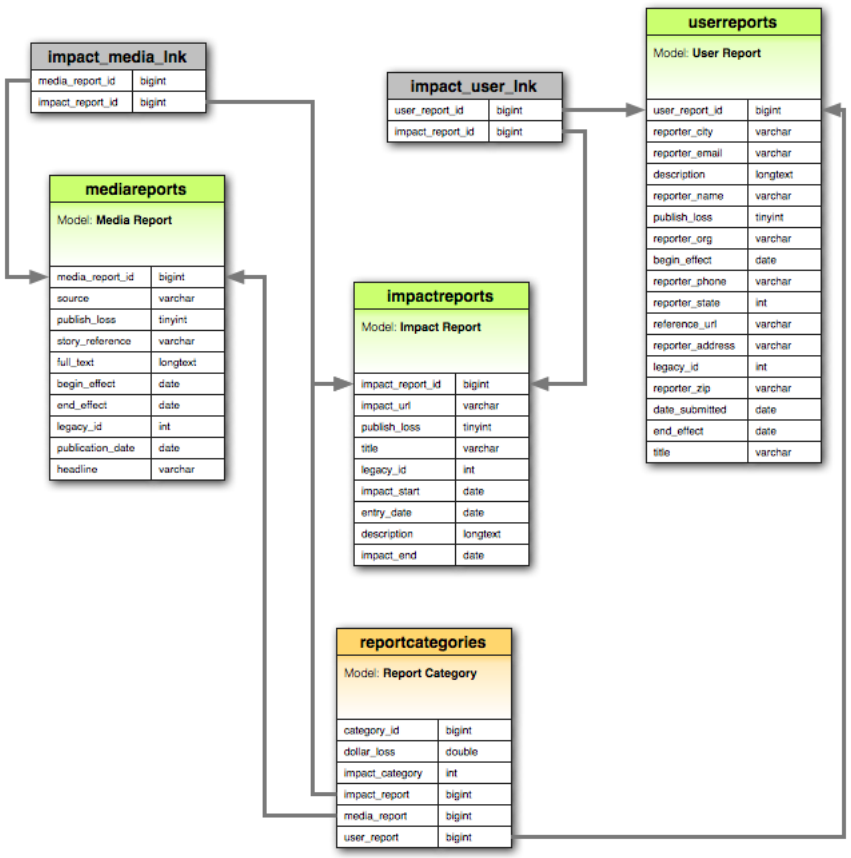

Figure 17. Persistent drought data model relationships

Each report type in the model can have multiple categories associated with it. Category types are defined by the data source and include: agriculture, water supply and quality, energy, tourism, business and industry, plant and wildlife, wildfire, disaster declaration, society and public health, relief restriction, general and other.

The following tables describe the table schema provided by the persistent drought data model. Tables for media, user, and impact all maintain common elements to indicate the beginning and ending of the event or occurrence associated with the drought. These metadata for the report are not mandatory with the beginning date matching the entry date and the ending date matching the beginning if none is provided.[14] As a result, all reports can be queried temporally. Each report type also maintains a flag to indicate that any loss, if present, can be published. Setting this flag to false indicates that any client

[14] This convention is enforced by the system from which the data originates and is not a function of FIRM.

displaying the data should not display the loss data. Loss data is associated with specific report categories and is a field of the category and not the report itself. The aggregate loss for a report is the total of all category losses.

Field	**Description**
media_report_id	The model generated primary key.
source	The news outlet from which the report was taken.
publish_loss	A flag indicating that a dollar loss associated with the report can be published.
full_text	The full text, summarized[15].
story_reference	The URL, if available, to the original story.
begin_effect	The date attributed to the beginning of the impact described by the report.
end_effect	The date attributed to the ending of the impact described by the report.
legacy_id	The ID of the report in the source database. Included for reference purposes.
publication_date	The date of publication of the original news article.
headline	The news story headline.

Table 11. A description of the media_reports table.

User reports include information beyond the standard report information found in the media and impact report definitions. Because a specific user of the system enters a user report it includes information about the user. This information is used by the originating system to identify individuals entering reports for verification purposes and is not published. As a component of the data model, FIRM incorporates this information to maintain data parity with the system from which the data is taken. The user data are not normalized in the model in order to prevent the need for registration when creating reports. This results in significant data redundancy. The general FIRM model would incorporate user data as part of the core foundation services *Identity* entity, allowing the data to be properly normalized across the models in 3NF.

Field	**Description**
user_report_id	The model generated primary key

[15] For copyright purposes a media report contains a summary of the actual article text. The original article text is only referenced through the story reference.

reporter_city	The reporter's city.
reporter_email	The reporter's email address.
description	A description of the event or occurrence.
reporter_name	The reporter's name.
publish_loss	A flag indicating that a dollar loss associated with the report can be published.
reporter_org	The reporter's organization.
begin_effect	The date attributed to the beginning of the impact described by this report.
reporter_phone	The reporter's phone number.
reporter_state	The reporter's state.
reference_url	A reference URL to more information provided by the reporter.
reporter_address	The reporter's street address.
legacy_id	The id of the report in the source database. Included for reference purposes.
reporter_zip	The reporter's zip code.
date_submitted	The date on which the report was submitted.
end_effect	The date attributed to the ending of the impact described by the report.
title	The report title.

Table 12. A description of the user_reports table.

Impact reports are designed to be collections of media and user reports. As a result of this relationship the impact report fields should be dynamically computed by aggregating all matching data from the contained reports. For example, the total loss for a specific impact is the sum of all category losses of all media and user reports under the impact. Dates should be likewise computed with the exception of the entry date. Human reviewers moderate all report data and impacts are entered in the system when a media or user report that does not describe any other impact is published. The entry date is therefore not an aggregate value. The moderator also enters the description and title when the impact is created.

Field	**Description**
impact_report_id	The model generated primary key
impact_url	The URL of a reference site for the impact.
publish_loss	A flag indicating that a dollar loss associated with the report can be published.
title	The title of the report.

legacy_id	The id of the report in the source database. Included for reference purposes.
impact_start	The date attributed to the beginning of the impact described by the report.
entry_date	The date on which the impact was entered.
description	A description of the impact.
impact_end	The date attributed to the ending of the impact described by the report.

Table 13. A description of the impact_reports table.

This structure for impact reports suggests that many of the fields seen above should appear in the transient view of the data as opposed to the persistent. That is not the case due to a bridge between two model versions of the impact reports. At the time of development (and continuing to the time of this writing) the originating system exists in a state of transition between the multiple report-typed-models shown here and a single report (impact) model. The data model that is used to incorporate this data in FRIM digests the data in the legacy format, separating off media and user reports. For this reason many of the aggregate values for impacts must appear in the persistent model and are set on initial data import. Standard link tables are used to related impacts with media and user reports.

Field	**Description**
media_report_id	The model generated primary key
impact_report_id	The reference ID of an impact report.

Table 14. A description of the media_impact_link table.

Field	**Description**
user_report_id	The model generated primary key
impact_report_id	The reference ID of an impact report.

Table 15. A description of the user_impact_link table

Report categories provide a categorization for a repot type and the dollar loss, where applicable, associated with the event described by the report. The table maintains the category as a numeric value that maps back to the available category types paired with a loss value. These data are paired with the ID of the parent report, mapping back to one of the report typed tables. Since the category names are not provided in the persistence model, it is possible that the category data could become inconsistent if incorrect values are provided for the category or if the category value definitions change. This issue is dealt with through the transient data model as discussed below.

Field	**Description**
category_report_id	The model generated primary key
dollar_loss	The dollar loss, if any, associated with the category.
impact_category	A numeric value that maps to one of the static category types for reports.
user_report	Reference ID to a user report.
media_report	Reference ID to a media report.
impact_report	Reference ID to an impact report.

Table 16. A description of the report_category table.

Transient Data Model

The transient drought data model is considerably less complex than the transient climate data model. Many of the entities defined by the persistent drought data model map 1:1 to entity objects in the transient model, those that do not are computed time sequence data and utilize the *CalendarDataCollection* for encapsulation. The 1:1 mapping of persistence to transient data mappings is facilitated by the Java Persistence Architecture (JPA). JPA is enabled by the core foundation services where the entity management functionality if configured and maintained. As a result, any implementing transient data model may be used in the definition of the persistent data model through JPA. In this view of the transient model entities are represented as Java Beans whose fields are annotated with data relationships. The JPA entity manager automatically synthesizes the persistence definitions (i.e. the SQL structure) and manages all persistence operations: create, read, update, and delete (CRUD). Since the class definitions are used in the generation of the persistent data model which was previously described and the objects are simply transient views of the underlying persistent data, the following transient drought data model entities map to the corresponding definitions in the persistent model: *UserReport*, *MediaReport*, *ImpactReport*, *ReportCategory*. Each of the three report objects is represented trannsiently by an entity of one of the following types: *UserReportBean, MediaReportBean,* and *ImpactReportBean.* Each of these three entities extend the *BaseReport* class. This extension is done in order to support CRUD operations in the components as discussed in Chapter 5. The purpose of this super class is to provide fields that are not present in the persist data model at runtime to components; namely, a boolean flag to indicate if the object is persisted (tied to a persistence context).The transient drought data model defines

several entities, which are not persistent. These entities are used to support computation of drought index values and CRUD operations on drought reports.

In order to prevent data integrity issues in the impact category field of the report category table, the transient drought data model introduces an enumeration of all possible category elements. The enumeration, *ImpactCategory,* incorporates the ordinal of each enumeration instance, and serves as the category value in the persistent state. The later discussed component model then utilizes the enumeration to ensure data integrity in the persisted data. The drought report entities are complemented by two transient entities designed to support query operations. The primary object is the *ImpactQueryResult*. Like the *CalendarDataCollection* defined in the transient climate data model, the *ImpactQueryResult* encapsulates a set of methods for working with a collection of *ImpactReport* objects.

The *ImpactReport* object represents the entry point into drought report data. Since all *MediaReport* and *UserReport* objects belong to an *ImpactReport*, higher-order system functions access the objects through the *ImpactReport* as opposed to directly. The *ImpactQueryResult* allows collections of *ImpactReport* objects to be grouped based on common query operations. The component model defines specific operations. The *ImpactQueryResult* provides common traversal operations on the object collection and is therefore *java.lang.Iterable*. The object maintains two internal collections of type *ShadowImpact* and *ReportPlot*. These two collections support all of the object operations.

The *ShadowImpact* is a 'light weight' view of the *ImpactReport*. Since the *ImpactReport* is a JPA entity, an instance may contain a large object graph of related media or user reports, spatial references, and categories, the *ShadowImpact* is provided as a 'display object'. A *ShadowImpact* encapsulates the basic fields of an *ImpactReport*, a collection of *ReportCategory* objects, and the unique ID of the *ImpactReport* it represents. All query operations populate these *ShadowImpact* objects by calling directly to the persistent model instance (i.e. using SQL). If the full object graph is required for display or edit, the entity is loaded using JPA CRUD operations. This object allows the data to be easily displayed as a query result while maintaining the ability to load the complete object.

The *ReportPlot* represents a summarized spatial view of the *ImpactReport* collection. The object maps *SpatialReference* objects to *ShadowImpact* collections, subdividing the collection maintained by the

ImpactQueryResult based on some geo-political boundary entity.[16] The *ReportPlot* allows *ShadowImpact* objects to be added to a geographic unit in much the same way as the *PeriodOrderedDataBuilder* allows time series data to be added to a weather station. Object state is represented as a current *SpatialReference* and as *ShadowImpact* objects are added statistics for the *SpatialReference* are updated. An example usage would be to define a *SpatialReference* for the state of Nebraska (the plot) and add *ShadowImpact* objects for Nebraska to the plot. As these objects are added category counts, total dollar losses, and effect boundary dates are updated. The result is that collective report data can be aggregated to a specific spatial extent.

The *ReportPlot* defines methods for iterating over the *ShadowImpact* collection based on the spatial partitions. The partitions can be iterated over, or based on a specific category, allowing the *ReportPlot* to partition the *ImpactReport* view by category or spatial reference. The *ReportPlot* also defines methods for accessing statistical data for each plot. This object is used by the *ImpactQueryResult* to control the iteration of the *ShadowImpact*. The *ReportPlot* implements a protected constructor and can only be created while building an *ImpactQueryResult* object. The organization of *ShadowImpact* objects into plots is the standard interaction model for the *ImpactQueryResult.* The unorganized collection of *ShadowImpact*objects can be obtained from the *impactList* field. The following is an example use of the *ImpactQueryResult* object.

[16] *SpatialReference* objects and their function are discussed in the next section.

```
Publicstaticvoidmain(String[] args) {
        try {
                ...
                ImpactQueryResult result = query.searchAllReports("Prayer");

                for ( ReportPlot current_plot : result ) {

                        System.out.printf("\n --- Plot Report Data ---\n");
                        System.out.printf("Plot Name: %s\n", current_plot.getPlotName());
                        System.out.printf("\tReport Categories:\n");

                        for ( ImpactCategory cat : current_plot.categories() ) {
                                System.out.printf("\t\tCategory: %s\n",
                                                cat.getPrintName());
                                System.out.printf("\t\tPercent in plot: %f\n",
                                        current_plot.categoryPercent(cat));
                        }

                        System.out.printf("\tReports: \n");
                        current_plot.iterate();
                        while ( current_plot.next() ) {
                                System.out.printf("\t\tReport Title %s\n",
                                        current_plot.getImpactTitle());
                        }
                }

        } catch ( Exception e ) {
                e.printStackTrace(System.err);
                System.err.println("Execution halted");
        }
```

Figure 18: Example drought report iteration code

The following output is produced by the above code.

```
Container created for token: FIRM_BASE
[FIRM (codename Thresher) version 1.0.20080618.1-R]

--- Plot Report Data ---
Plot Name: Search Result Plot
        Report Categories:
                Category: Society and Public Health
                Percent in plot: 0.920000
                Category: Other or Unclassified
                Percent in plot: 0.080000
        Reports:
                Report Title A pastor in Cullman invites residents and leaders ...
                Report Title A prayer service was held at Mount Olive AME Zion ...
                Report Title Alabama's Governor Bob Riley has declared that Jun...
                Report Title An ancient Jewish prayer ritual will be held at th...
```

Figure 19: Example FIRM output

Data that are output from the drought index components, discussed below, generally utilize the *CalendarDataCollection*. The exception to this is the Newhall Simulation Model, a drought index used to quantify drought based on soil taxonomies. The organization of the data generated by the Newhall, while time sequenced, are not single data values but several collections of values. In order to accommodate these data values the transient drought data model defines the *NSMSummaryDTO* and *NSMCompleteDTO*. Each of these objects conform to the Java Bean pattern and encapsulate several collections of data values and soil taxonomy definitions for each type of Newhall operation: complete and

summary. The data values contained in each object are numeric (float) values organized by time unit, soil taxonomy names (string), and enumerated values belonging to the *SoilMoistureRegime*, *SoilMoistureRegimeSubDivision*, or *SoilTemperatureRegime* enumerations. This collection of class definitions is specific to data from the Newhall computations and does not serve any reuse value within the framework. These definitions are provided for a single object result type for operations of the Newhall component.

GEOGRAPHIC DATA MODEL

Underlying all of the data in FIRM is a spatial property. In the climate data model the temporal data are organized by weather station, allowing the data to be modeled spatially. Drought data are also linked spatially whether through weather station for which they are computed or for regions in which they are reported. In order to accommodate this and to allow data to be related to the ecosystems that they describe, FIRM partitions spatial data, modeling it as a first-class element. Ecosystem data as well as domain-generalized spatial data are modeled collectively as geographic data. These data include geo-political units (like cities, counties, or states), land usage, soils, and agricultural statistics. In general, these data crosscut the domain specific decision support data and are used by the framework to transition data to information or information to knowledge.

The spatial aspects of the geographic data model ate strongly coupled with the spatial components provided by the core foundation services. These data were originally intended to be included directly in the service model for the system. Many of the spatial data types, while not specific to the decision domain being considered, are structured around domain concepts. As a result, it becomes difficult to reuse the spatial data model in its entirety in unrelated domains. For this reason, the framework includes the spatial data model as part of the domain rather than service data model, tolerating a certain amount of data reuse between domain implementations of the framework.

Persistent Data Model

The persistent geographic data model differs from the previously discussed persistence models in that the data it describes are realized both relationally and spatially. As a result of this, the

implementation of the model exists both as a relational database (SQL) and a geospatial information system (GIS) storage engine (SE). In the case of the geospatial storage engine, the data are partially maintained in a relational structure; however that structure alone is insufficient for reasoning about the data. The following diagram illustrates the relational persistence model for the geographic data model in FIRM.

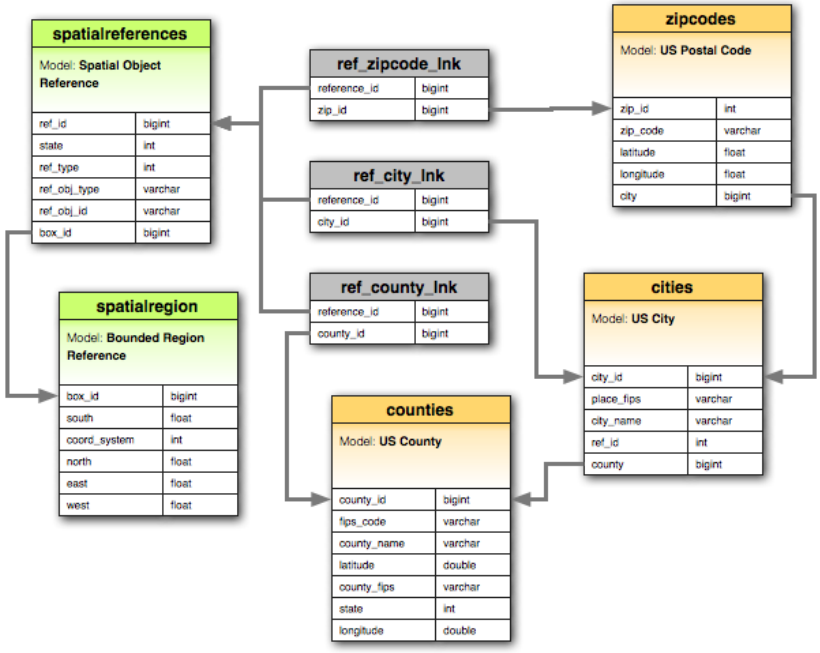

Figure 20. Persistent geographic data model relationships

This relational model is designed to facilitate spatial views of other model data in FIRM by linking non-spatial elements with spatial structures. This model can be thought of as a bridge between a non-spatial relational model and a fully spatial GIS model. These tables allow data that are not accessible to the GIS system to be defined in such a way that they can be augmented with purely spatial data and manipulated by GIS components. The following tables describe the schema and its use in creating a spatial view of data.

The persistent geographic data model is defined to incorporate elements of spatial locality and as a result contains definitions for several common geo-political units common to FIRM data. The tables below show the field definitions used to describe US cities, zip code areas and counties. US states are not

included as part of the persistent data model but are represented by numeric values. This design decision mirrors that for drought report categories with the same potential consequences. The political boundaries included in the model group boundary identification data with spatial definitions. US cities are a notable exception as they do not currently include position data.[17] In general, the geo-political boundaries define regions in space rather than points; all geo-location data for a region are based on computing the center point (the centroid) for the region. More exact boundary definitions are delegated to GIS stores and are linked by core foundation service spatial components.

Field	**Description**
county_id	The model generated primary key.
fips_code	The full US postal place code for the county, this is a combination of the state and county FIPS.
county_name	The name of the county.
latitude	The latitude of the county centroid.
longitude	The longitude of the county centroid.
county_fips	The US postal place code of the county alone.
state	A numeric value that maps to one of the static US states.

Table 17. A description of the counties table.

Field	**Description**
city_id	The model generated primary key
place_fips	The US postal place code for the city.
city_name	The name of the city.
county	The ID of the county in which the city is located.
ref_id	The reference data set ID for mapping cities back to the GIS data set from which they are extracted.

Table 18. A description of the cities table.

Field	**Description**
zip_id	The model generated primary key
zip_code	The US postal code.
latitude	The latitude of the zip code centroid.

[17] This oversight in the model is presently dealt with at the application layer through geo-coding and was identified as a future update to the model.

longitude	The longitude of the zip code centroid.
city	The city in which this zip code is defined.

Table 19. A description of the zip_code table.

Non-spatial data are spatially related to the geo-political boundaries discussed above through spatial references. A spatial reference links any other data element in the FIRM data model to a zip code, city, county, state, or other geographically bounded region. The table definition includes fields for the reference table as well as the reference ID. These fields allow the definition to facilitate spatial referencing even in the event of new entity definitions in the persistent model. A side effect of this generalization is that invalid reference data can be provided, leading to orphans in the instantiation of the data model. Checking against this is left to the component model.

Field	**Description**
ref_id	The model generated primary key
state	A value mapping to the static list of US states. Set if the type is a state reference.
ref_type	Indicates the type of reference is described by the data, values can be zip code, city, county, state, region.
ref_object_type	The table name containing the referencing object.
ref_object_id	The id of the referencing object.
box_id	An regional id matching the spatialregion schema.

Table 20. A description of the spatial_reference table.

In the definition of a spatial reference, states and spatial regions are explicitly defined as fields. As was previously discussed, states do not have a definition in the persistent model and therefore are referenced only by the numeric value associated with the state. Spatial regions can be defined for arbitrary boundaries defined by the persistent model; they are explicitly defined as they reside in the same model. Spatial references encapsulate the coordinates of the north/east and south/west corners of a bounding box, allowing regions in space to be defined by components and associated with other entities in the model. The region may be defined for any coordinate system supported by FIRM.

Field	**Description**
box_id	The model generated primary key
north	The maximum latitudinal bound of the box.
east	The maximum longitudinal bound of the box.

ssuth	The minimum latitudinal bound of the box.
west	The minimum longitudinal bound of the box.
coord_system	The coordinate system used for the bounding values.

Table 21. A description of the spatial_regions table.

Standard link definitions are included in the definition to link reference data to the appropriate geo-political entities.

Field	**Description**
reference_id	The model generated primary key
zip_id	ID of the reference zip code.

Table 22. A description of the reference_zip_code_link table.

Field	**Description**
ref_id	The model generated primary key
city_id	ID of the reference city.

Table 23. A description of the reference_city_link table.

Field	**Description**
ref_id	The model generated primary key
county_id	ID of the reference county.

Table 24. A description of the reference_county_link table.

Transient Data Model

Because many of the spatial operations provided by FIRM are delivered through the core foundation services, the transient geographic data model consists of few definitions. As was the case with the transient drought data model, the objects described by the transient geographic data model provide JPA mappings to the persistence model. As a result, the model definitions from the persistence model map directly to the object definitions of the transient model. A full description of each object and its use would be redundant. The transient geographic data model defines the following objects: *SpatialReference, BoundingBox, USZipCode, USCity, USCounty, USState, SpatialReferenceType*. The *SpatialReferenceType* and *USState* objects are enumerations of all possible values for which they describe. They are used to ensure data consistency in the persistent model ensuring correct value by mapping

enumeration ordinal values to the type fields of the *SpatialReference* definition. The *BoundingBox* object represents an instance of the spatial region definition defined above.

The transient geographic data model does provide one class definition that does not map to the persistent model. The *SpatialFrequency* class defines fields and methods to tie spatial classifications to periods of time, modeling change over time. Map data contained in a spatial engine, like Arc GIS Server, may have classified regions indicating some event, attribute or occurrence. These classifications would appear on a cartographic representation of the data as colored or shaded regions. An example could include coloring Lancaster county Nebraska based on the political party winning a presidential election. That classification may change over time; the *SpatialFrequency* class allows that change to be captured by showing frequency of a given classification (i.e. votes Republican) over a time period. The object maintains a starting and ending date describing some period, a reference to the underlying map instance provided by the core foundation services, and a *java.util.Map<String, Float>* of percentage of region for a given classification. This object can then be used by the component model to tie frequencies to geographic bounded regions of any variety.

CONNECTOR ARCHITECTURE

The FIRM domain data model can be thought of as a connector architecture. The normalization of data types identified through an analysis of domain models encapsulated by the components facilitates uniform component I/O. This results in limited data translation on the part of the components, allowing each to easily digest input to produce output. The result is that a component can be connected directly to any other component in the system by virtue of some higher-level process defined by a sequence of component calls. The implicit 'connections' that are formed as a result of the domain data model become the drivers of the application model. This is the primary role of a well-defined domain data model, and is the central requirement in order for the component model to facilitate data integration.

With an understanding of the data elements common to this domain one can begin to encapsulate operations on those data in components. The following chapter discusses the FIRM component model and how it takes advantage of the domain data model to integrate the disparate data

sets it describes. The inclusion of a component model completes the FIRM architecture and allows for the development of an application model and higher-level software products.

CHAPTER 5: FRAMEWORK COMPONENT MODEL

The FIRM component model builds on the domain data model to encapsulate logic and operations for domain specific processes and methodologies. The component model can be considered the core functionality of the framework as all client software interacts with the component model to realize some set of use cases. The FIRM component model is partitioned into service groupings based on the primary data models realized by each component in the grouping. Below is a diagram view of the FIRM component model.

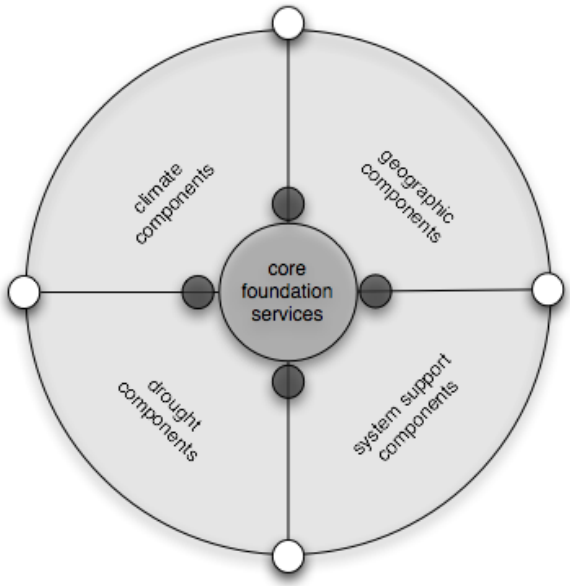

Figure 21. The FIRM component model

The four service groupings represented on the diagram above as sections of the circle represent the domain data classifications found in the data model: climate, drought, and geographic. An additional component grouping is included for system support; components in this group are responsible for the functionality of the system as a whole. Central to all components are the core foundation services that deliver framework functionality to the components and provide access to crosscutting operations.

The component model is partitioned based around the service offerings of the framework. This partitioning allows for common configuration and runtime elements to be delivered to components

requiring them. The goal of grouping components under a common component service is to create a level of abstraction above the components though which clients can interact with the functionality. In much the same way as multiple table definitions from a data model exist in a single data source, may component definitions from a component model will exist in a single component service. This organization allows for broad extension of the component access model in the future.

The component service organization relies heavily on the Manager/Accessor pattern discussed in Chapter 3, providing the basis for integration of the component model with the core foundation services. In the diagram, the interior and exterior small spheres represent the Manager and Accessor classes respectively. Components in each service grouping have access to Manager functionality to bind them to the core foundation services, exposing those services to the component. Components are referenced and delivered by an Accessor to allow clients to bind to them. The unified framework model is derived from the groupings of components to create service bindings to the core foundation services and clients.

As is the case with components found in the core foundation services, all components in the FIRM component model, with the notable exception of native components, are implemented as Enterprise Java Beans (EJBs). The implementation pattern for EJBs is that each component implements a Remote and optional Local interface. In the FIRM component model, standard naming conventions are used for each. The remote interface to the component is always given the component name, for example *MyComponent*. The component implementation takes the component name and appends bean to the end; *MyComponentBean*. If an optional local interface is also included 'Local' is appended to the beginning of the name; *LocalMyComponent*. This standard nomenclature allows easy recognition of components when packaged with other supporting classes.

The remainder of this chapter will focus on the components in each of the service groupings and their key support classes. FIRM maintains many classes to support the internal functionality of components that are never exposed to callers. For the sake of brevity a discussion of these support classes is omitted in favor of only discussing those class not defined in the domain model that are exposed to clients; as a result, the system support service components are omitted from this discussion. The discussion will address how each is bound to the corresponding data model to deliver integrated

functionality. An example of a client demonstrating how the components interact to form the unified framework model can be found in Appendix A.

CLIMATE COMPONENTS

The climate service is made up of components for accessing data defined by the persistent climate data model and for computing values based on that data. The service is comprised of two components, the *ClimateDataQuery* and the *ClimateMetaDataQuery*. This service is bound to the core foundation services with the *ClimateServiceManager*. The *ClimateServiceManager* maintains the JNDI names of persistent climate data model data sources, JNDI connection strings for the component instances, default values for missing tolerance, and connection information for external data supporting services[18]. The *ClimateServiceManager* allows the climate component model to bind to the framework. The *ClimateServiceAccessor* allows clients to bind to the climate model components, providing the *getClimateDataQuery()* and *getClimateMetaDataQuery()* methods, each returning an instance of the *ClimateDataQuery* and *ClimateMetaDataQuery* respectively. These interfaces are described in the following tables.

ClimateDataQuery	
Method	Description
getPeriodAnnualData getPeriodMonthlyData getPeriodWeeklyData getPeriodDailyData	The getPeriodData methods are used to access data values contained in a climate data source. These methods each take a *java.util.List* containing weather station IDs, two *DataTime* arguments to indicate the beginning and ending of the period over which the data should be queried, and a *DataType* enumeration to indicate the variable to be queried. **Returns:** *CalendarDataCollection*
getAvailableAnnualData getAvailableMonthlyData getAvailableWeeklyData getAvailableDailyData	The getAvailableData methods are used to access all data values within a station record. These methods take only a *java.util.List*of weather station IDs and the *DataType* enumeration to indicate the variable to be queried. Since stations do not all have the same historical records the absolute minimum and maximum date values are computed for the station list, OUTSIDE_OF_REQUEST_RANGE flags are set for stations that do not have a record reporting within the entire window. **Returns:***CalenadrDataCollection*
getHistoricalAverageAnnualData getHistoricalAverageMonthlyData	The getHistoricalAverageData methods are used to compute the average over the entire reporting record of a weather station. A single year of data is returned containing the averages values for each unit value, a unit value is implicit in the method (i.e. daily, weekly etc.). The method takes only a *java.util.List* of station IDs and a *DataType* enumeration.

[18] In this case the external service is the Applied Climate Information System, accessed through a CORBA interface.

getHistoricalAverageWeeklyData getHistoricalAverageDailyData	**Returns:** *CalendarDataCollection*
getAnnualDataNormals getMonthlyDataNormals getWeeklyDailyDataNormals	The getDataNormals methods are used to compute average values over an argument period of years for a list of weather stations. A single year of data is retuned containing the normal values for each unit value, a unit value is implicit in the method (i.e. daily, weekly etc.). The method takes a *java.util.List* of station IDs, the first year in the normal period, the last year in the normal period and a *DataType* enumeration for the variable to be queried.
getDailyDataNormals	**Returns:** *CalendarDataCollection*
getGrowingDegreeDayNormals	This method computes average growing degree values over an argument period of years for a list of weather stations. The method takes a *java.util.List* of weather station IDs, the starting year of the normal period, the ending year of the normal period, and the minimum and maximum daily temperatures for the GDD calculation. **Returns:** *CalendarDataCollection*
getForstFreePeriodNormals	This method computes the average frost-free period over a period of years for a list of weather stations. The method takes as arguments a *java.util.List* of weather station IDs, the first year of the period, and the last year of the period. The results are a single value for each year in the period. **Return**: *CalendarDataCollection*
getGrowingDegreeDays	This method computes growing degree for a list of weather stations over a time period. The method takes a *java.util.List* of weather station IDs, the *CalendarPeriod* for which the values should be computed, the starting date of the period, the ending date of the period, and the minimum and maximum daily temperatures for the GDD calculation. **Returns:** *CalendarDataCollection*
getFrostFreePeriod	This method computes the frost-free period for each year over a period of years for a list of weather stations. The method takes as arguments a *java.util.List* of weather station IDs, the first year of the period, and the last year of the period. The results are a single value for each year in the period. **Return**: *CalendarDataCollection*
getTemperatureDays	This method computes the number of time unit temperature days in a given period for a list of weather stations. The method takes a *java.util.List* of weather station IDs, the first date in the period, the last date in the period, a *CalendarPeriod* to indicate time unit type, a *TemperatureDayType* to direct the computation and a reference value. **Returns:***CalendarDataCollection*
getAnnualExtremeTemp	This method computes the annual extreme high or low temperature over a period of years for a list of weather stations. The method takes a *java.util.List* of weather station IDs, the first year of the period, the last year of the period, and a *DataType* enumeration to determine high or low computation. **Returns:** *CalendarDataCollection*
getPercentageNormal	This method computes the percentage of normal data value for a list of weather stations over a time period. The method uses the entire station record to determine the average value for a given time unit up until that point, the current value is then divided by the average to produce the % of normal. The method takes as arguments a *java.util.List* of weather station IDs, a *DataType* enumeration for the variable to compute for, the starting date of the period, the ending date of the period, and a *CalendarPeriod* enumeration to indicate the time unit to compute for. **Returns:** *CalendarDataCollection*

Table 25: The ClimateDataQuery component definition.

The *ClimateDataQuery* maintains an internal reference to a climate data source as defined in the core foundation services and injected using a *DataSourceInjector*. The component is stateless, with each method call encapsulating a full component transaction. Several enumerations including the *CalendarPeriod* are used to control time units, the *DataType* to indicate the weather station variable being queried, and the *TemperatureDayType* support the component. The *DataType* enumeration is a system-wide enumeration that contains values representing each type of data made available by FIRM components; these range from persistent data model data, to component computed information and component synthesized knowledge. A catchall is included for support as new data types are integrated in the system.

The methods on the *ClimateDataQuery* all return *CalendarDataCollection* objects, as all climate data in FIRM map to a set of calendar time sequences. Several of the methods include a *CalendarPeriod* argument to determine the time unit over which the computation or query is executed; others are broken out into unique method calls based on that information. There is no driver for one approach over another, the methods that are broken out do not incorporate any logic that would prevent them from being structured to take a *CalendarPeriod* argument.[19] Several of the methods on the *ClimateDataQuery* compute values dynamically as opposed to querying them from a climate data source. These computational methods are encapsulated in the *ExtendedClimateCalculations* class to preserve the *ClimateDataQuery* component as a query component only. The computed values are 'queried' at runtime from an instance of the *ExtendedClimateCalculations* object, allowing a black-box implementation of the calculation component.

ClimateMetaDataQuery	
Method	Description
findStations	The find stations method will locate stations based on search terms. The method makes several calls to the various station methods based on analysis of the search terms. The method takes as an argument a *StationSearchTerms* object. **Returns:** *MetaDataCollection<MetaDataType>*

[19] This was actually done in the Mozart builds of FIRM to create consistency within the component and decrease method clutter.

getMetaData	This method will return a single meta data type value for each station in a list of stations. It takes a *java.util.List<String>* of weather station IDs, a *MetaDataType* enumeration indicating the type of meta data and a *CalendarPeriod* to indicate the time unit for which the metadata is queried. **Returns:***MetaDataCollection<MetaDataType>*
getAllMetaData	This method will return all station metadata values for a list of weather stations. It takes a *java.util.List<String>* of weather station IDs, and a *CalendarPeriod* enumeration to indicate the time unit for which the metadata is queried. **Returns**: *MetaDataCollection<MetaDataType>*
getLongestPeriod	This method will compute the longest period of record for a list of weather stations. The longest period consists of the earliest starting date and latest ending date of the station records represented by the list. The method takes as arguments a *java.util.List<String>* of weather station IDs and a *CalendarPeriod* enumeration. **Returns:***TemporalPeriod*
getEndingDate	This method will compute the latest ending date in the persistent climate date model data source for a given time unit. It takes a single argument of *CalendarPeriod*. **Returns**: *DateTime*
getVariableMetaData	This method will query the available variable metadata for each station in a list of stations. The method takes a *java.util.List<String>* of weather station IDs. **Returns**: *java.util.Map<String, java.util.Map<DataType, VariableMetaData>>*
filterStations	The filterStations method is used to filter a list of weather stations by a set of runtime-defined operations. This method allows caller classes to generate precise lists of station IDs for use in other component operations. The method takes a *java.util.List<String>* of weather station IDs, a *java.util.List* of *VariableFilter* objects to apply to the station list, a *TemporalPeriod* indicating the overall period for which the variables must report data, and the overall tolerance for missing data before a station is filtered out. **Returns:** *java.util.List<String>*
getIntervalGaps	This method will compute a list of gaps in data reporting for each station In a list of stations. The method takes a *java.util.List<String>* of weather station IDs and a *DataType* enumeration to indicate the variable type for which the gaps are being searched. **Returns:** *java.util.Map<String, java.util.List<Interval>>*
isValidStation	This method is used to validate station IDs against a climate data source. It takes a *String* station ID. **Returns:***bool*
removeInvalidStation	This method will remove any invalid station IDs from a list of stations. It takes a *java.util.List<String>* of weather station IDs. **Returns:** *java.util.List<String>*

Table 26: The ClimateMetaDataQuery component definition.

The *ClimateMetaDataQuery*, like the *ClimateDataQuery*, maintains an internal reference to a climate data source; it also maintains a reference to a geographic data source. The tie to the geographic data model supports the explicit linkage between this component and the *SpatialQuery* discussed later. The *ClimateMetaDataCollection* makes use of the shared *DataType* enumeration for identifying the specific climate value that will be used for many of the queries. In several cases a *CalendarPeriod* is also

required. Time units must be provided for several metadata queries to allow the query logic to determine the end date metadata value. Time units partition end dates due to the fact that time unit end dates will differ over the reporting record. For example, when data are queried in the middle of the month the end date for daily data may be the current day, however insufficient data are available to generate a monthly summary, so the end date for a monthly time unit would be the last day of the previous month. By passing a *CalendarPeriod* to the metadata query methods, the caller can indicate which time unit to use when determining the end date.

The *ClimateMetaDataQuery* also includes an operation to 'fine tune' a list of stations based on the desired reporting frequency for variables in reported by the station. The filterStations() method allows a list of weather station IDs to be filtered to include only stations matching the parameters of a list of *VariableFilter* objects. This method provides upper bounds to the filter set that are checked for the aggregate station-reporting window. If the upper limits are not met, the station is excluded before the filters are applied. The remaining stations are run through each filter in order and removed from the list if the variables do not match the filter criteria. The method includes a flag to indicate that a 'deep filter' be executed. If this is false, the reported metadata from a climate data source is used as an input to each filter. If true, the actual data sequence for the period described by the filter is queried and iterated over to determine if it matches the criteria; this method considerably increases the runtime of the method call. The *MetaDataQuery* also provides a method to generate a list of station IDs based on a set of search terms. The *SearchTerms* object is generated from a search query string using ANTLR[20]. This allows the climate components to support search engine functionality. The implementation of a search engine, however, is left to the application model.

DROUGHT COMPONENTS

The drought component model is comprised of components for computing drought index values and querying drought impact reports. Drought components interact with a climate data source through

[20] http://www.antlr.org/

the climate component service in order to obtain climate data for drought index computations. The components in the drought component model are grouped to form the drought service; the service is made up of the following components: *DroughtIndexQuery, DroughtImpactQuery, DroughtImpactManager, NSMObject, PDSIObject,* and the *SPIObject*. The last three components in this model do not conform to the naming conventions for components due to being implemented as native code that is deployed in the native component container. The objects represented above are proxy objects for native library. Access to the components is made available through the *DroughtServiceAccessor*. Standard getter methods for each component type, excluding the native components, are used to access component instances. The *DroughtServiceManager* maintains configuration parameters for the drought components including the native component binding URLs, and ties the drought component model to the core foundation services. The *DroughtServiceManager* is also used to obtain a reference to the *DroughtImpactManager*, a system support object that is not remote. The following tables describe the interfaces for each of these components.

The primary drought component is the *DroughtIndexQuery*. This stateless componentis responsible for computing drought index values for all drought indices supported by the framework. Component operations complete over the course of a single synchronous method call. The following table describes the methods provided by the *DroughtIndexQuery*.

DroughtIndexQuery	
Method	Description
runNewhallSummary	This method runs a Newhall summary simulation on each station in a list. The inputs for the method are a *java.util.List<String>* of weather stations IDs and the start and end date of the period over which the simulation will be run. **Returns:***NSMSummaryDTO*
runCompleteNewhall	This method runs a complete Newhall simulation for each weather station in a list. The inputs for this method are a *java.util.List<String>* of weather station IDs and the start and end dates of the period over which the simulation will be run. **Returns:***NSMCompleteDTO*
computeWeeklyPdsi	This method will compute weekly PDSI values in a single year for each weather station in a list of stations. The number of values returned for the year will be 52 / step count; skipping periods equal to the step count. The arguments to the method are a *PdsiType* enumeration indicating the type of PDSI to compute, a *java.util.List<String>* of weather station IDs, a*PdsiWeeklyStep* enumeration to indicate the number of steps for the computation, and the ending date of the computation.

	Return: *CalendarDataCollection*
computeMultiYearWeeklyPdsi	This method will compute weekly PDSI values over multiple years for each weather station in a list of stations. The number of values returned for each year will be 52 / step count; skipping periods equal to the step count. The arguments to the method are a *PdsiType* enumeration indicating the type of PDSI to compute, a *java.util.List<String>* of weather station IDs, a*PdsiWeeklyStep* enumeration to indicate the number of steps for the computation, the ending date of the computation, and the year in which the computation will begin. **Return:** *CalendarDataCollection*
computeContinousMultiYearWeeklyPdsi	This method will compute weekly PDSI values over multiple years for each weather station in a list of stations, computing a step count PDSI for each week in the year. The arguments to the method are a *PdsiType* enumeration indicating the type of PDSI to compute, a *java.util.List<String>* of weather station IDs, a*PdsiWeeklyStep* enumeration to indicate the number of steps for the computation, the ending date of the computation, and the year in which the computation will begin. **Return:** *CalendarDataCollection*
computeMonthlyPdsi	This method will compute monthly PDSI values for each month in a single year for each weather station in a list of stations. The arguments to the method are a *PdsiType* enumeration indicating the type of PDSI to compute, a *java.util.List<String>* of weather station IDs, and the ending date of the computation. **Return:** *CalendarDataCollection*
computeMultiYearMonthlyPdsi	This method will compute monthly PDSI values for each month over multiple years for each weather station in a list of stations. The arguments to the method are a *PdsiType* enumeration indicating the type of PDSI to compute, a *java.util.List<String>* of weather station IDs, the ending date of the computation, and the year in which the computation will begin. **Return:** *CalendarDataCollection*
computeContinuousMultiYearMonthlyPdsi	This method will compute monthly PDSI values for each month over multiple years for each weather station in a list of stations. The arguments to the method are a *PdsiType* enumeration indicating the type of PDSI to compute, a *java.util.List<String>* of weather station IDs, the ending date of the computation, and the year in which the computation will begin. **Return:** *CalendarDataCollection*
computeWeeklySpi	This method will compute weekly SPI values in a single year for each weather station in a list of stations. The number of values returned for the year will be 52 / step count; skipping periods equal to the step count. The arguments to the method are a *java.util.List<String>* of weather station IDs, an integer to indicate the number of steps for the computation, and the ending date of the computation. **Return:** *CalendarDataCollection*
computeMultiYearWeeklySpi	This method will compute weekly SPI values over multiple years for each weather station in a list of stations. The number of values returned for the year will be 52 / step count; skipping periods equal to the step count. The arguments to the method are a *java.util.List<String>* of weather station IDs, an integer to indicate the number of steps for the computation, the ending date of the computation, and the year in which the calculation will begin. **Return:** *CalendarDataCollection*
computeContinousMultiYearWeeklySpi	This method will compute weekly SPI values over multiple years for each weather station in a list of stations. The number of values returned for the year will be 52 / step count; skipping periods equal to the step count. The arguments to the method are a *java.util.List<String>* of weather station IDs, an integer to indicate the number of steps for the computation, the ending date of the computation, and the year in which the

	calculation will begin. **Return:** *CalendarDataCollection*
computeMonthlySpi	This method will compute monthly SPI values in a single year for each weather station in a list of stations. The number of values returned for the year will be 52 / step count; skipping periods equal to the step count. The arguments to the method are a *java.util.List<String>* of weather station IDs, an integer to indicate the number of steps for the computation, and the ending date of the computation. **Return:** *CalendarDataCollection*
computeMultiYearMonthlySpi	This method will compute monthly SPI values over multiple years for each weather station in a list of stations. The number of values returned for the year will be 52 / step count; skipping periods equal to the step count. The arguments to the method are a *java.util.List<String>* of weather station IDs, an integer to indicate the number of steps for the computation, the ending date of the computation, and the year in which the calculation will begin. **Return:** *CalendarDataCollection*
computeContinousMultiYearMonthlySpi	This method will compute monthly SPI values over multiple years for each weather station in a list of stations. The number of values returned for the year will be 52 / step count; skipping periods equal to the step count. The arguments to the method are a *java.util.List<String>* of weather station IDs, an integer to indicate the number of steps for the computation, the ending date of the computation, and the year in which the calculation will begin. **Return:** *CalendarDataCollection*
computeRangeKbdi	This method will compute daily KBDI values over an argument period for each weather station in a list of weather stations. Arguments to this method are a *java.util.List<String>* of weather station IDs, and a *DateTime* representing the starting and ending day for the calculation respectively. **Returns:** *CalendarDataCollection*

Table 27: The DroughtIndexQuery component definition.

As a computational component, the *DroughtIndexQuery* generates information from raw data, maintaining references to supporting components rather than corresponding data sources. The primary data source for the *DroughtIndexQuery* is the *ClimateDataQuery*. Methods on the *DroughtIndexQuery* are responsible for obtaining the raw data needed for the index computations and then passing it to some delegate object. Delegate objects range from pure Java implementations of a drought index to bindings to native component implementations. The KBDI, for example, is computed by a pure Java delegate object, the *KeetchByramDroughtIndex* class defines the index functionality. The *DroughtIndexQuery* maintains an instance of the *KeetchByramDroughtIndex* allowing that object to compute values in a thread safe manner.The component utilizes standard climate data objects for most method computations due to the time sequenced nature of the computed values. A notable exception can be found for the

methods computing Newhall Simulation Model results. As discussed in Chapter 4, the Newhall returns several data sets for each simulation which must be encapsulated in a unique ADT.

Due to the computational nature of the functionality provided by the *DroughtIndexQuery*, the method calls utilize very few FIRM-specific data types. The arguments are primarily the station IDs and period over which the computations should be run. In the case of the PDSI and the SPI an additional step period is provided. The PDSI and SPI consider the deviation from 'normal' climate conditions for a given site with the conditions defined by some subsequence of time units within the period of computation. For example, a 3-month SPI will consider the deviation for normal precipitation conditions over a composite 3-month period. Run for a single year, the result will contain 4 values, one each for Jan. - Mar., Apr. - Jun, Jul - Sept., and Oct. - Dec. The step period is therefore a required input parameter. The SPI allows any number of weekly steps from 1 - 52 and monthly steps of 1 - 12. The PDSI differs in the way in which it computes results, limiting the step period to specific units: 1 month, and 1,2,4, and 13 weeks. In order to prevent incorrect periods being used, the *PDSIWeeklyStep* enumeration is used to restrict values. The PDSI also computes several index values internal to the computation, these can be used as the result value by passing in the desired enumerated value from the *PDSIType* enumeration.

The persistent drought data model includes definitions for several entities encapsulating drought report functionality. The drought component model includes two components for interacting with this data. Report data are made available through the *DroughtImpactQuery,* while report creation operations are provided by the *DroughtImpactManager*. The *DroughtImpactQuery* is stateless and contains several methods for returning drought impact reports from a drought data source. It maintains a reference to a drought data source provided by the core foundation services, as well as references to *MapArchive* and *MapQuery* components that are part of the core foundation services. The *DroughtImpactQuery*provides the following synchronous methods.

DroughtImpactQuery	
Method	Description
queryStateImpacts	This method will query all drought impact reports for a given state over a given time period. The arguments to this method are the *USState*enumeration for the desired state and the *DateTime* beginning and ending of the period over which the query should be run. An overload method requires only the *USState* enumeration.

	Returns:*ImpactQueryResult*
queryCountyImpacts	This method will query all drought impact reports for a given county over a given time period. The arguments to this method are the *USCounty* value of the desired county and the *DateTime* beginning and ending of the period over which the query should be run. An overload method requires only the *USCounty* value. **Returns:***ImpactQueryResult*
queryZipCodeImpacts	This method will query all drought impact reports for a given region defined by a US zip code over a given time period. The arguments to this method are the *USZip* value of the desired area and the *DateTime* beginning and ending of the period over which the query should be run. An overload method requires only the *USZipCode* value. **Returns:***ImpactQueryResult*
queryImpacts	This method will return all impacts over a given period of time for an arbitrarily defined geographic region. The arguments to the method are a *BoundingBox* describing the region, the start and end *DateTime* of the period, and a *SpatialReferenceType* indicating the type of reports to be queried. **Returns:** *ImpactQueryResult*
queryStationImpacts	This method will return all impacts that can be associated with a given weather station over a period. The method takes a *java.util.List<String>* of weather station IDs as well as the start and end *DateTime* of the period. **Returns:** *ImpactQueryResult*
searchAllReports	This methods will return all impact reports matching a given search string. The method takes a single *java.lang.String* query string. **Returns:***ImpactQueryResult*
loadReports	This method will load impact report entities for all of the *ShadowImpact* objects in the argument *ImpactQueryResult*. **Returns:** *java.util.Set<ImpactReportBean>*
loadReport	This method will load an impact report entity for the argument ID value. **Returns:** *ImpactReportBean*
queryDroughtMonitor	This method will compute the spatial frequency of drought monitor classifications across the area of effect for a drought report given as the ID of the *ImpactReportBean*. **Returns:** *java.util.List<SpatialFrequency>*
searchReportsByDroughtMonitor	This method will return a list of impact reports that contain impact regions that have an occurrence of some drought monitor classification over a period of time. The arguments to this method are a *DMClassification* enumerated type to indicate the DM classification, and the *DateTime* period start and end objects. **Returns:***ImpactQueryResult*

Table 28: The DroughtImpactQuery component definition.

Query operations on the drought data set operate against space, and space and time. Access to report data is limited by use case to the primacy of space when accessing drought reports. The query methods provided by the *DroughtImpactQuery* are overloaded to allow optional temporal constraint arguments to be passed, allowing a specific period of time to be considered over some space. The queries map to

spatial definitions provided by the geographic data model. The searchAllReports() method allows an unconstrained query against all reports. The method will query impact reports based on the occurrence of the query string in the title or description of the report. The method will analyze the string such that if the value matches a *ReportCategory* type, all reports with that category will be returned.

The linkage to the spatial functionality provided by the geographic component model enables several unique queries based on the spatial nature of the data but which do not appear directly in the data set. Namely, the ability to link this data based on its spatial aspects to other unrelated purely spatial data sets; this process forms what would be considered 'knowledge' in the framework. The examples of this process implemented in the *DroughtImpactQuery* component are the *queryDroughtMonitor()* and *searchReportsByDroughtMonitor()* methods. These methods links the drought impact data set with the US Drought Monitor[21] data set. The drought monitor is an example of a purely spatial data set whose access is enabled through core foundation services. The *queryDroughtMonitor()* method will compute the DM classifications and their relative frequency in the effected region described by the report over a period of time; yielding Drought Monitor results from drought impact data. The *searchReportsByDroughtMonitor()* method does the inverse, returning drought impact results from Drought Monitor data.

The *DroughtImpactManager* is a stateful component that provides drought report management functionality to clients. This functionality is, in the current implementation, limited to system components only; the component interface is *Local*. Since the component is designed around import use cases, it is transactional and allows multiple reports to be persisted at the same time; this transactional nature necessitates that the component instance be stateful. The transactional nature of the component protects against orphan data being created in the system in the event of loss of communication with the component. Data is only persisted to the drought data source after a transaction is started, entities

[21] The US Drought Monitor is a cartographic product authored weekly by the National Drought Mitigation Center. The drought monitor maps depict the 50 US states and the severity of drought occurring in various regions. Drought severity is subjectively assessed through various data sources. More information can be found at: http://drought.unl.edu/dm/.

updated, and the transaction explicitly committed. The *DroughtImpactManager* provides the following synchronous methods.

DroughtImpactManager	
Method	Description
startTransaction	This method creates a transaction for updating a drought data source drought report entities. **Returns:** *void*
commitTransaction	This method will commit all changes made to the data model during the transaction. **Returns:** *void*
rollbackTransaction	This method will close the current transaction and lose any changes made. **Returns:** *void*
isInTransaction	Flag method to determine of the current object instance has a transaction. **Returns**: *bool*
loadReport	This method will load a report of the argument type. The method takes a *java.lang.Class<reportType>* argument to determine the report type and a *long* ID of the report. The method is parameterized as a generic return which will be determined by the report type argument. **Returns:***<reportType extends BaseReport>*
addReport	This method will persist a *BaseReport* instance to a persistent drought data model data source, creating an entry for each entity value contained in the object graph. The method takes *java.lang.Class<reportType>* and *BaseReport* arguments. **Returns:** *<reportType extends BaseReport>*
updateReport	This method will persist updates to a *BaseReport* instance to a persistent drought data model data source, updating all entities for each entity value contained in the object graph. The method takes *java.lang.Class<reportType>* and *BaseReport* arguments. **Returns:** *<reportType extends BaseReport>*
removeReport	This method will delete from a persistent drought data model data source the entity referenced by the *BaseReport* argument as well as all entities in its object graph. **Returns:***void*
removeCategory	This method will delete a *ReportCategory* from a persistent drought data model data source, any entities referencing the *ReportCategory* instance will be updated when the transaction is committed. **Returns:***void*

Table 29: The DroughtImpactManager component definition.

The *DroughtImpactManager* maintains a reference to both a drought and geographic data source. Components of the drought component model make significant use of the geographic data model. For this reason, a single data source is used to instantiate both data models. The *DroughtImpacctManager* interacts with each data source through a single JPA *EntityManager*. This

component relies heavily on the persistence functionality provided by JPA, wrapping that functionality to create instances of transient objects. The CRUD methods of the *DroughtImpactManager* each treat the report object instances as a *BaseReport,* allowing a single CRUD operation for all report types rather than one for each. The CRUD operations make use of generics to appropriately cast the entity that is returned. CRUD operations return an instance of the entity to ensure that, after performing an operation, a client can maintain a reference up-to-date entities (i.e. those tied to a persistence context). For this reason, when updating an entity the held reference should always be updated.

Native Components

The drought component model contains several components that are implemented as native components and deployed in the native component container. These include implementations of the Self Calibrated Palmer Drought Severity Index [15,16], Standardized Precipitation Index [18] and Newhall Simulation Model [14]. Each of these indices were taken directly from a previous decision support system and were not ported to Java due to significant testing taking place over several years. The implementations, written in C and C++, were updated for FIRM to include a component interface that allows binding between the native code and the higher-level Java code. The native component structure is made up of the native library, a native interface and a Java class instance with a remote interface.

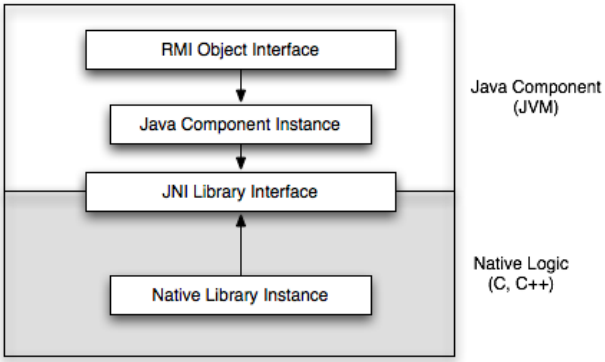

Figure 22: The structure of a native component

Higher-level components interact with native components through the RMI object interface. The running object instance is bound to the native component container maintained by the integrated component runtime. The native code is compiled as a library and is interacted with by the Java component instance

using the Java Native Interface (JNI). Each library has a component interface associated with it to provide JNI method bindings to the library.

JNI allows Java code to execute and pass methods to C++ method implementations. Call marshalling using variable reference passing add considerable overhead to the computation and so the FIRM implementation opts for direct memory access between components. The role of the Java component instance is to interact with higher-level components to obtain data needed by the native libraries. This data is then placed in previously allocated memory and a pointer passed to the library. On return from the library, the Java component instance retrieves any values from a memory pointer returned by the library. The use of direct memory I/O provides a considerable increase in performance. Direct memory access is provided by the Java Native IO library that is part of the Java Runtime Environment. After retrieving computed values from memory, the Java component instance creates an instance of a transient data model entity, populates it and returns it to the calling component. The *PDSIObjct, SPIObject,* and *NSMObject* are Java component instances that each communicate with the appropriate library through a JNI library interface.

The *DroughtImpactQuery* is the only calling component to the native components, serving as a proxy component for other components to use for computing drought index values. As a result, each of the objects named above contains the same method signatures found for the functionality on the *DroughtIndexQuery*. The *DroughtIndexQuery* is used as a catch-all for drought index computations so that all calling components interact uniformly using EJB. This allows internal calling components to benefit from built in service functionality provided by the EJB container like instance injection and transaction management. It also facilitates a more uniform interaction model with core foundation services.

GEOGRAPHIC COMPONENTS

The geographic service is comprised of two components, the *SpatialQuery* and the *LayerStatisticQuery*. These two components interface with a geographic data source and GIS services provided by the core foundation services. Their primary purpose is to computationally relate data elements to spatial data, allowing for data to be retrieved based on spatial definitions rather than values

explicitly defined by the data model for which values are being queried. The service includes the *GeographicServiceAccessor* for obtaining references to each component and a *GeographicServiceManager* to bind it to GIS components and the core foundation services. The *GeographicServiceManager* maintains configuration properties for accessing both GIS and static spatial data sets.

The service includes an object for accessing persistent geographic model geographic boundaries called the *SpatialQuery*. In addition to access to the geographic data source, the *SpatialQuery* also links to the climate data source, providing methods to query weather stations based on geographic boundaries. This stateless component maintains both a JDBC and JPA connection to a geographic data source as well as a JDBC connection to a climate data source. Both JPA and JDBC are used to facilitate more robust queries while allowing for reuse operations on the entities. The following synchronous operations are provided by this component.

SpatialQuery	
Method	Description
getCountiesByState	This method will query a geographic source model data source for all US counties that are defined in a US state. The argument for this method is a*USState* enumeration. **Returns**: *java.util.Set<USCounty>*
searchCounties	This method will query a persistent geographic data model data source for all counties based on an argument county name string using fuzzy matching. **Returns**: *java.util.Set<USCounty>*
getCountiesByFips	This method will query a geographic data source for all counties based on a list of FIPS code. The method takes a *java.util.List<String>*as an argument. **Returns**: *java.util.Set<USCounty>*
searchCountiesByState	This method will query a geographic data source for all counties based on a name string constrained to a US state using fuzzy matching. This method takes a *java.lang.String* county name string and a*USState* enumeration as arguments. **Returns**: *java.util.Set<County>*
getCountiesByRegion	This method will query a geographic data source for all counties where a majority of the defined area exists in an argument area definition. The method takes a *BoundingBox* argument. **Returns**: *java.util.Set<USCounty>*
getStatesByRegion	This method will query a spatial engine for all states where a majority of the defined area exists in an argument area definition. The method takes a *BoundingBox* argument. **Returns**: *java.util.Set<USState>*
getCountyById	This method will load a US county entity from a geographic data source based on the argument county ID.

	Returns: *USCounty*
getCityById	This method will load a US city entity from a geographic data source based on the argument city ID. **Returns**: *USCity*
getCitiesByState	This method will query a geographic data source for all US cities that are defined in a US state. The argument for this method is a*USState* enumeration. **Returns**: *java.util.Set<USCity>*
getCityByZip	This method will query a geographic data source for all US counties that are defined in a US zip code. The argument for this method is a *java.lang.String* zip code. **Returns**: *java.util.Set<USCity>*
queryStations	This method will query all weather stations that exist in a given geographic extent. The method takes a *BoundingBox* as an argument. **Returns**: *StationList*
getStationsForDefinedRegion	This method will query all weather station IDs for stations that exist in a given geographic extent. The method takes a *BoundingBox* as an argument. **Returns**: *java.util.List<String>*
getStationsForState	This method will query all weather station IDs for stations that exist in a state. The method takes a*USState* enumeration as an argument. **Returns**: *java.util.List<String>*
getStationsByZipCode	This method will query all weather station IDs for stations that exist in a given zip code within a *k* mile radius. The method takes a *java.lang.String* zip code and *int* radius as an arguments. **Returns**: *java.util.List<String>*

Table 30: The SpatialQuery component definition.

The operations provided by the *SpatialQuery* are generally divided by the type of geographic boundary being used for the query, supporting all US states, counties, cities, zip code, and custom defined spatial extents. The component also provides logic to search for US counties. This method operates as a standard query but does not require an exact match for return data. A county name string is used to perform a fuzzy search of persistent geographic data to return any matching counties; a similar method allows the query to be constrained to a US state. Collections of the corresponding entities or ID values are returned by the operations.

The *LayerStatisticsQuery* is a geographic component that allows classification data to be extracted from spatial objects (i.e. maps). This stateful component includes operations to compute the frequency of a classification occurrence over either time or space for a spatial data object represented by an *ArchivedMap*. This is an example of a component that builds exclusively off functionality provided by

core foundation service components. These operations are general enough that this component was considered a candidate for the core foundations services; however, due to its use of data elements from the persistent geographic data model it was included as a part of the FIRM component model. The inclusion of this component in the FIRM component model is an example of the reuse of spatial elements/operations across domain models discussed previously. The following synchronous operations are provided by the *LayerStatisticsQuery.*

LayerStatisticsQuery	
Method	Description
queryPercentageByBoundingBox	This method will extract classifications for a map and compute their frequency over a defined geographic region. This method takes an *ArchivedMap* and *BoundingBox* as arguments. **Returns:** *SpatialFrequency*
queryPercentageByCounty	This method will extract the classification for a map and compute their frequency for a given county. This method takes an *ArchivedMap* and a *USCounty* as arguments **Returns:** *SpatialFrequency*
queryPercentageByState	This method will extract the classification for a map and compute their frequency for a given state. This method takes an *ArchivedMap* and a *USState* as arguments **Returns:** *SpatialFrequency*
queryPercentgeByDateRange	This method will extract the classification for a map and compute their frequency for a given geographic region over a given time interval. This method takes a *DateTime* start and end value, a *ViewableLayerType* to indicate the spatial data type to compute against and a *BoundingBox* region as arguments. **Returns:** *SpatialFrequency*

Table 31: The LayerStatisticQuery component definition.

The *LayerStatisticsQuery* is a stateful component due to its reliance on an active connection to a GIS engine as provided through a reference to a *MapQuery* instance (also a stateful object). Due to the overhead associated with opening a connection to and loading a spatial object from the GIS engine, the object maintains the connection to the GIS engine across calls from the caller component. This prevents the need to reopen a connection or reload an object to perform a query against a different spatial extent for the same map. Loading a new map, either for a new map type or different period type, requires that the GIS server connection be opened to a newly loaded map object, incurring the same penalty as creation of the object. The *LayerStatisticsQuery* uses the GIS engine to extract classification data for an *ArchivedMap* and determine the frequency of occurrence in a region on the map.

Spatial frequency can also be computed over time for a series of *ArchivedMap* instances using the *LayerStatisticsQuery*. This operation relies on a data type classification defined by the *ViewableLayer* enumeration. This enumeration contains a list of map data types available to the system and having associated *ArchivedMap* objects. An *ArchivedMap* maintains a date period over which the data represented by the map are 'valid'. Using this data, the *LayerStatisticsQuery* will extract classification data from each 'valid' map over a specified time interval. The frequency of a classification for a defined region of that map will be computed over time, providing a summary of occurrence of a classification across time for a fixed spatial extent.

A FRAMEWORK ARCHITECTURE

The realization of a component model provides a gateway to a domain data model and completes the overall framework architecture. Paring the domain model (inclusive of both the data and component models) with the core foundation services leads to an extensible framework to support both the integration of new component services, components to a service, and application development atop the components. The framework itself requires an application model in order to drive software application use cases. That is to say, the framework does not provide end-user functionality but serves as a foundation for software providing such functionality and the data and operations needed to support it. The goal of the unified framework is to support the development of application model instances through a uniform set of operations provided by components and tied together through a data model, while allowing ease of extension in the future.

A single framework should support many application models within a single domain. If the framework component and data models are too specific, supporting a single application model, generalized domain logic encapsulation or definition has not been achieved. This prevents the usefulness of the framework in supporting data-driven decision support. Failure to properly generalize operations and data definitions results in a framework that cannot easily adapt to changes to existing, or inclusion of new, data sets. Extension of such a framework would require significant modification to the application model.

In order to evaluate the efficacy of the framework architecture for FIRM, and ensure that the model is sufficiently extensible, an analysis of the integration of application models for decision support systems in the agricultural domain with FIRM would need to be preformed. Chapter 6 of this thesis discusses that work by providing two decision support systems as case studies, detailing the differences between the application models and the efficacy of FIRM to support the use cases they represent.

CHAPTER 6: CASE STUDY – TWO DECISION SUPPORT SYSTEMS

This chapter presents as a case study two web-based decision support systems that used FIRM as a foundation for acquisition of data to support services. Each represents a unique application model in the domain of agricultural decision support that was able to take advantage of the architecture of FIRM. The first provides an example of an application model created expressly to take advantage of FIRM, demonstrating the specific benefits of the data integration approach embodied by FIRM. The second provides an analysis of FIRM as a framework to support an existing application model, discussing the benefits to refactoring, stability, and integration. The focus of this discussion revolves around the insights gained through the use of FIRM in the development of a new system and its integration with a legacy system. The process of development and extension led to the following insights into the use of FIRM as a component architecture for data integration:

1. The provision of orthogonal framework logic through a set of managed services frees the framework component model to incorporate domain-specific business logic, supporting greater decoupling of components and increasing application model flexibility (component decoupling);
2. The internal decoupling of the component model supports the development of a strongly typed, well defined domain data model, which can act as a connector architecture to support decoupling of the component model from the application model (data connectors);
3. The decoupling of the application from component model supports the integration of new data sets through tool development in the application domain while at the same time allowing new processes to be easily incorporated into the component model with no impact to the application model (model decoupling);
4. The encapsulation of the domain model into independent, well defined units, supports software developers in the creation of applications from the framework. This support facilitates greater software evolution in the space by allowing developers to more easily incorporate use cases in their software systems (developer support).

Each of the lessons learned addresses a different aspect of the overall framework architecture, building from the lowest level framework elements to the most abstract application concepts. The remainder of this chapter discusses each of these insights as demonstrated though the two cases studies, beginning with an introduction to each system.

THE GREENLEAF PROJECT: A REFERENCE IMPLEMENTATION

Beginning with the second major milestone of FIRM, a web-based decision support system was being developed in order to provide a reference implementation for a FIRM application model. The GreenLeaf Project (GreenLeaf), a decision support system designed to provide users with integrated data in the agricultural domain, was that model. The complete FIRM architecture, including GreenLeaf as an application model, can be seen in Figure 24.

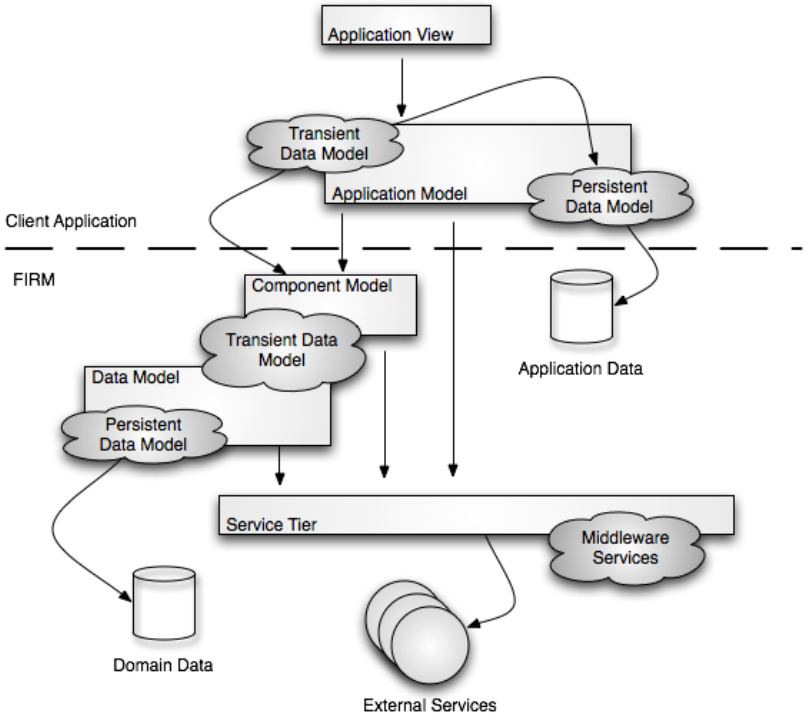

Figure 23: FIRM application model diagram (The GreenLeaf Project)

In this view of the system, FIRM is composed of the service, data and component models. This is shown beneath the dotted line in. The application model is built based on the available services provided by FIRM and incorporates the use cases defined for the specific application. As a result of application-

specific use case definitions, the application model may contain both data and component models. These models would represent domain logic that is specific to the application in question but may link to common data elements from FIRM; for example, the definition of an RSS feed service for drought impact reports. The application-side data and component models are not shown in the diagram, as they are sub-elements of the overall application model.

A characteristic of the component architecture is the application model is able to act as a view. Application view can be thought of as the element of the application responsible for data visualization and user interaction with the component model. In this regard, the application model can be thought of as a view model for FIRM. Breaking out the view from the model in this way recognizes the fact that interaction and presentation models my differ dramatically based on device specific use cases and, particularly in the web ecosystem, may exist in several forms for a single application model; consider a website with both pages designed to be viewed on mobile devices and pages for standard browsers. This segmentation of the architecture, and its view relative to integration with FIRM, allows for several interesting extensions that are discussed later.

With this loosely coupled view of the complete architecture for FIRM, GreenLeaf was designed to deliver data through task-oriented workflows. The underlying component structure was masked in GreenLeaf by the implementation of the use cases. The decision support system enables information discovery through a small set of well-defined tasks as opposed to a collection of tools. The tasks that GreenLeaf supports are: search, subscribe, access, and map. Data are presented to the user in the context of each of those tasks, allowing novice users to easily access data without significant background knowledge while at the same time supporting more advanced users requiring specific data sets. The model for decision support presented by GreenLeaf differs significantly from the workflow-driven models of pervious systems in the agricultural domain. This characteristic was born out during development through a process that started with a more workflow-driven set of use cases that evolved to take advantage of the FIRM data integration structure. That fact that data integration was easily achievable at several points in the system allowed for the development of more robust, task-oriented, tools. The evolution of the use cases resulted in the release of a highly extensible umbrella for decision support that

not only provided a POC for FIRM, but also enables the development of tools for decision support in several dimensions including web, desktop and mobile devices.

THE NATIONAL AGRICULTURAL DECISION SUPPORT SYSTEM: AN EXTENSION

In contrast to the GreenLeaf Project, The National Agricultural Decision Support System (NADSS) [19,20] was a decision support system built to provide agricultural data to users of three primary types: decision makers, researchers, and producers. The system was structured around the concept of transitioning data to information and paring information, data, and other information in order to form knowledge. The NADSS application conformed to the standard conventions for decision support of the time, beginning development in 2002 and being released in 2004[22]. NADSS was built on the 3Co framework discussed in Chapter 2. NADSS partitioned the tools that it provided based on the data, information, or knowledge classification. While the user types were not intended to be mapped to the data types, the use cases for the application were very tool-oriented, and geared primarily to researchers. The following diagram depicts the architecture for NADSS [19].

[22] This work discusses NADSS 2.x. A pervious version, 1.x was developed from 1999 – 2002. The 1.x version did not take advantage of a supporting data framework and therefore is not applicable for this comparison.

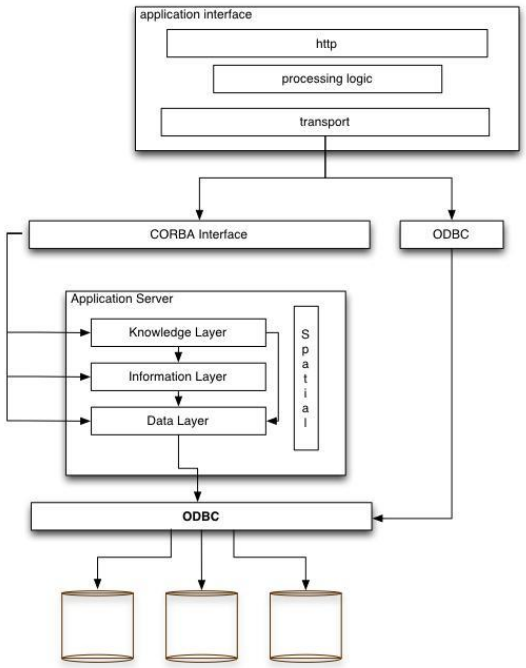

Figure 24. NADSS application architecture

The application architecture of NADSS looks very similar to that of FIRM. Figure 25 depicts a separation of the application model from the underlying component model, here represented by the 3Co framework. Missing, however, are the explicit service and data model definitions. The 3Co framework provided a component model that was accessed by the application model using CORBA data types and SQL. In order to maintain a loose coupling between the application and component models, the NADSS application model provided an abstraction layer for translating CORBA data objects to more verbose Java types. This could be considered a quasi-data model, however the abstraction stopped short at a formal definition of domain objects, relying on more 'standard' ADTs to pass data in the system.

The NADSS application encapsulated several tools for assessing risk in an agricultural landscape; these have largely been discussed previously and included basic climate data access, Self Calibrated Palmer Drought Severity Index, Standardized Precipitation Index, Keetch-Byram Drought Index, and Newhall Simulation Model. 3Co grouped components as a layered architecture as discussed in Chapter 2. The climate data components belonged to the data layer, the drought components to the information layer, and several higher-level static spatial products for accessing soil related risk belonged to the knowledge layer. Intra-component communication (i.e. a drought component using climate data, or a

knowledge component using spatial computations) was facilitated using explicitly defined static connectors.

NADSS was used as an application model for FIRM in order to serve as a test case for FIRM component computations. Tool functionality from NADSS, backed by 3Co,was used to verify the same functionality in NADSS backed by FIRM. In addition to serving as a test case for the development of the FIRM component model, NADSS also allowed the flexibility of FIRM to be demonstrated by integrating application functionality not designed to take advantage of the framework. A version of NADSS was released with FIRM substituted for 3Co beginning with second milestone of FIRM (Thresher M2).[23] FIRM was not completely substituted for 3Co in support of NADSS, rather significant components within 3Co were deprecated and disabled in the architecture. Those applications that utilized the depreciated components were updated to make use of FIRM components through the integrated runtime container. The application model for NADSS was modified only to the extent that required changing the data integration layer. Higher-level tools and application functionality were not modified to utilize the FIRM domain model; the previous Java objects were maintained and populated with data from the FIRM data model as opposed to the CORBA model.

The substitution of FIRM components did not take place for all 3Co components due to tight coupling of the application model with the 3Co spatial layer, and by proxy, knowledge layers. While the information and data layer components were sufficiently generalized to allow for FIRM substitutions with minimal impact to the NADSS application model, spatial and knowledge components were designed to specifically support application model use cases. As a result, the abstraction between these aspects of the component and application models were minimal. In order to effectively substitute FIRM spatial functionality to support NADSS spatial functionality, significant modification to the application model would have been required. The goal of the substitution was not to completely replace 3Co as the supporting framework for NADSS, but to demonstrate the efficacy of the architectural approach for

[23] The first Thresher milestone included the major components of the core foundation services.

building FIRM and to provide QA metrics during the development process. For this reason, the extension effort for NADSS paired FIRM and 3Co, rather than replacing 3Co entirely.

What follows is a continuation of the discussion of the integration of FIRM for NADSS, organized to focus on each of the key insights to data integration that we gained through the case studies.

COMPONENT DECOUPLING

FIRM is composed of component and data models that are integrated with the core foundation services to create the unified framework. The abstraction of orthogonal framework services into the core foundation services frees the components to encapsulate the logic responsible for realizing the use cases for each component. This prevents the need to tie components together explicitly with framework logic for various orthogonal operations discussed in Chapter 3. The decoupling of components allows each component to act as an atomic unit of functionality and results in the integration of new components taking place with no required impact on the rest of the framework, including the application model. This latter point is key when considering the development of new software systems that are defined by the application model. New components can be added to provide new functionality without modification to other components in the system. This increases the general stability of the system, as errors in one component effect only those components relying on it. The component decoupling also supports more efficient unit and regression testing; a new component can maintain its own set of unit tests and regression only is required on those components that interact with the addition.

Furthermore, the core foundation services are able to provide a 'framework' context to the components at runtime, allowing applications developed on the component model to transparently take advantage of both framework services and component integration. The communication protocols, configuration activities and general low-level data wiring that are often required in order to interact with distributed systems are incorporated into a single environment variable. This framework middleware of sorts supports application modeling in isolation, freeing the model to encapsulate the use cases with loose bindings to the underlying component model; a model required to access the data necessary to achieve the use cases. This level of abstraction and its benefit are demonstrated through the use of the

framework to support both a legacy application transparently as well as facilitate the integration of completely disparate data in an application model to support the development of new tools.

The connector architecture employed by 3Co and the component coupling that occurred as a result of itled to a static application model definition for NADSS. Tool implementations utilized single component communication for computation. For example, each drought tool was implemented as a component with a connector to the climate components. To generate PDSI data, the application would make calls to the drought component supported by a connector to a data component. Data integration was most easily achieved by defining connectors through components in the component model and binding to these in the application model. While the application could integrate data by making multiple component calls, significant logic would be required to deal with data conversion between CORBA and Java components, resulting in a 'stove pipe' type application architecture. Whether integrated in the component or application models, new application functionality required the introduction of data translation elements and connectors between existing components. The result was a component model that was not, in practice, truly black box to the application model.

Integrating a new information tool in NADSS would result in the development of a new set of connectors. While these connectors could be defined using the ADL and casetools provided with 3Co, the process of adding the new tool required modification to the component model by introducing a new static connector. This fact can be seen in the incomplete replacement of 3Co with FIRM. In order to support spatial data integration for several tools, 3Co had introduced connectors to bind components to support the generation of new data. This process had pushed a portion of spatial service logic into the component model, and resulted in the modification of several components. As a result, these components could not be replaced without substituting their functionality in the application model or by introducing application specific components to the component model. The component coupling that was required to support the functionality prevented it from being easily replaced by another system.

By contrast, FIRM was able to integrate several service specific data relationships with no modification to the component model. The component isolation allowed the JBoss application server to be replaced by the Glassfish application server between FIRM M3 and M4, integrating several new

middleware services with little impact on the component model. Several of the geographic components in FIRM were also used make use of spatial data provided though 3rd party spatial services (ArcGIS Server). The connection logic responsible for binding the components to the spatial services was defined purely in the core foundation services with each of the components atomically providing data to the connection. As a result, the implementation of the core integration services spatial component could change without requiring that other components to be updated and with no impact to the geographic component connections.

This decoupling was demonstrated by the integration of the US Drought Monitor spatial data set and drought impact reports. The *DroughtImpactQuery* was extended beyond its original specification to include two methods for computing spatial frequency based on the Drought Monitor data set. This connection was facilitated through the unmodified use of the *LayerStatisticsQuery*, backed by the *MapQuery* and tied together by the data model. When integrated with the display of drought impact reports, a new view of impact reporting emerges, allowing ground-truth analysis of Drought Monitor classifications to take place. This functionality was developed as an afterthought based on the needs of a researcher and required the introduction of two methods on a component. These methods were originally developed in a free-standing analysis application; they were later included directly in the *DroughtImpactQuery* due to general usefulness.

DATA CONNECTORS

By freeing the component model from the framework, the component model is able to address the specific needs of the domain model; in the case of a data-driven system: to provide data to higher-level components and applications. This freedom allows for the development of a rich and verbose domain data model that can act as a connector architecture between the application and component models. It is this attribute of the system that was unanticipated and was the most important aspect of data integration. Without the domain data model acting as an abstraction between the component and application models, cross-model logic of some kind would be needed to allow integration of components with the application. The lack of this coupling, or rather its provision through the data model, frees both

the component and application model to encapsulate only logic and data relevant to its use cases. The data model acts as a semantic bridge between the two, allowing each to evolve independently from the other. The decoupling enabled by the data model also allows for multiple application models to be built against a single component model, increasing the flexibility of the framework in support of an overall ecosystem of software components within a single domain.

In allowing the decoupling of the component and application models, the data model achieves the goal of data integration at multiple points in the system. Base data can be easily integrated through the development of new components that take advantage of a static or dynamic data source; an example of which is the generation of drought index data from a climate data source accessed through a climate component. Dynamically formed data can be integrated at the application layer through the development of new tools that combine baseline data to visualize some new result. In each case, the inclusion of the new data does not need to impact other elements of the system, and once included it is free for use to all other components of the system due to the common domain semantics defined by the data model. This is most clearly demonstrated in the differences between external mapping components in NADSS and GreenLeaf.

One of the most significant developments introduced in GreenLeaf as a result of decoupling was the use of Goolge Maps for spatial data visualization. NADSS had required a tight coupling between application models and spatial operations. A result of the model decoupling was the limitation of the use of external visualization elements for spatial display. While Google Maps (and similar products) were not available during the development of NADSS, retrofitting them later would require significant development of new spatial components and connectors to drought and climate data providers. The generation of spatial visualizations in NADSS, accomplished with 3rd party tools, required significant coupling of the application and component models due to a lack of formal definitions for spatial operations being encapsulated in a data model. Within GreenLeaf, the concept of completely externalizing visualization of spatial data was a natural fit with the architecture. The structure of the spatial components provided by the core foundation services resulted in components that were data retrieval and computation oriented. Display was left completely to the application model, connected through the data model. As a result, the

GreenLeaf project was able to use both Google Maps and Microsoft Virtual Earth as visualization platforms, settling on Google Maps in response to use case requirements.

MODEL DECOUPLING

The complete decoupling of the application from the component model leads to tools that do not directly conform to the layered architecture that was explicit in the structure of NADSS. The FIRM service model, which replaced the layered deployment structure found in 3Co, is also hidden in the organization of the application models built on FIRM. The application model is able to focus solely on the use-case driven requirements for the software, using the data model to connect results from component calls. The result of this is that building software around the decision support use cases becomes considerably less dependent on the underlying framework. GreenLeaf milestones were tied to Thresher milestones, yet the application development process was not. Tool support appeared in GreenLeaf as the basic data-providers became available in FIRM. An example of the use-case driven software functionality is the introduction of a weather station search engine with the first milestone of GreenLeaf. The search engine uses the *ClimateDataQuery* and *ClimateMetaDataQuery* to provide a google-like search experience to the user. The data are organized around that use case, giving the user the ability to access it without having to first know where to look for it. This implementation took place off a build of FIRM designed primarily to support NADSS rather than this functionality.

A second benefit of the decoupling of application from component models is that it allows application model workflows to be task-centric as opposed to tool centric. This wokflow difference can be seen in contrasting NADSS and GreenLeaf. In NADSS, generating PDSI results using the PDSI tool was a single task; in GreenLeaf, PDSI data results are present in the system in a number of tasks where presenting the values make sense. For example, when a user searches for weather stations, the resulting page that is generated for a weather station result contains variables of many formats for the single station; values include raw climate data, PDSI, SPI, NSM, data visualizations of data as charts, and drought impact report data. These values can then be displayed over time for the single point. The use cases in this example are task-driven: 1) search a station; 2) view a station; 3) change station over time to discover

data. This approach integrates data elements relevant to the specific use case when required by the task. The result was a much more natural workflow based decision support system. In NADSS, all interaction was limited to workflows associated with specific tools; the tools being defined by various connectors in the component model.The complete decoupling of the application from the component model allowed the development of GreenLeaf and FIRM to take place in parallel, with GreenLeaf use cases being developed around data, as FIRM was able to provide it.

By decoupling the application from the component model, multiple application models can be created from the single framework. For example, higher-level services can be constructed to encapsulate specific data use cases, building targeted data results for delivery to mobile devices. Such an extension was developed as part of the 2.0 (codename Mozart) release of FIRM. A RESTful service application that incorporated several use cases found in GreenLeaf was constructed to push integrated data sets to a mobile application running on the iPhone OS platform. At the outset of FIRM development, the platform did not exist and the use case was not considered. However, introducing no new technology to the framework, the application model has been able to evolve to include the new platform. Several desktop applications were also developed to deliver specific data to users for a specific decision support activity. The framework enabled an evolution of the application model from a single web-based application to an entire ecosystem of tools supporting activities in the decision domain.

The explicit decoupling of application models from the framework that is the core of the architecture for FIRM allows for many applications to be developed with the framework as a base. This flexibility allowed FIRM to be evaluated during its construction in a legacy application as well as a newer implementation. Each system was modeled very differently, incorporating unique sets of use cases. At the same time, as decision support systems in the agricultural domain, the use cases for each system shared many common elements.

DEVELOPER SUPPORT

One of the key framework design goals was to enable developers to easily access framework components. Starting with the development of the integrated component runtime through the definition

of the object model, elements of FIRM were designed to hide as much of the framework as possible from the developer. This allowed the components that s/he would write to interact with the data in a domain-center way. The component context created by the core foundation services allowed client-code written by non-FIRM developers to easily interact with framework components as if they were standard Java objects. The fact that a large distributed framework was used to generate component call returns was abstracted completely from the developer. This abstraction not only increased the scope of development beyond GreenLeaf, but also lowered the learning curve associated with using the framework.

The strongly typed data model also supported the integration of domain components and the development of application components by serving as a set of semantics to describe application functionality and data consumption. It was this fact that originally drove the development of the strongly type data model, not the resulting abstraction layer between component and application models. With such a data model, software developers were able to more easily reason about the data provided by the components and are therefore able to build more robust use case driven systems rather than having to consider data integration as a first class development driver. Another benefit realized by the data model in regard to software development was the creation of standards within the software, allowing a greater level of interaction between software components without the need for complex connector logic. This resulted in more stable and streamlined software.

The developer support aspect of FIRM contributed greatly to the development of GreenLeaf as an application model, allowing that model to evolve with software requirements and technologies. Where NADSS was statically bound to a single framework with a number of technologies, GreenLeaf communicated with FIRM through the framework context provided by the core foundation services. Communication with the underlying framework relied solely on the use of standard Java conventions, allowing the GreenLeaf application model to be easily implemented in several paradigms. Deployed as a web application, GreenLeaf incorporated aspects of Java EE, JBoss Rich Faces, and Java Server Faces. However, the application model was also successfully implemented in Flex 3 utilizing BlazeDS to communicate with underlying Java components. Two very different application development paradigms were used to implement the same application model definition. Contrasted with NADSS, which required

tight coupling with J2EE and CORBA in order to effectively communicate with 3Co in the web application paradigm, GreenLeaf was able to easily transition from a standard web application to a distributed collection of multiple applications.

Beyond the use of the framework to support the development of decision support systems, FIRM was able to support the development of research components for incorporation into each application model. Working with researchers at the University of Nebraska at Kearney (UNK), the 3Co and FIRM development teams provided developer support for the creation of data mining components based on oceanic and drought indices. The effort of integrating the functionality with 3Co required that developers run an entire instance of the framework locally at UNK in order to implement and then test the connectors necessary to tie the component functionality together with the data mining algorithms. This process required that developers become familiar with all aspects of the 3Co framework. With the release of FIRM the data mining effort was reset to take advantage of the new framework. In development against FIRM, the research team at UNK was free to implement the data mining components in a Java framework of their choosing, they were no longer required to run the framework locally nor mirror its component organization in order to build the data mining components. Significantly less interaction between the development teams was required.

CONCLUSION

The ability to easily incorporate new analysis methodologies at the application or component model demonstrates one of the major strengths of using FIRM as the supporting framework for both NADSS and GreenLeaf. Throughout the GreenLeaf development process, tool development revolved around the concept of integrated data rather than encapsulation of a methodology. This more closely matches the general decision support practices that have emerged in modern decision support tools; namely, the process of providing seemingly unrelated data in ways that give the user a unique, and likely informative, view of the domain in which a decision is being considered. The tools contained in GreenLeaf are more closely aligned with the process of information discovery rather than explicit decision support based on 'expert' modeled workflows. The line between tools in GreenLeaf becomes somewhat blurred,

with the concept of tool transitioning towards a task that leads to another task and in turn leads to another. This view of decision support, as a discovery path, is supported directly by the abstractions of FIRM, mapping directly to the concept of a unified but loosely coupled application space. As a reference implementation, GreenLeaf demonstrates the ability of FIRM to easily support development activities within the given domain.

The use of FIRM as a basis for NADSS introduced a number of improvements to the application allowing it to outlive its originally intended lifespan. The introduction of the centralized management framework allowed NADSS to be more easily configured to adjust for changes in deployment and user load without requiring new deployments or builds; significant improvements to the process of running data builds were realized through the introduction for FIRM system support components. NADSS also benefited from increased stability due to the ability to dynamically distribute FIRM component services in multiple production environments. In the previous, non-FIRM, versions of NADSS, a component failure generally resulted in the loss of the entire component model runtime. With the introduction of FIRM, component stability improved dramatically, and system availability was able to be maintained in the event of a single component failure.

The GreenLeaf Project remains in production today as a replacement for NADSS, the application model has been used successfully to incorporate several data sets that had not been identified at the initial time of development, these data sets have been incorporated into the FIRM component model and integrated in the GreenLeaf application model. The decoupling of the models supported through the data model of FIRM has also allowed several tools to be created based on data integration alone, and required no modification to the underlying component model. The GreenLeaf Project replaced NADSS as of May 2008 and is scheduled to undergo a 2.0 release in November of 2009. The 2.0 release will continue the process of application model evolution and will incorporate a number of tools that run external to the web-based model. This tool suite continues to be supported by FIRM.

CHAPTER 7: CONTRIBUTION AND FUTURE WORK

This work contributes a framework architecture to support the development of data-driven decision support systems and other data-driven applications. The Framework for Integrated Risk Management (FIRM) is an implementation of the framework architectural concepts presented in this thesis. FIRM provides an example implementation of a component model supported by core foundation services to introduce a framework context and allow the development of component logic independent of framework constraints. This independence allows for the development of a strongly typed domain data model, facilitating loose coupling between the component and application models and allowing multiple application models to be developed against a single framework instance.

These capabilities are certainly not unique to FIRM. Many frameworks exist that support software development through various abstractions and conventions. The goal of this work is not to present a competing framework but to evaluate an architectural methodology for addressing the needs of data integration through decoupling of internal components. This methodology can be applied to any framework designed around data provision. It is particularly applicable to the decision support system class of applications due to the data-driven nature of the use cases for such systems. Its applicability, however, is not limited to that domain. From the standpoint of data integration, FIRM provides a verification of the architecture in its support of two complementary decision support systems.

FUTURE WORK

To date, application development with FIRM has been limited to the single domain of agricultural decision support, and within that domain to complementary systems. Several interesting problems emerge when considering the integration of data across domains or within a very large and partitioned domain. To achieve integration of such data within the approach presented here,the data model must be updated, introducing new connection points for components but requiring integration at both the component and application models. This does not present a significant challenge; it does, however, require the use of ‘standard’ integration approaches for the initial inclusion of the new data that once

integrated with the framework can be used 'transparently' by framework components and applications. As new data sets that result in changes to the data model are introduced, the model itself becomes complex and may introduce internal incompatibilities between components. To some extent, this can be seen with the inclusion of very specific data model definitions for the drought report structure of FIRM. As the data model grows, inconsistencies are likely to increase, eventually creating the need for layering within the data model. The result becomes an application model not unlike NADSS that has an abstraction for dealing with layers in the model between it and the component model.

This problem is magnified when considering the integration of data sets from two disparate decision domains. While achieving a great deal of success in providing data in the agricultural decision domain, FIRM has been considerably less successful at incorporating elements to support projects in the humanities that required access to climate data. The main issue that is presented with this integration is the inclusion of very orthogonal data model elements. These do not fit well in the original design of the data model. In order to integrate these data, the data model could be modified to introduce new elements, requiring change at the component and application levels. The development of a separate data framework for the data with integration points provided by a unique application model can address the problem, however preventing that kind of development for each data set was the stated goal of the framework.

Addressing this issue, namely the integration of data across decision domains or within large and partitioned domains is the goal of future work. That work will investigate the use of semantic web technologies such as ontologies to define a meta-language for describing data models. With such a meta-language, one could envision abstractions internal to the data model being defined that provide rules for data polymorphism that could be used to transition data definitions form one model to another. Various classes of integration components could be created and incorporated into core foundation services and application models that enable data discovery and integration. Significant work in data modeling, semantic web, and domain modeling are fourth coming to determine how best to address the larger-scale integration issues that emerge as a result of this work.

BIBLIOGRAPHY

[1] Lee, Daniel T., "Decision support in a distributed environment", Proceedings of the July 9-12 1984 national computer conference and exposition, (477 - 488), 1984.

[2] Orman, L., "A multilevel design for decision support systems", *ACM SIGMIS Database,* Volume 15, Issue 3, pp. 3 - 10, 1984.

[3] Lepreux, S., "Design method for a decision-support system centered on the decider and using component based programming", *Proceedings of the 15th French speaking conference on human-computer interaction,"* pp. 295 - 298, 2003.

[4] Zhang, S., Goddard, S. "A Software Architecture and Framework for Web-based Distributed Decision Support Systems", *Journal of Decision Support Systems.* 43.1 (2007): 1133 – 1150

[5] Burbeck, S.The Tao of e-business services. http://www.ibm.com/developerworks/webservices/library/ws-tao/

[6] Alur, D., Crupi, J., Malks, D., "Core J2EE Patterns, Best Practices and Design Strategies", Prentice Hall PTR; 2 edition (May, 2003). ISBN: 978-0131422469.

[7] JSR 255: Java Management Extension (JMX 2.0) Specification. http://www.jcp.org/en/jsr/detail?id=255

[8] JSR 244: Java Platform Enterprise Edition 5 (Java EE 5) Specification. http://jcp.org/en/jsr/detail?id=244

[9] Dan, A., Johnson, R., Arsanjani, A. Information as a Service: Modeling and Realization. *Proceedings of the International Workshop on Systems Development in SOA Environments*, (2007), 2.

[10] Papzoglou, M., Heuvel, W. "Service Oriented Architectures: approaches, technologies and research issues.",*The VLDB Jounral - The Information Journal on Very Large Databases,* 16,3 pp. 389 - 400, 2007.

[11] JSR 220: Enterprise Java Beans (EJB 3) Specification. http://jcp.org/en/jsr/detail?id=220

[12] Wu, X. "Interchangeable GIS components in spatial decision support systems", *Master's Thesis,* (August 2004).

[13] JavaBeans Specification, Version 1.01 http://cds-esd.sun.com/ESD4/JSCDL/javabeans/1.01/beans.101.pdf

[14] F. Newhall and C.R. Berdanier, "Calculation of Soil Moisture Regimes from the Climatic Record", *Soil Survey Investigations Report No. 46,* National Soil Survey Center, Natural Resources Conservation Service, Lincoln, NE, 1996.

[15] W.C. Palmer, “Meteorological Drought”, *Research Paper No. 45,* US Department of Commerce Weather Bureau, Washington D.C., 1965

[16] Wells, N., Goddard, S., Hayes, M. “A Self-Calibrating Palmer Drought Severity Index.” *Journal of Climate* 17.7 (2004): 2335 – 2351

[17] Keetch, J., Byram, G. "A drought index for forest fire control", *Res. Paper SE-38*. Asheville, NC: U.S. Department of Agriculture, Forest Service, Southeastern Forest Experiment Station. 33 pp.

[18] McKee, T.B., Doesken, N.J., Kleist, J., " The Relationship of Drought Frequency and Duration to Time Scales," *Proc. of 8th Conference on Appied Climatology,* pp. 179-184, 1993.

[19] Cottingham, I., Zhang, S., “The Calyx Specification”, *National Agricultural Decision Support System Development Documents,* 2004.

[20] Cottingham, I.J., Goddard, S., Zhang, S., Wu, X., Lu, K., Rutledge, A., and Waltman, W., "Demonstration of the National Agriculture Decision Support system", Proceedings of the 2004 National Conference for Digital Government Research, Seattle, WA. pp. 303-305, 2004.

[21] Jazayeri, M., "Some Trends in Web Application Development", *International Conference on Software Engineering, 2007 Future of Software Engineering*, pp. 199 - 213, 2007.

[22] Zdun, U., Hentrich, C., Dustdar, S. Modeling process-driven and service-oriented architectures using patterns and pattern primitives. *ACM Transactions on the Web, 1,3* (2007).

[23] P. Oriezy, P., Medvidovic, N., Taylor, R.N., Roseblum, D.S., "Software Architecture and Component Technologies: Bridging the Gap", *Proc of Workshop on Compositional Software Architectures,* Jan. 1998.

[24] JBoss Seam Framework, http://seamframework.org/Home

[25] JBoss AOP Reference Guide, http://jboss.org/jbossaop/docs/2.0.0.GA/docs/aspect-framework/reference/en/html/index.html

APPENDIX A

This appendix contains the Java source code for an example program that is a client of FIRM. The program makes use of many of the elements discussed in this thesis to provide a printout of climate and drought information for a zip code. When executed, the program will query weather stations within some miles of a given zip code and filter them to only those stations reporting 30 years of rainfall data at 90%. It will then query rainfall data, compute an SPI, obtain drought impact reports and display all of the station data to the screen in a report format. While many of the software applications built on FIRM incorporate considerably more complex use cases, this simple example demonstrates the basic functionality of FIRM that enables more complex examples.

A1 – SAMPLE FIRM CLIENT PROGRAM

```
package examples;

import java.text.NumberFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import org.joda.time.DateTime;
import org.joda.time.chrono.GregorianChronology;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

import edu.unl.firm.climate.ClimateDataQuery;
import edu.unl.firm.climate.ClimateMetaDataQuery;
import edu.unl.firm.climate.ClimateServiceAccessor;
import edu.unl.firm.climate.MetaDataCollection;
import edu.unl.firm.climate.MetaDataType;
import edu.unl.firm.climate.VariableFilter;
import edu.unl.firm.climate.VariableMetaData;
import edu.unl.firm.drought.DroughtImpactQuery;
import edu.unl.firm.drought.DroughtIndexQuery;
import edu.unl.firm.drought.DroughtServiceAccessor;
import edu.unl.firm.drought.ImpactQueryResult;
import edu.unl.firm.drought.ImpactReportBean;
import edu.unl.firm.drought.MediaReportBean;
import edu.unl.firm.drought.ShadowImpact;
import edu.unl.firm.shared.CalendarDataCollection;
import edu.unl.firm.shared.CalendarPeriod;
import edu.unl.firm.shared.DataType;
import edu.unl.firm.shared.TemporalPeriod;
import edu.unl.firm.terra.SpatialQuery;
import edu.unl.firm.terra.TerraServiceAccessor;
import edu.unl.firm.terra.USCity;

publicclass FirmExample {

        privatestaticfinal DateTimeFormatter D_FORMATTER = DateTimeFormat.mediumDate();
        privatestaticfinal NumberFormat N_FORMATTER = NumberFormat.getPercentInstance();

        publicstaticvoid main(String[] args) {

                try {
                        long start = System.currentTimeMillis();

                        SpatialQuery spatial_query = GeographicServiceAccessor.
```

```
        getInstance().getSpatialQuery();

DroughtImpactQuery drought_impact_query = DroughtServiceAccessor.
        getInstance().getDroughtImpactQuery();

DroughtIndexQuery drought_index_query = DroughtServiceAccessor.
        getInstance().getDroughtIndexQuery();

ClimateDataQuery climate_query = ClimateServiceAccessor.
        getInstance().getClimateDataQuery();

ClimateMetaDataQuery meta_query = ClimateServiceAccessor.
        getInstance().getClimateMetaDataQuery();

List<String> stations = spatial_query.getStationsByZipCode("68521", 10);

USCity city = spatial_query.getCityByZip("68521");

TemporalPeriod period = newTemporalPeriod(
                new DateTime(1980,1,1,0,0,0,0,
                                GregorianChronology.getInstance()),
                new DateTime(2009,6,30,0,0,0,0,
                                GregorianChronology.getInstance()));

VariableFilter filter = newVariableFilter();
filter.setVariableType(DataType.PRECIP);
filter.setMissingTolerance(10f);
filter.setValidPeriod(period);

List<VariableFilter> filters = new ArrayList<VariableFilter>();
filters.add(filter);

System.out.printf("\nTotal stations in zip code: %d\n", stations.size());

stations = meta_query.filterStations(stations, filters, period, 90f, false);

System.out.printf("Total filtered stations in zip code: %d\n",
        stations.size());

MetaDataCollection<MetaDataType> station_meta_data =
        meta_query.getAllMetaData(stations, CalendarPeriod.MONTHLY);

Map<String, Map<DataType, VariableMetaData>> variable_data =
        meta_query.getVariableMetaData(stations);

CalendarDataCollection precip_data = climate_query.getPeriodMonthlyData(
        stations,
        new DateTime(2009,1,1,0,0,0,0,GregorianChronology.getInstance()),
        new DateTime(System.currentTimeMillis()), DataType.PRECIP);


CalendarDataCollection spi_data = drought_index_query.
        computeContinuousMultiyearMonthlySpi(
                stations,
                1, newDateTime(2009,1,1,0,0,0,0,
                GregorianChronology.getInstance()),
                2009);

ImpactQueryResult impacts = drought_impact_query.queryCountyImpacts(
        city.getCounty(),
        new DateTime(2009,1,1,0,0,0,0,
                GregorianChronology.getInstance()),
        new DateTime(System.currentTimeMillis()));

for ( String station : station_meta_data ) {
        Map<MetaDataType, Object>meta = station_meta_data.
                getStationMetaData(station);
        Map<DataType, VariableMetaData> var = variable_data.get(station);

        System.out.printf("\n --- %s ---\n",
                meta.get(MetaDataType.STATION_NAME));

        System.out.printf("Network: %s\n",
                meta.get(MetaDataType.NETWORK_NAME));

        System.out.printf("Network ID: %s\n",
                meta.get(MetaDataType.NETWORK_ID));

        System.out.printf("Station Start Date: %s\n",
                D_FORMATTER.print((DateTime)meta.get(
                        MetaDataType.START_DATE)));
```

```
                System.out.printf("Station End Date: %s\n",
                        D_FORMATTER.print((DateTime)meta.get(
                                MetaDataType.END_DATE)));

                System.out.printf("Station Lat: %s\n",
                        meta.get(MetaDataType.LATITUDE));

                System.out.printf("Station Lon: %s\n",
                        meta.get(MetaDataType.LONGITUDE));

                System.out.printf("Station Elevation: %s\n",
                        meta.get(MetaDataType.ELEVATION));

                System.out.printf("\n --- Station Variables ---\n");

                for ( DataType type : var.keySet() ) {
                        VariableMetaData var_meta = var.get(type);
                        System.out.printf("\tVariable: %s\n ", type.getName());
                        System.out.printf("\tStart Date: %s\n",
                                D_FORMATTER.print(var_meta.getStartDate()));

                        System.out.printf("\tEnd Date: %s\n",
                                        D_FORMATTER.print(var_meta.getEndDate()));

                        System.out.printf("\tMissing Percent: %s\n\n",
                                N_FORMATTER.format(var_meta.getMissingPercent()));
                }

                System.out.printf("\n --- 2009 Monthly Rainfall ---\n");

                System.out.printf("\tJ\t\tF\t\tM\t\tA\t\tM\t\tJ\n");

                for ( float val : precip_data.getDataMatrix(station)[0]) {
                        if ( val != DataType.OUTSIDE_OF_RANGE ) {
                                System.out.printf("\t%fin.", val);
                        }
                }

                System.out.printf("\n\n--- 2009 Monthly SPI ---\n");

                System.out.printf("\tJ\t\tF\t\tM\t\tA\t\tM\t\tJ\n");

                for ( float val : spi_data.getDataMatrix(station)[0]) {
                        if ( val != DataType.OUTSIDE_OF_RANGE ) {
                                System.out.printf("\t%f ", val);
                        }
                }
                System.out.printf("\n");
        }

        System.out.printf("\n\n --- Impacts for the city of %s in %s County, "+
                "%s ---\n",
                city.getName(), city.getCounty().getName(),
                city.getCounty().getState().name());

        System.out.printf("\tTotal Impacts: %d", impacts.getImpactList().size());

        for ( ShadowImpact impact : impacts.getImpactList() ) {

                ImpactReportBean the_impact =
                        drought_impact_query.loadReport(impact.getId(), true);
                System.out.printf("\n\tEffect Beginning: %s",
                        D_FORMATTER.print(impact.getStart()));

                System.out.printf("\n\tEffect Ending: %s",
                        D_FORMATTER.print(impact.getEnd()));

                System.out.printf("\n\tEntered: %s",
                        D_FORMATTER.print(impact.getEntryDate()));

                if ( the_impact.getMediaReports().size() > 0 ) {
                        MediaReportBean media =
                                the_impact.getMediaReports().get(0);

                        System.out.printf("\n\tMedia Report Content:\n");
                        System.out.printf("\t%s", media.getFullText());
                        System.out.printf("\n\tStory Reference: %s",
                                media.getStoryReference());

                } elseif ( the_impact.getUserReports().size() > 0 ) {
```

```
                                        System.out.printf("\n\tWe do not have permission to "+
                                                "print user report data.");
                                }
                        }


                        System.out.printf("\n\n\n Report runtime: %d seconds",
                                (System.currentTimeMillis() - start)/1000);
                } catch ( Exception e ) {
                        e.printStackTrace(System.err);
                        System.err.printf("Execution halted!\n");
                }
        }
}
```

A2 – CLIENT PROGRAM OUTPUT

```
Container created for token: FIRM_BASE
[FIRM (codename Thresher) version 1.0.20080618.1-R]

Total stations in zip code: 23
Total filtered stations in zip code: 4

 --- LINCOLN MUNI AP ---
Network: COOP
Network ID: 254795
Station Start Date: Jan 1, 1948
Station End Date: Jun 30, 2009
Station Lat: 40.831
Station Lon: -96.764
Station Elevation: 1170.0

 --- Station Variables ---
        Variable: Precipitation
        Start Date: Jan 1, 1948
        End Date: Jul 5, 2009
        Missing Percent: 29%

        Variable: High Temperature
        Start Date: Jan 1, 1948
        End Date: Jul 5, 2009
        Missing Percent: 29%

        Variable: Average Relative Humidity
        Start Date: Dec 31, 2003
        End Date: Jul 5, 2009
        Missing Percent: 0%

        Variable: Average Wind Speed
        Start Date: Dec 31, 2003
        End Date: Jul 5, 2009
        Missing Percent: 0%

        Variable: Average Temperature
        Start Date: Jan 1, 1948
        End Date: Jul 5, 2009
        Missing Percent: 29%

        Variable: Low Temperature
        Start Date: Jan 1, 1948
        End Date: Jul 5, 2009
        Missing Percent: 29%


 --- 2009 Monthly Rainfall ---
J               F               M               A               M               J
0.380000in.     0.640000in.     0.180000in.     1.520000in.     1.170000in.     6.180000in.

--- 2009 Monthly SPI ---
J               F               M               A               M               J
-0.580000       -0.080000       -1.910000       -0.880000       -1.920000       0.980000

 --- RAYMOND 2NE ---
Network: COOP
Network ID: 257055
Station Start Date: Aug 2, 1942
Station End Date: Jun 30, 2009
Station Lat: 40.974
Station Lon: -96.766
Station Elevation: 1320.0
```

```
 --- Station Variables ---
        Variable: Precipitation
        Start Date: Aug 2, 1942
        End Date: Jul 6, 2009
        Missing Percent: 7%

        Variable: High Temperature
        Start Date: Jan 1, 2000
        End Date: Jul 6, 2009
        Missing Percent: 4%

        Variable: Average Temperature
        Start Date: Jan 1, 2000
        End Date: Jul 6, 2009
        Missing Percent: 4%

        Variable: Low Temperature
        Start Date: Jan 1, 2000
        End Date: Jul 6, 2009
        Missing Percent: 4%


 --- 2009 Monthly Rainfall ---
J               F               M               A               M               J
0.320000in.     0.750000in.     0.160000in.     1.310000in.     1.670000in.     6.130000in.

--- 2009 Monthly SPI ---
J               F               M               A               M               J
-0.370000       0.200000        -1.840000       -0.820000       -1.500000       0.790000

 --- MALCOLM ---
Network: COOP
Network ID: 255105
Station Start Date: Aug 1, 1942
Station End Date: Jun 30, 2009
Station Lat: 40.908
Station Lon: -96.865
Station Elevation: 1310.0

 --- Station Variables ---
        Variable: Precipitation
        Start Date: Aug 1, 1942
        End Date: Jul 6, 2009
        Missing Percent: 3%


 --- 2009 Monthly Rainfall ---
J               F               M               A               M               J
0.420000in.     0.570000in.     0.110000in.     1.340000in.     1.100000in.     5.760000in.

--- 2009 Monthly SPI ---
J               F               M               A               M               J
-0.350000       -0.120000       -1.990000       -0.970000       -2.060000       0.710000

 --- LINCOLN UNIV PWR PLT ---
Network: COOP
Network ID: 254815
Station Start Date: Sep 1, 1955
Station End Date: Mar 31, 2009
Station Lat: 40.823
Station Lon: -96.702
Station Elevation: 1160.0

 --- Station Variables ---
        Variable: Precipitation
        Start Date: Sep 1, 1955
        End Date: Apr 1, 2009
        Missing Percent: 16%

        Variable: High Temperature
        Start Date: Sep 1, 1955
        End Date: Aug 1, 1998
        Missing Percent: 4%

        Variable: Average Temperature
        Start Date: Sep 1, 1955
        End Date: Aug 1, 1998
        Missing Percent: 4%

        Variable: Low Temperature
```

Start Date: Sep 1, 1955
End Date: Aug 1, 1998
Missing Percent: 3%

--- 2009 Monthly Rainfall ---

J	F	M	A	M	J
0.370000in.	0.630000in.	0.180000in.			

--- 2009 Monthly SPI ---

J	F	M	A	M	J
-0.420000	0.010000	-1.500000			

--- Impacts for the city of Lincoln in Lancaster County, Nebraska ---
Total Impacts: 1
Effect Beginning: May 9, 2009
Effect Ending: May 9, 2009
Entered: May 11, 2009

Media Report Content:
The mayor of Lincoln has asked residents to voluntarily conserve water because other portions of the state that have some bearing on Lincoln's water supply haven't gotten enough rain.

Story Reference:
http://www.journalstar.com/articles/2009/05/09/news/local/doc4a03576544749673673380.txt

Report runtime: 33 seconds