Annals of Software Engineering 13, 381–411, 2002

A Framework for Community Information Systems

PAULO S. C. ALENCAR and DONALD D. COWAN {palencar; dcowan}@csg.uwaterloo.ca
Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

MARTIN LUO mluo@microsoft.com
Microsoft Corporation, Commerce Server Product Group, One Microsoft Way, Redmond, WA 98052, USA

Abstract. In this paper we present a generic framework architecture for Web-based community information systems (CIS). The framework has an open architecture based on COTS (commercial-off-the-shelf) software components and network technologies. We discuss how a component-based approach, a layered architecture model, and design patterns can be used to provide a common framework for CIS. The CIS framework architecture results in significant benefits that include reuse, a flexible user interface, powerful search mechanisms and an integrated and scalable architecture. XML and rule-based StyleSheet languages are used for storage, information search and graphical presentation at the server or client. The overall framework architecture, its individual components and the interaction among these components are outlined.

Keywords: Community information systems, frameworks, design patterns, XML, reuse, extensibility, separation of concerns, COTS

1. Introduction

A community information system (CIS) can be viewed as a huge repository of valuable information, which needs to be organized, maintained, augmented, searched and presented for the benefit of a broad client base [Cowan 2000]. With the popularity of the World Wide Web, the Internet has been adopted as the primary method of providing access to a CIS, where access points can be through a variety of devices including personal computers, network computers, and kiosks as well as pervasive devices, such as Personal Digital Aids (PDA), smart-cards, digital wireless telephones.

The core of any geographic community information system is local-based content and online services such as local news, community events, government, business and tourist information, and e-commerce transactions. This scattered information presented on the Web is provided and maintained by individuals, organizations and businesses within the community. A closer look at existing community-oriented systems, both large and small, reveals two broad design approaches. A community can:

- maintain a set of customized web pages for each individual entity within the community; or
- index, store and maintain a repository containing catalogue information for each entity. When an entity's page(s) is accessed and requested, the corresponding web pages are generated dynamically.

For a large community with hundreds of entities, only the latter approach is feasible. Since most information in a community is geo-referenced, hypermaps [Alencar *et al.* 1997, 1999], in which map elements are related to different types of multimedia information, such as text, pictures, and sound, can also be used as search engine index for retrieval.

CIS services cover a broad range from locating community-specific information or notification of community events to online services, such as paying utility bills, reserving tennis courts or obtaining parking permits. Because the CIS interacts with so many processes in the community, a CIS solution must incorporate a broad range of core services or functions. Furthermore, those functions must be built on a scalable, extensible platform.

Many current web-based applications use a three-layer (presentation, business logic, system layer) approach, which promotes strong coupling among individual components that address the basic or special purpose concerns in the system. For example, in the presentation layer, one often finds navigation, layout and user interface component information. Thus, separation of concerns is often not maintained, creating difficulties in identifying and assigning functionality and thus acting as a detriment to both reuse and extensibility.

1.1. A framework-based approach

To avoid strong coupling and facilitate the reuse of a collection of proven design patterns as well as implemented components, we have designed an object-oriented framework [Alencar *et al.* 2000; Fayad *et al.* 1999a, b; Fontoura 1999; Knudsen 2001; Larman 1999] for the domain of community information systems that can be used as the core of similar systems. The objective of this paper was to present a framework architecture [Kruchten 1999; Perry and Wolf 1992] that defines an overall system structure and component environment [Szyperski 1997], develops reusable components, and build a CIS application framework to be used with different applications. The CIS framework has a layered architecture and uses the Extensible Markup Language (XML), the Extensible Stylesheet Language (XSL) and the Vector Markup Language (VML) to separate presentation of information from content [German *et al.* 1998; W3C 2001].

The primary contributions of this paper are to develop a sound application framework for a community information system. The work involved in attaining this goal has produced the following visible contributions:

- identify and specify the abstract functional and non-functional requirements of community information systems;
- develop an architecture model for community information systems;
- develop customized design patterns, such as "XML-based document view" and "broker", to cope with various design issues;
- develop and demonstrate a technique for extending the framework to construct actual CIS applications.

1.2. Paper outline

Section 2 describes the scope of the CIS framework, illustrates its context, and describes the CIS core object services, along with the well-defined domain-specific services packaged as reusable components. Section 3 presents the details of the proposed architecture design using the conventional object-oriented paradigm and design patterns. In section 4, two case studies are used to illustrate how to extend the CIS framework and use the framework to construct CIS applications. Finally, section 5 summarizes the results, shows the lessons we learned from the framework development, and discusses possible future work.

2. Scope and domain of CIS framework

In this section, we illustrate the CIS context, its framework, core services, and domain-specific services. Finally, we also present some of the design goals of the CIS framework such as software design reuse, interoperability, flexibility and extensibility.

2.1. Scope of CIS framework

The overall CIS framework architecture can be viewed as a set of layers consisting of components with the same degree of generality. A CIS application framework manages the communications between requesters of services (clients) and providers of services (servers). Figure 1 shows the 3-layer architecture.

- The Infrastructure layer provides standardized software services to upper layers through common interfaces;
- the Domain-Specific Component layer represents the business logic in the CIS domain.
- The CIS Application layer is at the top of the hierarchy. The CIS applications are solutions that are extended from the CIS framework and developed based on specific domain requirements and delivered to end users. An application can be built either as an instance of an application framework, or as a composition of the services defined by the CIS framework. In the latter case, an application will be built by instantiating and customizing the components in the lower layers, as well as implementing and integrating custom components.

2.2. Behavior specification

A CIS framework offers a set of high-level services to both application developers and end users. Some services are CIS-unique (for example, Message Transportation service), while some are common database [Ullman 1988] management system services like transaction management, or operating system services, such as security, network, file, and print services.

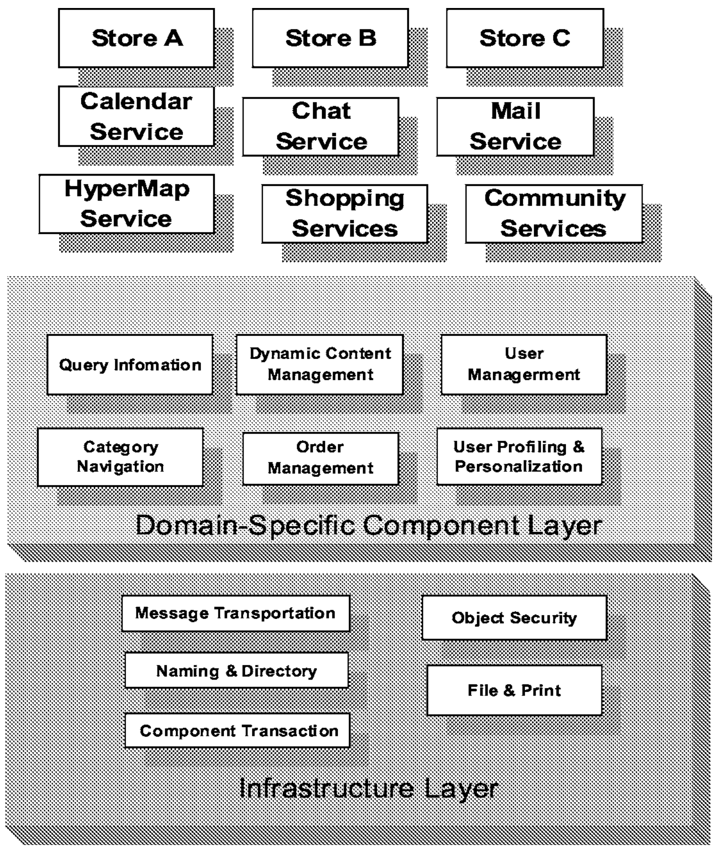

Figure 1. CIS application framework.

2.2.1. Infrastructure support and core services

The CIS framework depends upon a set of core services that are provided by the underlying operating system or database management system.

The main *core services* are:

- Component Transaction services.
- Naming and Directory service for objects.
- Message Transportation service – an event-based messaging mechanism for event propagation and notification.
- Object Security service – supports a security policy between collaborating units.
- File and Print service.

2.2.2. CIS domain-specific components

The domain-specific components are:

- Query information – permits the querying of application objects, which are generally grouped into sets or collections.

- Category Navigation – provides a means for the web users to navigate through the various web pages within the CIS web site.
- User Management – supports storage and querying of all the user-related information:
  - User Authorization – uses the User ID/Password pair to grant permission for specific services;
  - Request Validation – validates a service transaction.
- Dynamic Content Management – makes site navigation easier through techniques such as site maps and local search engines.
- User Profiling and Personalization.
- Order Management – saves state of user interactions.

2.3. Design objectives of CIS framework

Special care must be taken to build a sound architecture for the framework that offers performance, flexibility, scalability, and maintainability. In particular, an effective technology platform should have the following characteristics:

- Reusable – it should facilitate the reuse of the growing collection of proven design patterns as well as implemented domain components.
- Separation of concerns – it should separate basic concerns from each other and from special-purpose concerns.
- Open and standard-based – since requirements for new functionality will arise the CIS framework should therefore have an architecture based on open standards in areas involving applications, data and security:
  - application interoperability: CORBA IIOP, Java RMI, EJB and COM/DCOM are industry standards for component development that could be adopted;
  - data exchangeability: XML, XSL, VSL and WML (Wireless Markup Language) are evolving industry standards for data exchange;
  - security: Secure Electronic Transactions protocol, Secure Socket layer and digital signature and certificates.

Adopting these open standards within the CIS framework should help to ensure that the CIS platform will be extensible, scalable and adaptable, and evolve as business changes.

3. CIS application framework architecture

3.1. An overview of the framework architecture

In its most basic form, the CIS framework as illustrated in figure 2 can be viewed as a logical three-*layer* model, which is represented in UML[1] [Eriksson and Penker 1998].

[1] In this paper, the following UML diagrams [Eriksson and Penker 1998; Larman 1999] are used: class diagram, sequence diagram and collaboration diagram. UML is derived from and unifies the notations

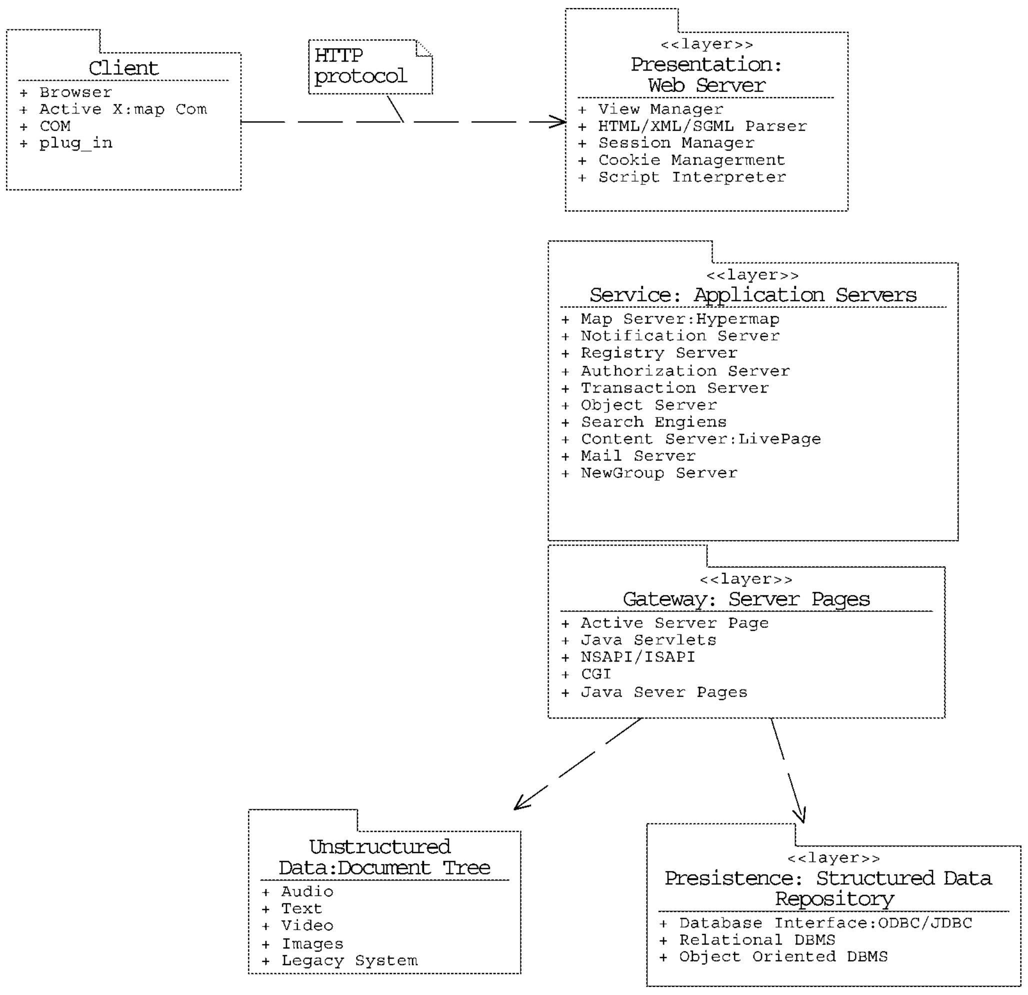

Figure 2. The CIS three-layer model.

The layers are:

- *Presentation layer* containing the session manager, HTML/XML parser and script interpreter.
- *Business logic layer* containing the domain specific business logic and processes with service components that include Transaction Server, Notification Server, and Content Server.
- *Data Repository layer* provides the data storage used by the other layers. The data can be further classified as unstructured data (such as audio, video and images) and

of at least three object-oriented analysis and design methodologies: Booch's diagrams [Booch 1994]; Rumbaugh's Object Modeling Technique (OMT) [Rumbaugh *et al.* 1991]; Jacobson's approach [Jacobson *et al.* 1992]. UML is an accepted standard of the Object Management Group (OMG) [OMG 2001].

structured data, where the structured data is usually stored in the database management system.

The application elements in these three logical tiers are connected through a set of standard protocols, services and software connectors. The server gateway connects the application components residing in separate layers through a set of standard interfaces, such as CGI (Common Gateway Interface), Java Servlet, NSAPI (Netscape API), ISAPI (Microsoft IIS API), and Active Server Pages. The connectors represent the interactions among components [Garlan and Shaw 1993]. Communications are both synchronous (database queries, RPC calls), or asynchronous (message queues, event broadcasts or pipes).

However, the three-layer architecture has the following limitations:

- The business logic and data repository layers are tightly coupled. Changing the operating system, or database management system may cause changes in the business logic layer and even the presentation layer.
- There is also strong coupling between the presentation and the business logic layer. If there is a change in either layer, then the parts of the system that referenced them must also be changed.

To address these limitations, we introduce design patterns, and propose an n-tier ($n > 3$) framework architecture.

The CIS framework architecture overview in figure 3 illustrates how the CIS framework is structured, but not why it is structured using particular components and patterns [Nova *et al.* 1998; Larman 1999]. The rationale behind the choice and use of specific design patterns is discussed in the next sections. The specific patterns shown in figure 3 and used in the framework are:

- XML-driven Document–View pattern.
- Protocol Pipeline – Pipe and Filter pattern.
- Integrate legacy system – Adapter pattern.
- Dynamic load balance – Client–Dispatcher–Server pattern.
- Security and access control – Proxy pattern.
- XML converter – Broker pattern.
- Unified Data interface – Façade pattern.
- Event Management – Publisher–Subscriber pattern.

3.2. XML-driven Document–View pattern

Design challenges

As mentioned in previous subsection, most current web applications use a three-layer approach, which has two limitations:

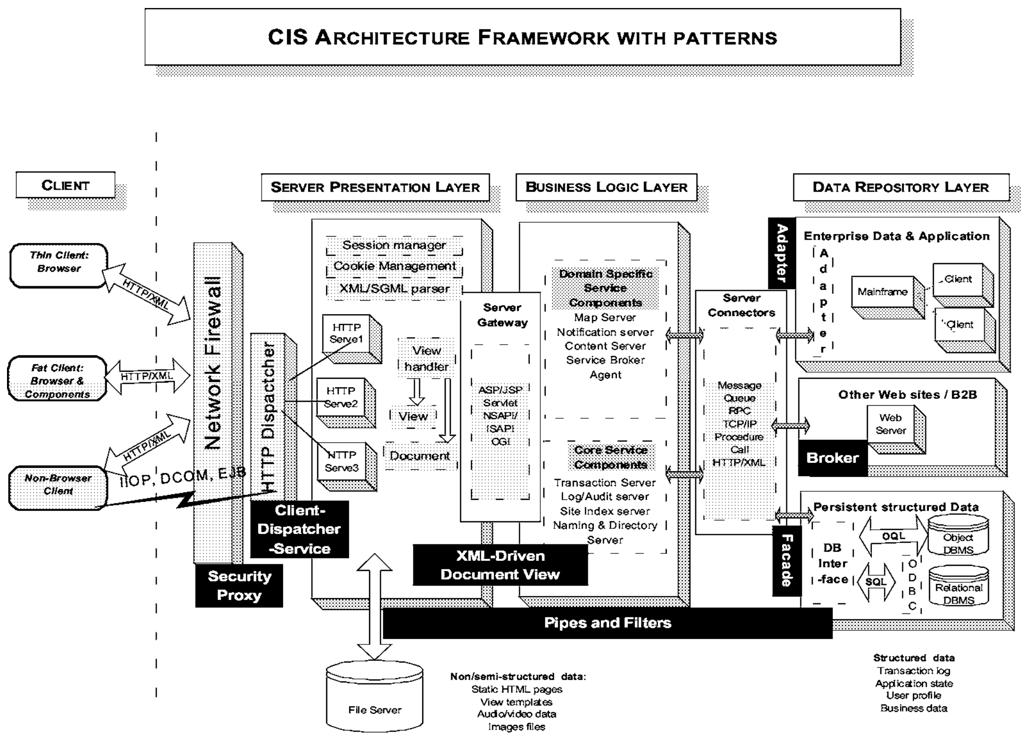

Figure 3. CIS architecture framework with patterns.

- Strong coupling between Presentation and Business Logic layer. If there is a change in either layer, then the parts of the system that referenced them must also be changed.
- Lack of customized views for different users. At the presentation layer, different users sharing the same data may have different needs and may want a different view of this data.

Solution – the Document–View pattern

Document–View
In object-oriented programs, the Model/View/Controller (MVC) is a popular pattern in which different parts of an application are separated into a model, a view and a controller. The Document–View pattern, however, as a variant of MVC, relaxes this separation of view and controller. Since in most cases the views are constructed for read-only purpose, a controller can ignore any user input. Such observation makes it possible to combine the responsibilities of a view and a controller into a single document.

XML-driven
The principle behind our “XML-driven” methodology is to develop mechanisms that provide a clean separation of the semantic knowledge of information and the presentation. The key concept here is to provide users with customized views. An XML doc-

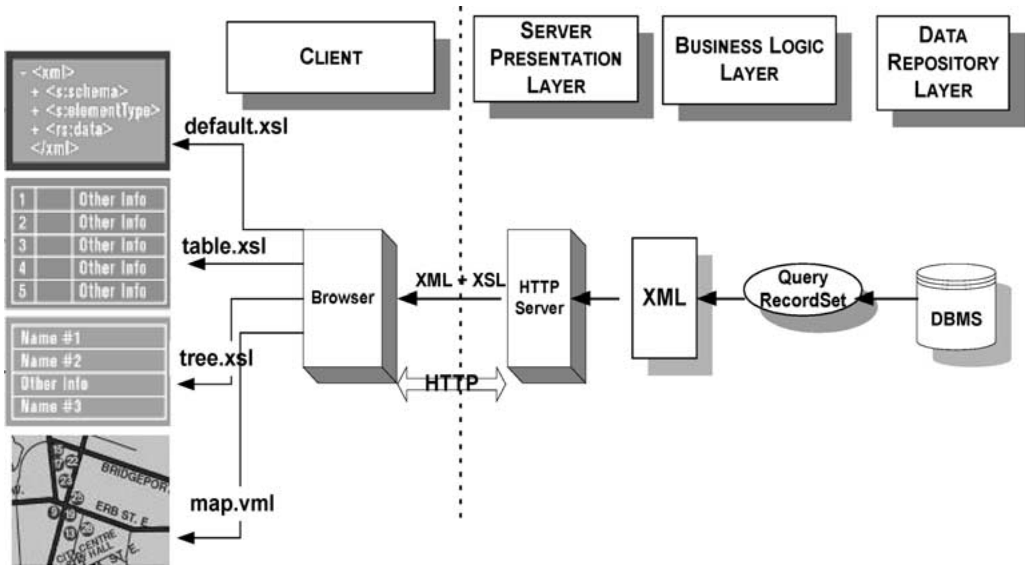

Figure 4. Separate content from presentation.

ument generally contains content and structure, not the presentation information. This presentation information is stored elsewhere using a system of structure-based presentation rules, such as XSL, CSL or VML.

XSL (Extensible StyleSheet Language) is an XML-based document that follows an XML document tree and transforms document nodes into other formats such as a viewable HTML page. It was introduced and specified by the World Wide Web Consortium (W3C) [W3C 2001] for the presentation and display of XML documents. Many XSL files can be associated with the same XML document to provide different representations. VML (Vector Markup Language) is also an XML-based language schema, similar to XSL that is used to present high-quality vector graphics on the Web. A draft specification for VML has been submitted to the World Wide Web Consortium.

Separating content from presentation with server-side and client-side XML
In this pattern, the source data is stored in XML format on both the server- and client-side (see figure 4). This has many advantages. First, since the communication media between server and client is XML, instead of an HTML file, the procedures of data transformation between XML and HTML are reduced. Second, the client now has the ability to manipulate the data in its native format. Using XSL/VSL, the client can manipulate the data and display it, without server intervention. In the end, the updated data could be stored locally or even sent back to the server. Finally, the stored source data and XSL/VML file can also be used as the cache for the client.

Implementation considerations
To implement this pattern, the source data can be stored in XML format and then be inserted in a text field of a relational database. Several fields should be extracted from

the XML document prior to insertion into the database and placed in separate columns for indexing and searching.

Consequences
Applying the XML and document-view approach, we get a new pattern that is very useful for web-based GUIs. This pattern provides three important features:

- Data can be manipulated in a language and platform-independent manner.
- Because documents specify nothing in the way of presentation, views can be modified or created without affecting the underlying data.
- It is possible to get an extensible way to provide different views of the same data.
- Since the XML data can be completely separated from presentation, web developers can assign the presentation functions to the client, freeing web servers to do more advanced processing.

3.3. Pipeline protocol – Pipe and Filter pattern

Design challenges
From the CIS service provider perspective, interaction with the web clients can be viewed as a task of processing data streams. For example, if a client such as a web browser issues a HTTP request for dynamic content, the backend CIS servers process the request, generate the content, transform the data into a specific format and send the data stream to the client. Implementing such a sequence of transformations is naturally done in stages and provides the advantage of supporting easy replacement or re-ordering of stages if requirements change.

Solution
Processing as just described, can be divided into several stages where each stage can be implemented as a filter component [Buschmann *et al.* 1996]. A filter is used to enrich, refine or transform its input data and pass the enriched data to the next available filter component. Filter components are the processing units of a pipeline. There are many situations when pipelines may be useful. For example, an advanced local search engine may construct data filters depending on the form of a query.

Implementation considerations
The first step in implementing a pipeline pattern is to define the XML data format to be passed along each pipe. Defining a uniform format results in the highest flexibility because it makes the recombination of filters easy. The pipe connection can be implemented as a synchronous communication model such as Remote Procedure Call, or an asynchronous model using a system service such as a message queue.

Consequences
The application of a pipeline pattern has a major benefit for reuse. The developer can create a new processing pipeline with the existing filter components, newly added components or the reordering of those filters. Reuse is further enhanced if the user implements each filter as an active component, which pulls its input from and pushes its output down the pipeline.

Applying the Pipe and Filter pattern imposes some liabilities as well. There is some data transformation overhead associated with the Pipe and Filter pattern. Error handling may be more difficult in some cases.

3.4. Integrate legacy system – Adapter pattern

Design challenges
Many existing or legacy applications are reused by being integrated into a web-based framework. Interfaces of existing systems don't usually match the one provided by the framework. Instead of discarding the legacy system completely and building a new one from scratch, we'd like to find a feasible way to reuse the legacy system as a starting point for modification and extension.

Systems evolve over time – new technologies emerge, new functionality is added and existing services are changed. Design for change is therefore a major concern when specifying the architecture of a software system. Changes should not affect the core functionality or key design abstractions, otherwise the system will be hard to maintain and expensive to adapt to a changing environment [Buschmann *et al.* 1996].

Solution
To address the problems of legacy systems and evolution, an intermediate component, Adapter, is introduced. It serves two purposes:

- To convert the interface of an existing class into another interface that the framework is expecting. Through encapsulation of the implementation details of legacy systems that were not developed to work as part of a framework, a wrapped/adapted component can coexist with the CIS framework [Gamma *et al.* 1995].
- To separate a minimal functional core from extended functionality and customer-specific parts. The adapter also serves as a socket for plugging in such extensions and coordinating their collaboration. To some extent, the Adapter pattern provides a plug and play software environment so that it allows the users of the framework to connect extensions and integrate them with the core services of the system.

Structure
By using an adapter to wrap the original interfaces, the legacy system can appear compatible to the framework. Any caller can then make requests of the existing implementation. The Adapter pattern can be structured either as a class adapter or as an object adapter relying on object composition [Gamma *et al.* 1995]. Figure 5 shows how this works using object composition. Here the adapter, also known as a wrapper, represents

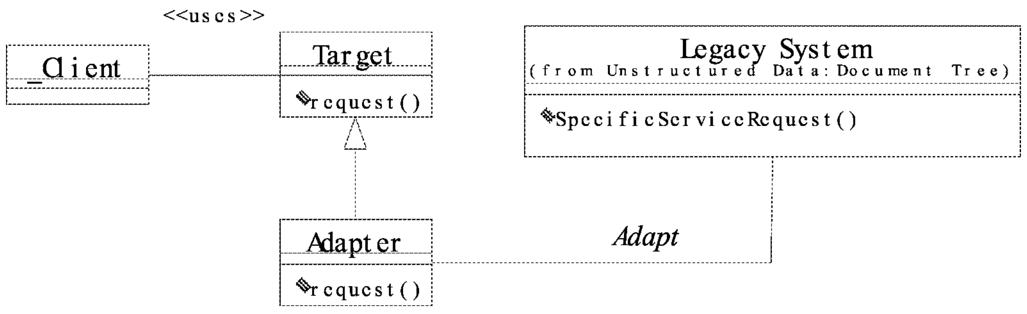

Figure 5. Class diagram: Adapter pattern.

the interface between clients and their service provider. It allows clients to access the services of the legacy system in a portable way [Gamma *et al.* 1995]. The adapter also protects clients from the specific implementation details and data format of the legacy systems.

The main classes of the Adapter pattern, as shown in figure 5, are:

- Target – represents the abstract interfaces which the clients expect.
- Client – represents an application, which collaborates with objects and conforms to the target interface.
- Adaptee – the legacy system provides the existing programming interfaces that needs adapting to be useable by its current clients.
- Adapter – adapts the interface of the adaptee to the target interface expected by the clients. The adapter can invoke the corresponding methods of the adaptee on behalf of the client, and it hides system dependencies such as specific communication protocols from the client.

Collaboration
The adapter acts as a binding between the client's interface and the legacy system implementation. When a client calls an operation on an adapter instance, the adapter generates and interprets the operation reference and maps the reference to the implementation of the legacy system. In turn, the adapter then calls legacy system operations that carry out the request. In some situations, the adapter can forward the requests for accessing the original legacy system interface to a delegate object.

Consequences
The adapter treats the existing legacy systems as a black box by encapsulating the original implementation details. This is much easier than rewriting the legacy system, and can be done without modifying a line of code of the legacy system. The adapter can also easily extend the legacy system by providing extra functionality. In this way, an adapter is like a decorator pattern [Gamma *et al.* 1995]. An adapter can provide several extra interfaces, not just the original object's interface.

3.5. Naming service and dynamic load balance – Client–Dispatcher–Service pattern

Design challenge
The CIS framework should scale with the growth of the new services and increasing number of client requests. To achieve this goal the framework should support simple integration of local or network distributed servers to support a large volume of client requests.

Solution
Those requirements are satisfied by the Client–Dispatcher–Service pattern, which is derived from the Client–Dispatcher–Server pattern [Buschmann *et al.* 1996]. The framework offers an intermediate dispatcher component between clients and servers. The primary purpose is to achieve dynamic load balancing and location transparency. The dispatcher uses a name service to provide location transparency by using names for service provider components instead of physical locations. The pattern can also hide the details of the establishment of the communication channel between clients and the eventual backend service provider components.

Structure
The main participating components as shown in figure 6 are:

Client – the client such as a web browser sends a request to a server, through a communications channel set up by the dispatcher to set up a communication channel.
Service – services are domain-specific functionality implemented by the server compo-

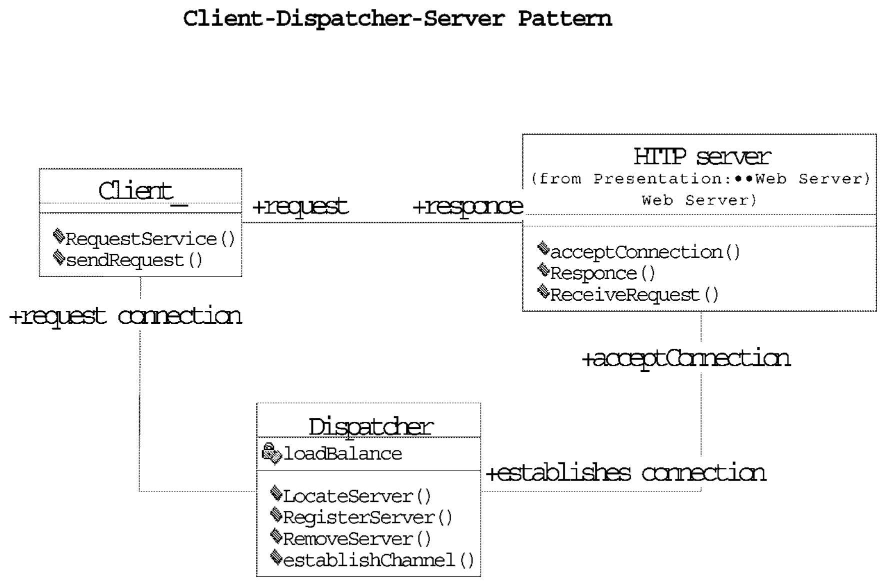

Figure 6. Class diagram: Client–Dispatcher–Service pattern.

nents such as a web server. They encapsulate resources and provide service interfaces to the clients.

Client.	The client such as a web browser sends a request to a server, through a communications channel set up by the dispatcher to set up a communication channel.
Service.	Services provide domain-specific functionality implemented by the server components such as a web server. They encapsulate resources and provide service interfaces to the clients.
Dispatcher.	Upon receiving a client service-request, a dispatcher uses its repository to determine which servers provide the specified service. A communication channel is then established to the lowest-loaded server. For location transparency, the dispatcher has to offer the functionality of a server location and name registration. Furthermore, the dispatcher can support fault tolerance by deactivating a failed service component and re-routing its load to other running processes.

Consequences
The Client–Dispatcher–Service pattern supports substitution of service components by separating the client, dispatcher and server components. Framework-compliant components will have well-defined specifications for each interconnection or dependency on other components. Any addition, modification or replacement of the service components will expect to work with minimal impact on the clients and dispatcher. Since the clients do not depend on physical location, the dispatcher can forward new client requests to the lowest-loaded service component. This allows the dispatcher to parcel out computing workload and dynamically adjust the system balance so that system can scale up gracefully with the growth of service types and increasing simultaneous users. The system is also easy to configure and migrate during runtime.

The Client–Dispatcher–Service pattern is a critical component since, if the dispatcher component fails, the whole system could crash. A failed dispatcher would decouple the connection from the client to the service components. The whole system also becomes less efficient since the dispatcher introduces extra communication overhead during communication channel setup.

3.6. Security and access control – Proxy pattern

Design challenge
The CIS applications must be protected against unauthorized access, since sensitive data about users, services and transactions is contained in the system. The system must be capable of supporting user authentication, secure transactions, and security auditing.

Solution
A Proxy pattern is introduced to address the security and access control problem. In this pattern, the proxy component [Gamma *et al.* 1995] was built as a representation of the service component to limit direct access for the client. The proxy pattern introduces a level of indirection between client and service components.

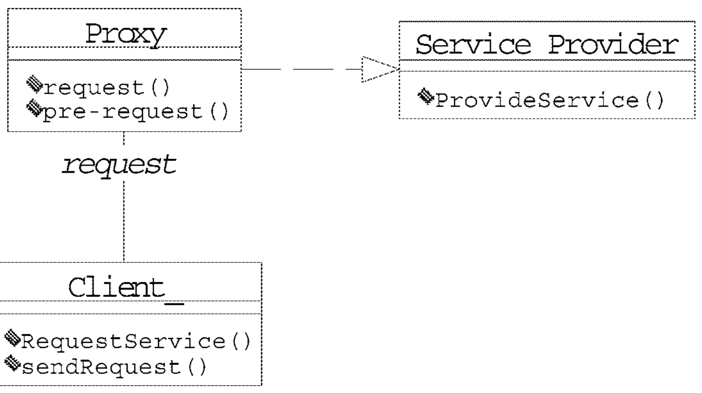

Figure 7. Class diagram: Proxy pattern.

Structure
As shown in the class diagram in figure 7, the major components of the proxy are:

- Client uses the interface provided by the proxy to request a particular service.
- Service provider encapsulates and implements the particular service through the proxy interface.
- Proxy in a web-based environment, acts as an intermediary between the client and the service component, so that the system can support security, administrative control, and caching service. The proxy, can be classified into different categories:
  - Firewall proxy that protects the enterprise network from outside intrusion;
  - Cache proxy that caches frequently requested results to improve system efficiency;
  - Remote proxy that accepts a request and forwards it to a remote server on the client's behalf.

Collaboration
A typical proxy pattern includes the following phases as shown in figure 8:

- an Internet service such as an HTTP request sent to the proxy server;
- if the request accesses the pre-processing, such as filtering requirements, the proxy server, assuming it is also a cache server, looks in its local cache of previously-downloaded Web pages;
- the particular page is returned if it is located. Otherwise, it requests that page on behalf the client using its own IP address. When the page is returned, the proxy server relates it to the original request and forwards it on to the user.

Deployment consideration
In the deployment phase, the functions of the proxy pattern can be deployed separately or combined in a single package. Different server programs can also be in the same or

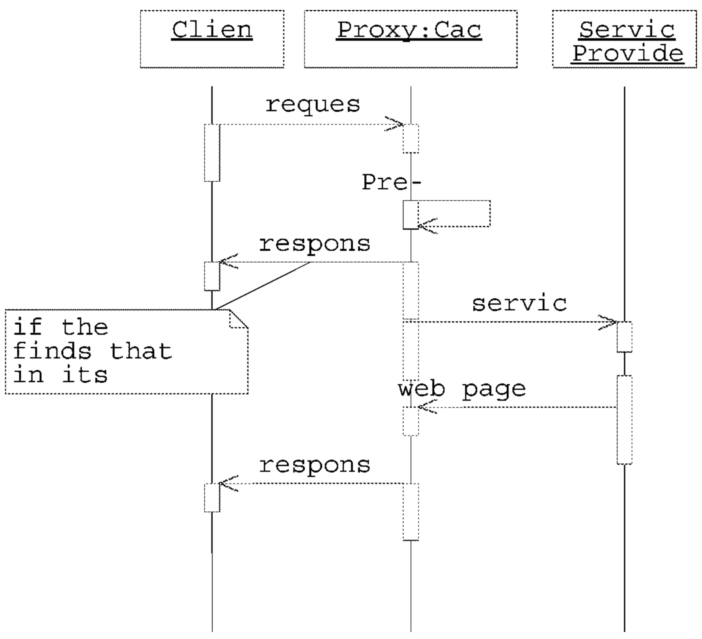

Figure 8. Sequence diagram: Proxy pattern.

different computers. For example, a caching proxy server may be in the same machine with a firewall proxy server or it may be on a separate server and forward requests through the firewall.

Consequences
The Proxy pattern renders the proxy server invisible. All requests and returned responses appear to be directly from the addressed server components.

3.7. XML converter – Broker pattern

Design challenges
A CIS application is a distributed and heterogeneous system of services composed of independent cooperating components. Typically the presentation layer components parse incoming requests and invoke the appropriate service components. The most important aspect of these services is that the partners who are cooperating to provide the services need to agree on the content, semantics, and sequencing of the request messages they exchange. A communication mechanism has to be defined for the data exchange in the Internet-based environment.

Solution
A Broker pattern is introduced that uses XML as a format for common data exchange among the content and/or service providers. XML is a good choice because of the continuing investment by major software companies in developing related XML standards.

The Broker pattern supports better de-coupling of clients and servers. Servers register themselves with the broker, and make their service available to clients through method interfaces. Clients access the functionality of servers by sending requests via the broker. A broker's tasks include locating the appropriate server, forwarding the request to the server, and transmitting results and exceptions back to the client.

Structure
This pattern comprises a few participating components: clients, servers, brokers and optional client-side proxies and server-side proxies [Buschmann *et al.* 1996]. In our solution the client makes requests in XML format for a task to be performed and the server always performs the requested task. The broker component acts as an XML translator or a middle layer between the client and server. The broker provides the intelligence necessary to translate the XML request, look up its internal server registration, and then map this abstract service from the client to a particular server implementation. Moreover, the broker can handle automatic service discovery and routing to an appropriate service component for the client. Once there is a response from the server components, the broker can package the result in XML format and deliver this result to the client.

Consequences
A Broker pattern using XML has the benefit of de-coupling server and client components, and supporting extensibility of components. Brokers usually locate a server by name rather than location. Similarly, servers do not care about the location of calling clients, as they receive all requests from the broker component. This way, only the broker needs to know the location and capabilities of the clients and servers on the system. This is particularly useful when the client asks for some external service or information outside the community application domain. For example, a client requesting a map will have to forward the request to an external map server (such as `Expadia.com`).

In a traditional client–server environment, request invocations are predefined, because the client is aware of all available servers. With the introduction of a broker component, clients can invoke requests on the server at run time since the broker can handle the automatic search and forward the request to the appropriate service component.

An XML-broker pattern has some significant liabilities as well. Obviously the broker pattern can impact performance. Efficiency is restricted because of the indirect communication through the broker. Fault tolerance of the system is lowered. If the server or broker component fails, all the applications that depend on the server or broker are unable to continue successfully. However, reliability can be increased through replication of components.

3.8. Event management – Publisher–Subscriber pattern

Design challenge
Increasing traffic on the Internet makes it necessary to manage large volumes of messages. This can be accomplished by message filtering to reduce the volume of network

traffic, eliminating unnecessary event processing and blocking unsolicited messages. Clients with interest in certain events must have a mechanism through which they can receive an event notification. Objects triggering events need not know the set of clients that desire to receive event occurrence notification. The event notification mechanism minimally supports interest registration, definition and triggering of events, and notification receipt of an occurring event. Event notification improves network efficiency because clients only receive messages from events in which they are interested. We describe an abridged pattern for message management based on the Observer pattern [Gamma *et al.* 1995].

Design alternative: callback
One design alternative could use a callback mechanism. This mechanism is able to notify subscribers of events without inheriting from the event producer. It involves sending messages to objects of an unknown type. This approach has the following limitations:

- Notification scalability. Callback works when there is only one consumer to handle the event and there are no further derivations of the producer. Since objects commonly distribute their events to multiple consumers, callback suffers from scalability problems.
- Callback persistence: must save the callback information in persistent storage in case of a server crash.
- Tight coupling between consumer and producer.
- Poor reliability when callback fails.

Structure
This service requires a messaging mechanism for event propagation and notification. An event is a message between two or more objects about the occurrence of a state transition. Message queues can be used to enable programs to communicate with each other across a network by writing and retrieving application-specific data to and from queues, without a private, dedicated, synchronized connection linking them.

Here are the major components in this pattern:

- Producer: generates the events.
- Consumer: waits for the events.
- Event channel: a logical communication and storage link for events sent by producers or consumers.
- Event Administrator: the central component in this pattern. It allows a user to define a particular event based on constraints. An event consumer can then subscribe to specific event(s). When an event is triggered, the event administrator notifies the event consumer by either a push or pull model.

The event is not the transition, although the transition may trigger it. With asynchronous messaging, the event producer proceeds with its own processing without waiting for a reply to its message. In contrast, synchronous messaging waits for the reply before it resumes processing.

Event management models
Two styles of access, push and pull, are also available thorough the event channel interposing itself between the event producer and consumer. The channel acts as a buffer or mailbox between the producer and consumer. There are at least three event models associated with this approach. They are:

- Pure push event model [Otte 1996]. In the push model, as illustrated in figure 9, the producer pushes a message into the event channel regardless of whether a consumer is waiting for the message or even available to receive the message. The event channel will then push the message to all consumers that have registered for such particular push events.
- Pure pull event model [Otte 1996]. In the pull model, a consumer attempts to receive an event message from the channel. The consumer pulls the event from the event channel, which in turn pulls it from the producer, if the producer is available for pull-style events.
- Hybrid push/pull event model. In this model, the event consumer and producer mix and match the push and pull styles of event processing. There may be multiple consumers as well as producers. This is a feature of de-coupling the event channel as a separate object interposing itself between the producer and consumer of events. An event producer may push to the channel, and consumers may pull that event from the channel. In the following paragraph, we elaborate on the event model using a server-side caching example.

A common strategy to improve the performance of a web server is caching. Of particular interest is the dynamic page caching performed at the web server, where the final result is stored just before being sent to the clients. The cache can be used as the response for similar future client requests such as a product catalog page. However, the cached results could be dynamic in nature. For example, a page displaying a merchant's price will change when the system administrator changes the price in the database.

To avoid caching invalid pages, a pull event model can be used. To implement this strategy the database is polled periodically for type changes and re-generates the result pages for the cache. On the other hand, push event models may be more commonly used in these types of situations. Instead of pulling, the web server relies on the database trigger to aggregate modified merchant and other catalog data into a separate table or file. Once collected, the data is processed and transformed into caching pages, which, in turn, are pushed to the web server and stored for the next requests.

Event management pattern applications
Email filter application, web server caching application, and registered event notification application.

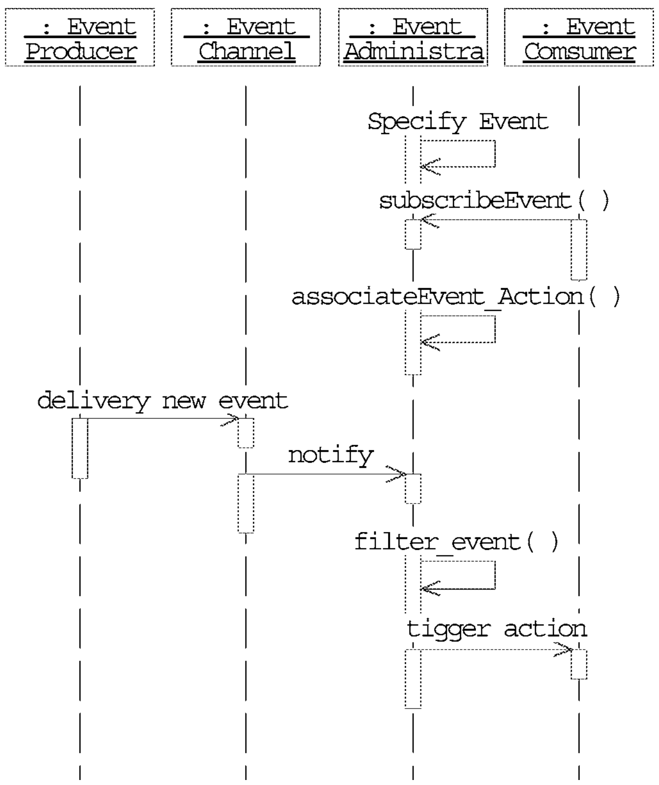

Figure 9. Sequence diagram: Pure Push Event model.

3.9. Unified data interface – Façade pattern

The Façade pattern [Gamma *et al.* 1995] provides a unified interface to a set of interfaces in a subsystem. The Unified Data Interface (UDC) acts as a façade for the data access service by offering clients a single, simple interface to the data repository, whether the underlying database system is a relational database or an object-oriented database system.

Solution
The CIS framework can use the structured query language (SQL) or the open database connectivity (ODBC) as a standard interface. This allows access to dozens of different software vendors' databases on dozens of different platforms. It insulates the business logic components from the details of the database access operations.

Consequence
A benefit of the Façade pattern is that it shields clients from subsystem components thereby allowing the developer to vary the components of the data subsystem without affecting its clients. However, the whole process can become less efficient because of the additional layer between the client and data subsystem.

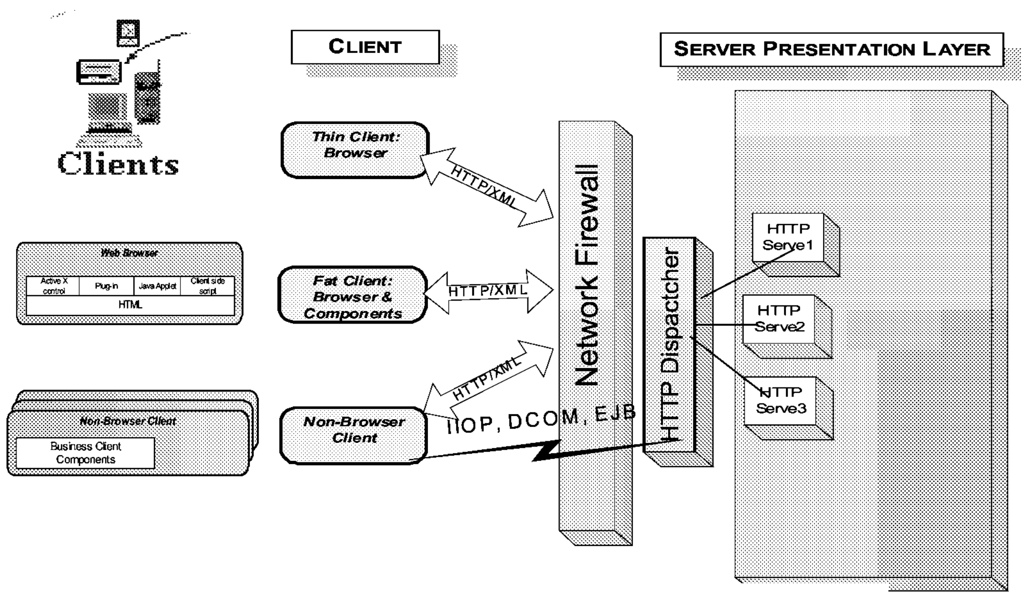

Figure 10. Client categories.

3.10. Client categories

A client application can make network HTTP connection with a web site, invoke the server-side business logic components and subsequently may change the state of the business information on the server. More specifically, the client applications can be divided into four categories as shown in figure 10.

- A *Thin client* requires only a standard web browser and is most commonly used. All the business logic is executed on the server side and there is minimal control of a client's configuration. The advantage of this system is not only reducing the cost of enhancement and maintenance, but also ensuring universal access to a CIS web site and having the biggest customer base.
- A *Fat client* can execute some part of the business logic on a client machine. Typically it uses a standard web browser equipped with extra plug-ins, such as Active X controls or Java applets. It shifts part of the business logic from the server side to the client to make use of the computer power of the client machine. For some applications, it increases performance and makes the server more scalable. This approach can provide a comprehensive user interface, which may be particularly applicable for certain applications, such as 3D models or animated graphs. It has some drawbacks as well. The biggest consequence is that there are more portability problems across different browser implementations. Maintenance and enhancement are issues as well. Users have to download the components before viewing and retrieving information.
- A *Non-browser-based client* includes a special software application running on a conventional computer, kiosk or even pervasive device, such as a Personal Digital

Aid (PDA) and digital wireless telephones. The biggest difference between the non-browser-based and the thin/fat clients' pattern is the communication protocol between the client and the server. In addition to HTTP, other protocols such as IIOP (Internet Inter-ORB Protocol) and RMI (Remote Method Invocation) may be used to support a distributed object system. In some situations, a client can still use a browser primarily as a container or user interface. This pattern, however, needs significant control over client and network configuration.

- A *Special client* has some special software application running on a conventional computer, such as the WAP (Wireless Application Protocol) server used by cellular phone subscribers. It is used as the gateway for some pervasive device, such as digital cellular phones and other wireless devices, to be integrated into the CIS system.

4. Case studies

In this section, we introduce two case studies in different application domains that mapped into the CIS application framework model.

We will use the two case studies to show the different aspects of the CIS framework. In particular, the first case study, WinNet, focuses on the architectural design of a community portal with a map-based user interface. The second study emphasizes patterns in a general-purpose e-service application.

4.1. WinNet

Maps are extremely useful for presentation of location data and also as an effective visual index for the purpose of classifying and retrieving data. In this case we view a map as a collection of objects where each object can have several types of associated data, all stored in the backend data repository such as an SQL database [Ullman 1988]. For example, map objects such as buildings can be associated with data such as history, occupants, energy consumption, room-configuration, selling prices, and taxes.

This association between map objects with different types of properties is called a Hypermap [Alencar *et al.* 1999]. The WinNet was developed from the CIS framework for the GIS (Geographic Information System) domain. We will use this case study example to show how the WinNet maps into the CIS architecture model. In particular, the adoption of the XML-based document-view pattern to separate the concerns will be highlighted in the following paragraphs.

4.1.1. Component-based architecture model

WinNet is based on the component-based CIS framework architecture model in the sense that WinNet is designed to be implemented efficiently using primarily COTS components and reusable domain-specific components. Available COTS components range from free shareware for prototyping, to high-end enterprise software packages for deployment. Some components are:

- A web server is the standard communication component in the presentation layer.
- A Message Queue is a public message delivery mechanism used for asynchronous communication, which is important when applying pipe and filter and/or publisher-subscribe patterns. The Message Queue is available from both Microsoft (MSMQ) and IBM (MQseries).
- Relational database management systems (RDBMS) [Ullman 1988] are mainly used at the backend data repository for data storage and transaction support.
- For object-oriented data, it might be more desirable to store data in an object-oriented database like ObjectStore which is queried through an OQL (Object Query Language) interface.
- Other software packages include ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), billing and accounting packages.
- Reusable domain-specific components are built over a standard infrastructure, which provides fundamental software services and a scalable run-time platform. A number of reusable domain-specific components have been implemented in this project. These include:
  - LivePage content management component – a Windows application, which accesses, views, retrieves, and searches documents or parts of a document that reside in a document repository;
  - HyperMap – a component that associates map objects with different types of geographic-specific properties. The HyperMap component provides a direct interactive user interface to geographic, numerical, textual and multimedia data.

All of these components use the low level services provided by the infrastructure layer. For example, the message transportation service provides an event-based synchronized messaging mechanism for event propagation and notification. Furthermore, the extensible high level services such as dynamic catalog navigation, either custom-made or outsourced COTS ones, are provided by assembling and configuring the components.

4.1.2. Design patterns applied

Extensive use of the customized design patterns has been proven to be critical to the successful design of WinNet. WinNet has been able to achieve separation of different concerns and low levels of coupling among components by implementing both the XML-based Document-View pattern and the Pipe and Filter pattern.

Separation of concerns and low coupling – XML-based Document–View and Pipe and Filter pattern. WinNet is a community portal and offers a collection of e-services to the clients, such as a vector map display, a local search engine and dynamic content presentation. The primary data (object in the map) is stored in XML and the secondary data is inserted in a relational database. The secondary features include all the semantic information related to a map object and can be different types, such as text, sound,

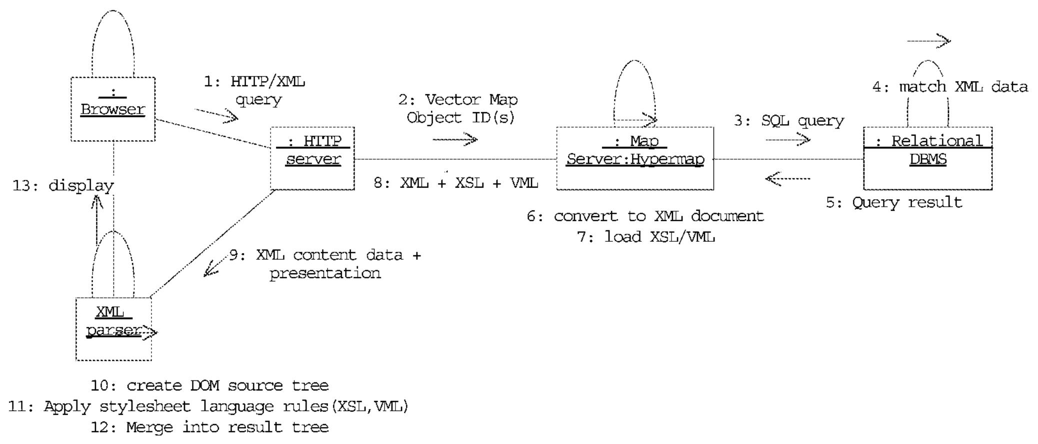
: Browser
1: HTTP/XML query
: HTTP server
2: Vector Map Object ID(s)
: Map Server:Hypermap
3: SQL query
4: match XML data
: Relational DBMS
5: Query result
6: convert to XML document
7: load XSL/VML
8: XML + XSL + VML
9: XML content data + presentation
13: display
XML parser
10: create DOM source tree
11: Apply stylesheet language rules(XSL,VML)
12: Merge into result tree

Figure 11. Collaboration diagram: retrieve map information.

images and so on. Several fields should be extracted from XML prior to their insertion into the database and placed in separate columns for indexing and searching.

We illustrate how the components, either custom-made such as the map server or COTS the browser, and web server, interact with one another to retrieve and present map information to the end user using a UML collaboration diagram.

As shown in figure 11, users access a web browser and request a page from a web server in the Presentation layer. Using the server-side script or CGI, the web server forwards the request to the HyperMap content server component in the Business Logic layer, along with the specific map object ID. The map server then accesses the database to retrieve the data tagged with XML, through an SQL interface. The HyperMap server can then transform the query result in XML format into an HTML document, using the XSL/VML presentation format template, if the client can only interpret HTML documents. Alternatively, the HyperMap server can route the XML data document directly, along with the style-sheet information in XSL format, to the web server. In turn, the web server will return the separated XML and XSL document to the client, which is capable of merging the layout information with the XML document, and producing the final presentation of the map query result. By using XML, XSL and VML, the processing of map-related property data can be completely separated from its presentation. As a result, Web developers can push much of the presentation to be displayed to the client, freeing web servers to do more processing.

Push and pull event modeling – Publisher–Subscriber pattern. The Publisher–Subscriber pattern from the CIS framework is used in the development of the notification server component in the event management environment. It supports the push and pull event model and the asynchronous messaging mechanism, which in turn reduces the coupling and makes the whole framework ready for extension.

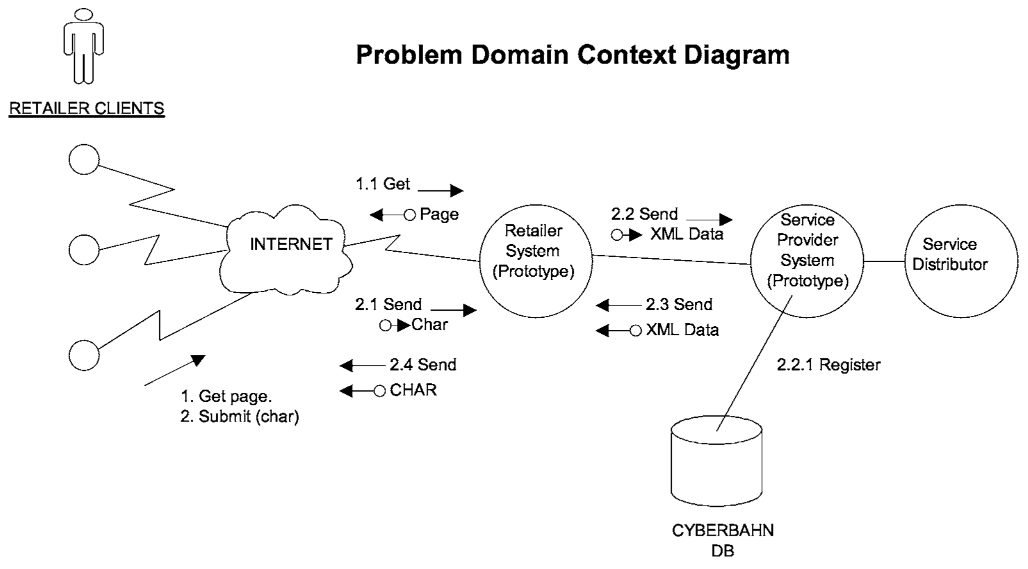

Figure 12. Problem domain context diagram.

4.2. Cyberbahn project

The Cyberbahn (CB) project is the second application, mapped to the CIS framework, which we are going to present as a case study. `Cyberbahn.ca` is a Toronto-based value-added online service, which provides its clients, mainly lawyers in Ontario and Quebec, with electronic access to various legal documents and reports related to corporation registration information. For example, instead of going to a government counter in person, the lawyers can now search, retrieve and print a particular corporation registration report online. A diagram of the CB system is in figure 12.

CB provides functionality in the following areas:

- Integrated legacy system functions. All the corporation data stored and managed by the CICS legacy application is hosted on the Ontario government AS-400 mainframe. The application was mainly written in COBOL in the 1980s. CB has to leverage the old COBOL code and, through the use of some conversion utilities, retrieve the stored report data and present it using the web based user interface to the clients.
- Support for both B2C (business to consumer) and B2B (business to business) models. CB should support both retailer and wholesaler models. As a retailer, the CB system delivers the requested services to the paid customers. As a wholesaler, CB purchases the information from the distributor, the Ontario government in this case, and sells this information to the other electronic retailers such as `bell.ca` and `streerling.com`, who wish to provide similar services to the customers.
- Both thin client and fat client support. Most of the users belong to the Thin Client category, which requires only a standard web browser acting as a user interface device. For the user who has a sophisticated browser or supports client-side scripts or

plug-ins, a fat client approach is more desirable, since it brings end users an enhanced user interface that can execute business logic on the client side.

- Easy to reuse and be extended. The CB system should support software reuse by both incorporating reusable components into the overall architecture framework and allowing the design of general components to be used in other systems. Changes and extensions should be made to a specific location in the system and the rest of system should not be negatively affected.

The CB system is built upon two main concepts: an extensible framework to provide high-level customized service through the composition of domain-specific components and COTS components; an extensive use of design patterns to satisfy both functional and non-functional requirements.

4.2.1. Architecture model

A consideration of system requirements, as detailed in the previous section, leads to a variation of the three-tier client/server conceptual architecture model to fulfill these needs. The overall system consists of four levels as depicted in figure 13.

- The Client level represents the end users who request the value-added services from the CB system. This level can be further divided into thin clients and fat clients.

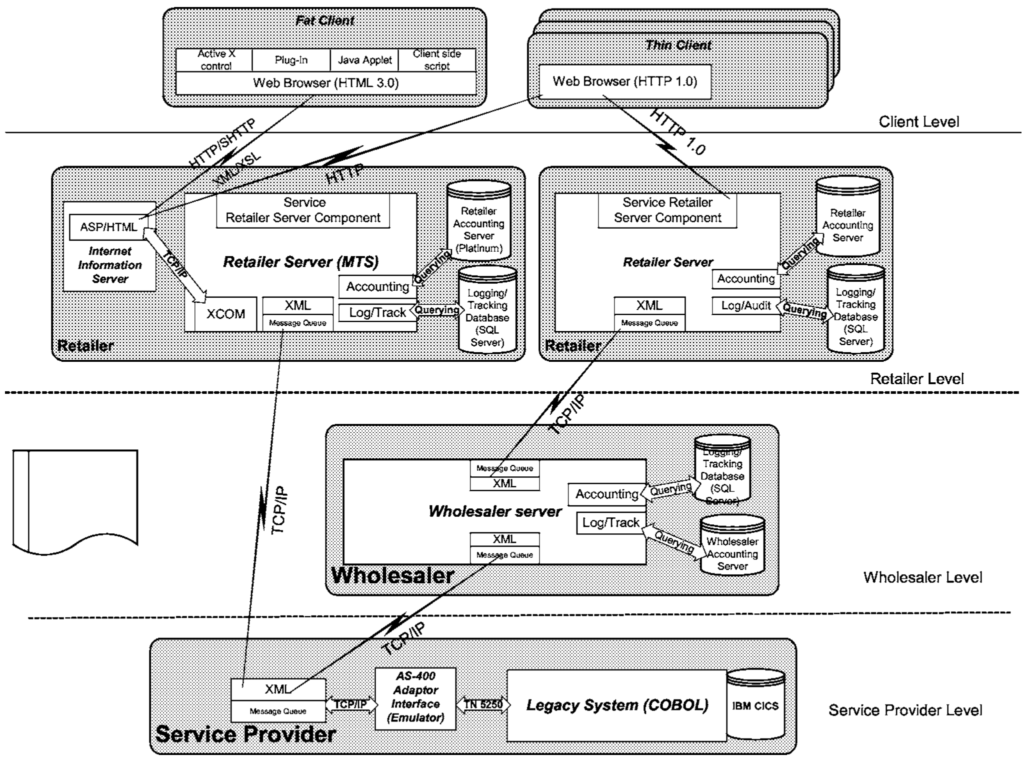

Figure 13. CB system architecture model.

- The Retailer server level, as the name suggested, represents a B2C electronic retailer for the direct clients. It provides a visual interface for presenting information and gathering data from a wholesaler and/or distributor. For a thin client, it provides the basic HTML document. But for a fat client, it can provide XML documents and enhanced user interface features by taking advantage of client side business logic execution. This level consists of web servers, retailer service components and backend databases.
- The Wholesaler server level serves as the broker for the B2B business model by taking the request from the retailer and forwarding the request to the distributor. Once the response comes back, the Wholesaler returns the response to the Retailer. In terms of implementation, the Wholesaler is very similar to the Retailer, except there is no visual interface.
- The Distributor server level integrates a government legacy system through encapsulation.

As illustrated in figure 13, central to the architecture is the asynchronous messaging mechanism and the coarse-grained business logic components, retailer and wholesaler server in this case. From the previous description it is clear that the use of components in this framework has some advantages.

- It simplifies the application composition task since it allows the use of reusable components for the high-level domain specific service. For example, with minimal runtime change in interface, the Retailer component can be re-configured and deployed as Wholesaler since they have very similar functionality.
- It provides an extensible framework-architecture for adapting to changes in requirements. Communication between the coarse-grained components is exclusively via the message queue. This keeps the logic components loosely coupled and thus establishes the basis for component plug and play functionality.

4.2.2. Design patterns applied

Pipeline protocol – Pipe and Filter pattern. The Pipe and Filter pattern, implemented in CB, represents the communication backbone for the overall system. Conceptually, the CB system processes and transforms a stream of data in several stages. Figure 13 describes messages and data flow between individual components, which can be considered as filters in this pattern. For example, the client accesses Input Form through the Internet (messages 1 and 1.1), enters data to the Input Form (corporation name or registration ID) and selects the Submit button to send a request (with data) to the Service Provider and then waits for a response (message 2.2). The Service Provider applies Business Rules (tracking and billing), records the transaction (adding new record to the transaction table, message 2.2.1) and generates the result. The result is then returned to the Retailer (message 2.3), which, in turn, displays the result to the client (message 2.4).

As in figure 13, the flow of data between the Retailer System and the Service Provider System is in XML format. The data between these two systems are transferred through the MQ Series message queue. The reasons for these two technologies are clarified below.

- XML message format. XML is used as a meta-data language that provides a set of tags that are used with name-value pairs. This means that every value (part of data) is described by its name. And because XML is self-describing, applications are insulated from changes in the underlying message format. This provides great flexibility in adding new request forms or modifying existing ones without worrying about the position of name-value pairs in a XML document.
- Message queue. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. In contrast, synchronous messaging waits for the reply before it resumes processing. Moreover, commercial message queues provide the transaction semantics for the message propagation, which means once a message is sent, there is a guarantee that the message will be delivered to the desired destination once and only once using this communication mechanism.

As we can see, two important issues in framework customization in this case are related to the separation of concerns and decoupling. This system uses data of various components, and avoids globally shared data by using asynchronous message-passing.

Integrate legacy system – Adapter pattern. The implementation of both the Adapter (also known as Wrapper) and Facade patterns for abstract interfaces provides an integrated view of the legacy data sources, without changing how or where the data is stored. In the CB project, a key challenge was how to integrate the data source and business logic of the government legacy system into the application framework to take advantage of desktop computing models and the new default user interface, the HTML/XML browser. To address the requirement, we introduced two components, an XML message queue and an AS-400 adapter interface, as shown in figure 14.

- Legacy Interface Adapter. The Legacy Interface Adapter, also know as an emulator or wrapper, is used to access the CICS legacy application operating on a government mainframe host. Because the legacy application has no other communication mechanism other than the proprietary terminals they were designed to use, an emulator has to be built to simulate the keyboard input and manipulate the legacy application via TN5250 protocol. The major benefit of screen emulation is that the legacy application does not require any change whatsoever.

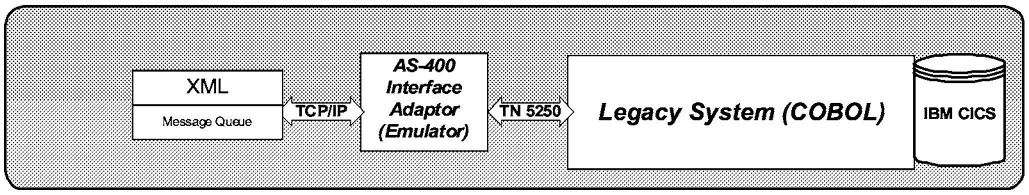

Figure 14. Integrate legacy system – Adapter pattern.

- XML Message Queue Interface. This interface serves as a message layer that isolates and uncouples the legacy application component from the rest of the information system. This legacy message modeling approach achieves this by abstracting the adapter's data and behavior in the form of a collection of messages.

By using these two key components, the Legacy Interface Adapter and the XML Message Queue Interface, we can leverage the storage or data management facilities provided by the underlying segregated legacy system. This will provide a unified schema and common XML-based message communication interface for the overall system without disturbing existing applications.

5. Conclusions

Object-oriented application frameworks and design patterns help to reduce development costs and improve the extensibility of software by leveraging proven software designs and implementation to produce reusable components that can be customized to meet growing application requirements.

In this paper, we first provided an extensible framework specification designed to meet the need for a flexible, cost-effective community information system (CIS) using the Internet and the World-Wide Web. We then illustrated how a collection of design patterns can provide solutions for the design challenges raised in the CIS domain. Finally, we presented two case studies, the WinNet and Cyberbahn projects, to show how to map the application into our framework model. The ultimate objective of this research was to create a framework that enabled a domain expert or less experienced application programmer to create specific Internet applications without having to acquire extensive knowledge about the framework.

From architecture model perspective alone, there exist many similar designs. For example, WebSphere Commerce suite from IBM [Websphere 2001], Enterprise Java services from Sun [Java2 Platform 2001], COM+ [Brown *et al.* 2000] and the latest .NET framework from Microsoft [Richter 2002]. The CIS framework is not fundamentally different from those web-based architectures mentioned above. However, design patterns applied in this paper are ubiquitous in the CIS framework. Its application not only greatly contributes to the understandability and traceability of the framework, but also improves software reusability and makes the architecture more integrated and scalable.

The following lessons resulted from our experience with this approach:

- Avoid excessive data coupling among framework components. With excessive data coupling, the framework loses its flexibility, and updating any components becomes quite difficult. The desired framework components should have clean interfaces, be cohesive, and have as little data coupling as possible.
- The framework should have a standard base whenever possible. It's especially important in a distributed computing environment. Using a standard base avoids problems with integration and communication with other applications.

- Use as many design patterns as possible. Using patterns is effective simply because it represents design reuse and works better than inventing solutions from scratch.

The Community Information System framework implements most of the features described early in the paper. The framework is open and easily extendable. Some of the framework components are off-the-shelf (COTS) and commercially available. This framework has been instantiated to produce many applications such as the ones described in section 4. However, the CIS application framework is not yet mature. The work presented in this paper can be extended further.

We extended the CIS framework by defining interfaces for components that can be plugged into the framework using object composition and design patterns. We were unable to take full advantage of object-oriented features, such as inheritance and dynamic binding, to extend the framework. In particular, we were unable to inherit from the framework component base class or override the predefined hook method using design patterns, to reuse and extend existing component functionality.

Besides design patterns, we believe that hot-spot analysis can be a powerful tool in our CIS framework designing process. Hot spots are the variable aspects of a framework domain, and different applications from the same domain differ from one another with regard to at least one of the hot spots [Alencar *et al.* 2000; Fayad *et al.* 1999a]. We intend to combine the hot-spot analysis and design patterns to achieve a more comprehensive understanding of our CIS framework development process.

Acknowledgements

This work is partially supported by NSERC and the Consortium of Software Engineering Research (CSER).

References

Alencar, P.S.C., D.D. Cowan, S. Crespo, M.F. Fontoura, and C.J.P. Lucena (2000), "Using Viewpoints to Derive Object-Oriented Frameworks: A Case Study in the Web-Based Education Domain," *Journal of Systems and Software (JSS) 54*, 239–257.

Alencar, P.S.C., D.D. Cowan, and M.A.V. Nelson (1997), "An Object-Oriented Framework for Hypermaps," In *Proceeding of the 2nd International Symposium on Environment Software System (ISESS'97)*, Whistler, BC, pp. 244–251.

Alencar, P.S.C., D.D. Cowan, and M.A.V. Nelson (1999), "An Object-Oriented Framework for Hypermaps," *Environmental Modeling and Software Journal*.

Booch, G. (1994), *Object-Oriented Analysis and Design with Applications*, Second Edition, Addison-Wesley, New York, NY.

Brown, R., W. Baron, and W.D. Chadwick III (2000), *Designing Solutions with COM+ Technology*, Microsoft Press, Redmond, WA.

Buschmann, F., R. Menuier, H. Rohnert, P. Sommerlad, and M. Stal (1996), *Pattern-Oriented Software Architecture – A System of Patterns*, Wiley, New York.

Cowan, D.D. (2000), "Community Network: A Next Generation," In *Electronic Commerce Technology Trends: Challenges and Opportunities*, IBM Press, pp. 41–53.

Eriksson, H.E. and M. Penker (1998), *UML Toolkit*, Wiley Computer Publishing, New York.

Fayad, M.E., D.C. Schmidt, and R.E. Johnson (1999a), *Building Application Frameworks*, Wiley Computer Publishing, New York.

Fayad, M.E., D.C. Schmidt, and R.E. Johnson (1999b), *Domain-Specific Application Frameworks*, Wiley Computer Publishing, New York.

Fontoura, M. F. (1999), "A Systematic Approach to Framework Development," Ph.D. Thesis, Computer Science Department, Pontifical Catholic University (PUC-Rio), Rio de Janeiro, RJ.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Garlan, D. and M. Shaw (1993), "An Introduction to Software Architecture," In *Advances in Software Engineering and Knowledge Engineering*, Vol. 2, World Scientific, Singapore, pp. 1–39.

German, D.M., D.D. Cowan, and P.S.C. Alencar (1998), "A Framework for Formal Design of Hypertext Applications," In *Proceedings of SBMIDIA*, Rio de Janeiro, Brazil.

Jacobson, I., M. Christerson, P. Jonsson, and G. Overgaard (1992), *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, MA.

Java2 Platform (2001), *Java2 Enterprise Edition*, Sun Microsystems, available at `http://java.sun.com/j2ee/overview.html`.

Knudsen, J.L., Ed. (2001), *The European Conference on Object-Oriented Programming*, Springer, Heidelberg.

Kruchten, P.B. (1999), "The 4+1 Views Model of Architecture," *IEEE Software 13*, 6, 42–50.

Larman, C. (1999), *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice-Hall, Frisco, TX.

Nova, L.C.M, P.S.C. Alencar, and D.D. Cowan (1998), "Modeling the Design Pattern Application Process," In *Proceedings of the Symposium on Software Technology (SST'98)*, pp. 203–210.

OMG (2001), *Object Management Group*, available at `http://www.omg.org/`.

Otte, R., P. Patrick, and M. Roy (1996), *Understanding CORBA – The Common Object Request Broker Architecture*, Prentice-Hall, Englewood Cliffs, NJ.

Perry, D.E. and A.L. Wolf (1992), "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes 17*, 4, 40–52.

Richter, J. (2002), *Applied Microsoft .NET Framework Programming*, Microsoft Press, Redmond, WA.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorenzen (1991), *Object-Oriented Modeling and Design*, Addison-Wesley, Reading, MA.

Szyperski, C. (1997), *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, Reading, MA.

Ullman, J.D. (1988), *Principles of Database and Knowledge Case Systems*, Vol. 1, Computer Science Press, New York.

W3C (2001), *World Wide Web Consortium*, available at `http://www.w3.org/`.

Websphere (2001), *Websphere Software Platform*, IBM Corporation, available at `http://www-3.ibm.com/software/info1/websphere/index.jsp`.