The Collection-View Model: A Pattern Language for Software Reuse Tools

Colin A. Depradine and Brian G. Patrick

Department of Computer Science, Mathematics and Physics
University of the West Indies
Cave Hill Campus
Bridgetown
BARBADOS
E-Mail: cdepradine@uwichill.edu.bb, patrickb@uwichill.edu.bb

Michel de Champlain

Department of Electrical and Computer Engineering
Concordia University
Montreal, Quebec
H3G 1M8
CANADA
E-Mail: mdec@acm.org

Abstract

Many programmers continue to depend on the syntax of a programming language and the associated documentation of a software component to perform code reuse. For a small number of components, this form of code reuse is easy to control. But as the number of components increases, the managing and searching of information becomes increasingly difficult and unwieldy. To support the activities of gathering, storing, querying and viewing software reuse information, the **C**ollection-**V**iew **M**odel (CVM) is introduced as a pattern language for the design of code reuse systems. These systems are aimed at the traditional users of standard and professional **I**ntegrated **D**evelopment **E**nvironments (IDEs).

1 Context

The practice of software reuse involves the systematic creation, congregation, retrieval and maintenance of software components for multiple deployment across more than one software application. It is a practice that holds out the promise of lower production costs and higher software reliability provided affordable,

Copyright (c) 2000, Colin Depradine.
Permission is granted to copy for the PLoP 2000 conference.
All other rights reserved.

flexible and unintrusive software tools exist to support each facet of this ongoing process [2]. With a rapidly expanding store of well-tested and readily available software components and with the movement toward object-oriented paradigms of design and implementation [15], it is imperative that software reuse tools are developed to counter the proliferation of code variations and to facilitate the access to existing repositories of reusable code [1, 9, 13].

The process of reuse is relatively straightforward. Initially, software components are developed internally or gathered externally for future reuse. Once a component is complete and in order to support the query process, information about the code itself is either collected manually or scanned automatically. The code along with its associated information is then stored in a repository of reusable components. To be effective, the consistency and the correctness of the repository must be maintained at all times. Software components that are added to a repository are made available for reuse in other projects. Components are retrieved either manually or via a querying system. In the latter case, the programmer examines the design requirements and formulates a query to determine if existing code satisfies these requirements. The query is submitted to and executed by the search engine of the repository. The results of the query are then passed back to the programmer for viewing. Based on these results, the query may be modified and re-executed. Finally, software components are periodically upgraded in response to environmental changes and errors.

In this paper, the **C**ollection-**V**iew **M**odel (CVM) is introduced. It provides an architectural framework that supports the development of software reuse tools that are aimed at the traditional users of standard and professional **I**ntegrated **D**evelopment **E**nvironments (IDEs). For current systems, reuse is limited to a basic proprietary repository for components and the search is constrained to templates indexed by keywords. Many of them are also restricted to a specific version of a programming language. This is undesirable since with the rising popularity of e-commerce and web-based applications, the use of two or more programming languages has become the norm.

The Collection-View Model provides a mechanism for the creation of a software reuse system that circumvents the major difficulties involved with a proprietary repository, lack of extensibility and the inability to accommodate multiple programming languages. Originally, the main focus group consisted of IDE end users where the model was a means of extending the reuse capabilities of these systems. However, the model is both flexible and powerful enough to be used by the actual developers of IDEs and small development companies. In fact, it is envisioned that the CVM could encourage the creation of an open, standard format for the development of reuse systems.

The overall structure of the model is described in section 2. This is followed by a description of its main activities in the five subsequent sections. Finally, concluding remarks are offered in section 8.

2 Structure of the Collection-View Model

The **C**ollection-**V**iew **M**odel (CVM) as outlined in figure 1 is comprised of a collection and four functions that reflect the primary activities of the software reuse process: The gathering, the storing, the querying and the viewing of software reuse information. The CVM does not define the process used for creating reusable software components nor does it define how they are to be maintained. Patterns such as [3] already exist

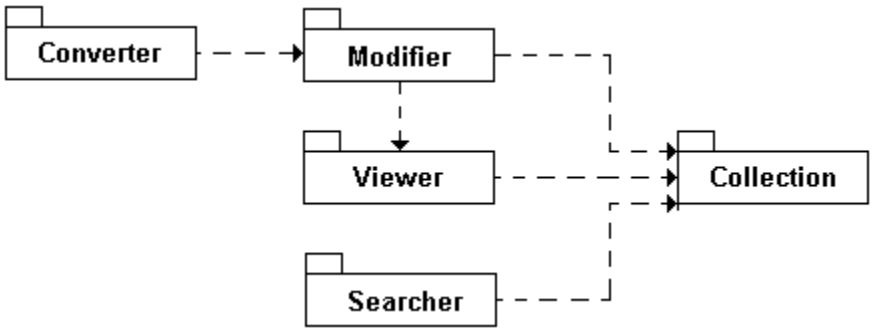

Figure 1: Collection-View Model

to define these processes. Each of the four activities is expressed in terms of a design pattern and together, these patterns specify a pattern language that defines the overall functionality of the model. Whereby the collection represents a storage area or repository for the reuse data, the design patterns define the interface for user interaction with the collection.

The converter scans a software component and transforms the resultant data into a form understood by the modifier. Programming language independence is therefore maintained since the complexities associated with each language is confined to the converters. This reduces the risk of creating a software reuse system that is restricted to a specific programming language. The modifier updates the collection with new or updated reuse information. It is the primary access point and so is responsible for maintaining the integrity of the data within the collection. The modifier is unaware of the converters thereby permitting the smooth addition and removal of programming languages within the system. The searchers are used to perform any queries of the collection. They determine how queries are created and submitted for execution. For example, a query language could be developed or a specialised graphical user interface employed. To ensure consistent results, the model requires each searcher to consult the collection before performing the actual search. Therefore, while each searcher provides its own user interface, the underlying search mechanisms would be the same. The viewers provide specialised views of the data stored within the collection. The content and display format used will be determined by the viewer. To keep the information current, the modifier will notify each active viewer of any changes made.

One advantage of separating the gathering, storing, querying and viewing activities among these four functions, is simplicity. By assigning each activity to a single function, the resultant design is simplified and therefore easier to maintain. This allows the functions to be separated into four distinct design patterns which are described in sections 4 to 7 .

An implementation of the CVM, the **R**euse of **O**bject-**O**riented **C**lasses (ROOC) system, will be included in the Known Uses section of each pattern. This system was developed with the primary aim of enhancing the software reuse facilities of both standard and professional IDEs [7]. It consists of a highly flexible and functional repository of classes supported by a suite of utilities for accessing and modifying the stored information. The ROOC system facilitates the reuse of object-oriented classes by smaller development teams

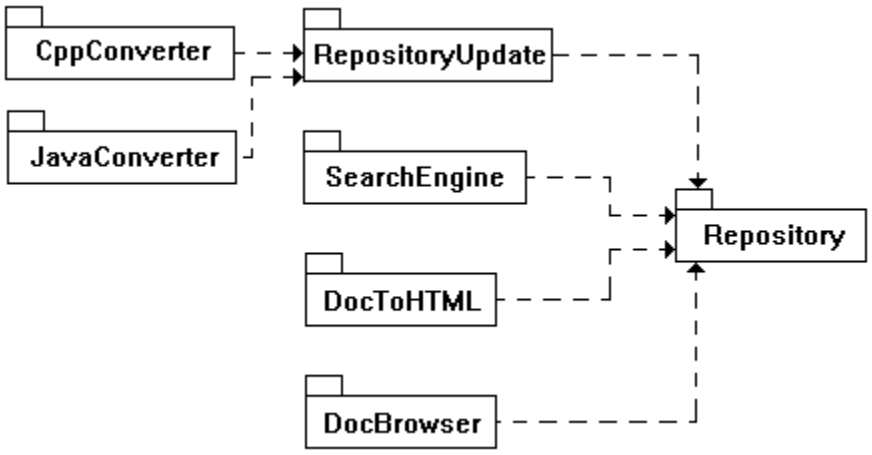

Figure 2: The ROOC System

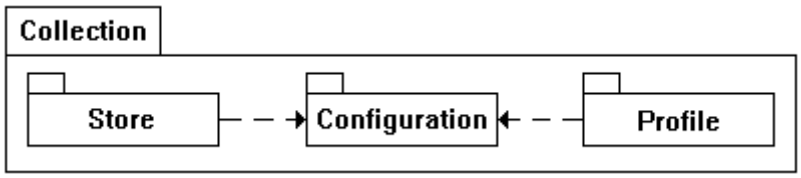

Figure 3: Structure of a Collection

and individual developers. And therefore, it was developed to execute with minimal resources but without severely compromising its power and flexibility. Figure 2 outlines the main tools of the ROOC system and their interactions with the repository. Comparing figures 1 and 2, CppConverter and JavaConverter are the converters, RepositoryUpdate is the modifier, SearchEngine is the searcher, DocBrowser and DocToHTML are the viewers and Repository is an implementation of the Collection.

3 The Collection

The collection is the repository of reusable software components and their associated documentation. It is based on object-oriented database techniques and consists of three main sections [11, 12]: The configuration, the profile and the store as shown in figure 3. The configuration defines how information associated with a component is gathered, stored and matched. The configuration is composed of one or more configuration objects. Each configuration object describes the properties of a single attribute or relationship and includes the following types of information:

- The *preface* which describes the use and purpose of an attribute or relationship.
- The *collection method* which defines how attribute and relationship values are gathered and stored.

For example, the attribute values may be stored either as a list or individually.

- The *matching process* which sets the parameters for the search engine and defines how matching is performed for a particular attribute or relationship value. For example, the match may or may not be case sensitive.

Thus, the configuration objects provide the developer with the ability to construct a repository that satisfies the reuse requirements of the environment. Furthermore, the semantics and use of attributes and relationships can now be standardised. The profile is a brief summary of the attributes and relationships stored in the configuration. It is also used by searchers to determine if the collection is worth scanning.

The actual software components and their respective component objects are contained in the store. The component object is composed of one or more instances of configuration objects where each instance of a configuration object states a value for an attribute or relationship used by the component. By encapsulating all relevant reuse data in the component object and storing it alongside its corresponding component in the central collection, searching and maintenance can be optimised and simplified.

The Collection-View Model takes into consideration that different types of reuse information are required at different points in the development process. For example, when selecting a reusable software component, the developer initially requires prefatory and usage information such as the version number of the component. With this in mind, configuration objects can be classified into various categories. Often though, these categories are too finely-grained to be used alone and hence, combinations of categories are typically implemented. Categorisation permits collection classification and defines the level of abstraction that is available to the user. Search engines also use classifications to determine the suitability of collection querying. Furthermore, the classification of a collection can be included in its profile.

3.1 The ROOC Repository

The main purpose of the ROOC system is the collection and storage of reuse information gathered from either C++ or Java programs, or both. Since two different programming languages were being considered, the repository storage format had to be language independent. A format called the **D**ocument format for the **R**euse of **O**bject-**O**riented **C**lasses (D-ROOC) was created and defines the syntax used by the repository [5]. Information is comprised of three basic objects: The User-defined Relationship Objects, the Syntax-based Relationship Objects and the Attribute Objects. The Syntax-based Relationship Objects contain reuse information that can be inferred from the syntax of the language itself, the User-defined Relationship Objects contain any user-defined relationships, and the Attribute Objects contain the rules that are required to interpret the information found in the User-defined Relationship Objects. There is one Syntax-based Relationship Object and User-defined Relationship Object for each class in the repository. Another major component of the repository is the usage document. One document is created for each class added to the repository and contains prefatory and functional information, for example, general descriptions of classes and their public slots and methods.

To maintain the programming language independence and flexibility of the repository, a mechanism for describing the Attribute Object rules was needed. A language called the Relationship Attribute Action

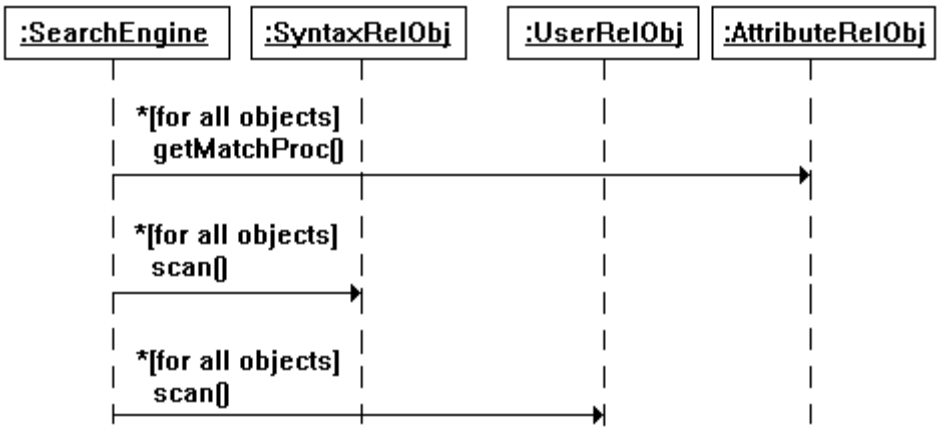

Figure 4: Sequence Diagram of the Repository

Language was created. It describes the structure and use of every relationship and attribute handled by the system. This allows the repository to handle any future changes in either the C++ or Java programming languages. Figure 4 shows the sequencing of the various objects within the repository where SearchEngine is the name of the search engine used to scan the repository. The Attribute Objects are first consulted by SearchEngine in order to obtain the matching and structural profile of all attributes and relationships handled by the repository. This enables the search engine to standardise the search process.

One important aspect involved with the creation of the repository, was the standardisation of the reusable attributes and relationships. This procedure consisted of documenting those features deemed as important for the software reuse process. The major risk associated with this process was ensuring that the reuse data was defined in a programming language independent form. This was not as difficult as initially expected. First, the reusable features were discovered and documented by considering previously created components at the high-level design stage. For example, many of the user-defined relationships were based on categorizing schemes, keywords and relationships used at the design stage. In the case of the syntax-based relationships, several popular object-oriented languages, such as C++, Java, Smalltalk, Python and Eiffel, were compared [4, 10, 15, 17]. This removed any bias towards a particular language.

4 The Converter Pattern

Once code has been created or is identified for reuse, a converter gathers relevant information by scanning and translating the source file into a form that is understood by the modifiers (section 5). However, before the translation is performed, the converter consults the configuration objects in the collection to ascertain the type of information that is required. Converters may be aware of specific modifiers and therefore, can select a specific modifier under certain conditions. This allows more than one programming language to be handled without compromising the programming language independence of the repository. One important factor to consider during the creation of converters, is the presence of legacy code. As a result of its age,

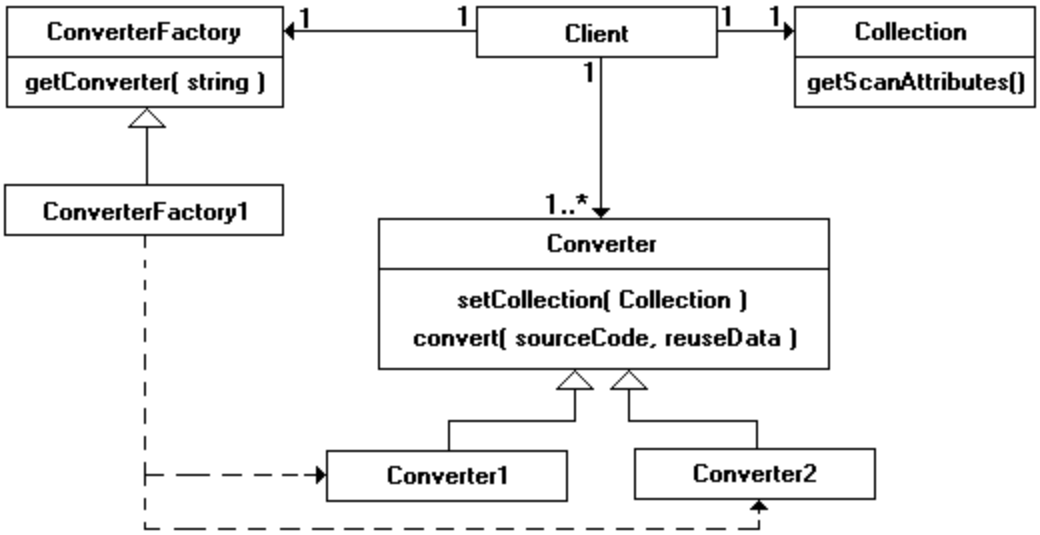

Figure 5: Converter Design Pattern

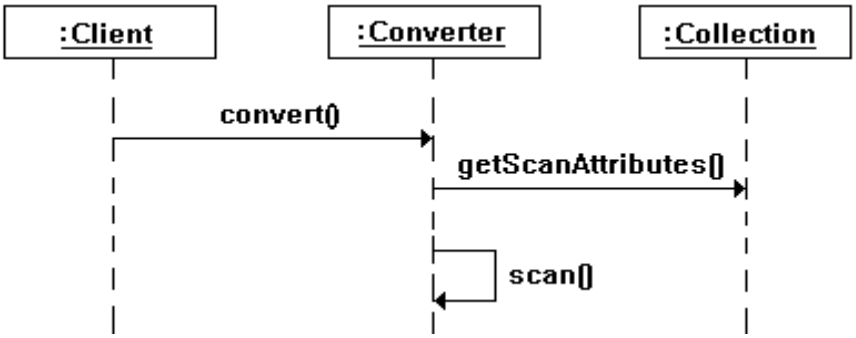

Figure 6: Converter Sequence Diagram

the legacy code may not have the required modern syntactic properties and so specific reuse details may be missing. Therefore, it may be necessary to create converters specifically targeted at the conversion of the legacy code.

Motivation:

For ease of maintenance, it is generally better to collect all reuse information in a single repository. However, it is not uncommon for a developer to use more than one programming language to satisfy the various needs of a project. For example, Java may be used for the client end and C++ for the server end of a database querying system. In these cases, the developer would collect all code in a single repository in order to simplify the process of searching. The information, stored in a language independent format, allows the repository to handle any programming language. Hence, the original code must be scanned and converted to this format.

Structure:

The structure is shown in figure 5.

Participants:

The pattern uses:

- **ConverterFactory:** Provides the converter class needed for the programming language. This is done via the interface method `getConverter`.
- **Converter:** Handles the actual conversion of the code via its `convert` method.
- **Collection:** Provides the interface to access information stored in the collection.

Collaborations:

The sequencing is shown in figure 6.

- A code conversion message `convert` is sent by `Client` to `Converter`.
- `Converter` then obtains the profile of the stored data via the `getScanAttributes` message.
- Finally `Converter` scans the code, collecting the required reuse information.

Consequence:

The main benefits of using this pattern are:

- The repository maintains its programming language independence.
- For each new programming language, only a new converter class need be created and added to the `ConverterFactory` class.

The main disadvantage of using this pattern is:

- The presence of a large number of converters may place contraints on the changes that can be made to the modifier data format.
- The complexity of a programming language may make future modifications to the converter difficult. It is expected that modifications will be required each time the programming language specification is changed.

Implementation:

In this example, the Converter design pattern is based on the Abstract Factory pattern where `ConverterFactory` generates the converter class necessary for the particular programming language under consideration. The current language is passed to the `getConverter` method and the required converter class is returned.

```
Converter *ConverterFactory::getConverter( Language currentLanguage )
{
     Converter *CurrentConverter;

     if( currentLanguage == Java_Language )
```

```
        {
             CurrentConverter = new JavaConverter;
        }
        else
        {
             CurrentConverter = new CppConverter;
        }
        return( CurrentConverter );
    }
```

The interface for the `Converter` class consists of two main methods: `setCollection` and `convert`. The `setCollection` method informs the converter of the collection currently under consideration.

```
void Converter::setCollection( Collection *NewCollection )
{
     CurrentCollection = NewCollection;
}
```

The `convert` method performs the actual conversion and is provided with two main parameters: `sourceCode` and `reuseData`. The `sourceCode` is the location of the code to be scanned and the `reuseData` is the location to place the output. The `reuseData` is important since this is where the modifiers will search for the new reuse data. `Location` refers to the physical location of the data such as a file path or an Internet address.

```
void Converter::convert( Location sourceCode, Location reuseData )
{
     ConfigurationPackage *config;

     // Consult the collection to determine what information should be gathered
     config = collection->getScanAttributes();

     // Scan the code
     scan( sourceCode, reuseData, config );

     // Notify the modifier
     modifier->update( reuseData );
}
```

Finally, the `getScanAttributes` method of the `Collection` class allows the converter to query the collection on what type of information to gather.

By adhering to the `Converter` interface, new converters can easily be added. Also, the resultant isolation of the `Converter` concrete classes by `ConverterFactory`, provides the programming language independence.

Known Uses:

The ROOC system provides two separate programming language converters, one for C++ (CppConverter) and the other for Java (JavaConverter) [7]. They are responsible for the scanning and conversion of the reuse information, found within the code, to the D-ROOC format. In keeping with the converter pattern, neither tool is allowed to directly modify the repository. This enables the creation of a converter without worrying about the details of repository modification. Also, the repository format can

be changed without affecting the converters. Figure 7 contains the sequence diagram of the CppConverter utility. The CppConverter tool obtains the current repository profile by sending a `getProfile` message to Repository. It then scans the C++ code and notifies RepositoryUpdate via an `update` message.

The initial populating of the repository required that legacy code be scanned and the reuse details extracted. However, only a small number of these components contained the embedded comment information required by D-ROOC. Two solutions were considered: The addition of the D-ROOC information directly to the legacy code or the creation of a converter that would fill in the missing details. The first scenario was the most accurate and reliable one but required a high initial effort. The second case had a lower initial effort but was not as accurate since many of the user-defined relationships would be missed. Also, depending on the number and type of relationships handled, the costs of creating such a converter could have been prohibitive. Therefore, the costs of creating the converter had to be weighed against the total number of legacy code components.

During the development of the legacy code, a standard structure for comments had been followed. This made the extraction of many of the reuse details rather straightforward and so the second solution was adopted. It should be noted that those reuse relationships that could not be automatically scanned were left out so as to reduce the overall initial costs.

5 The Modifier Pattern

A modifier updates the collection with the new information provided by the converters. Before carrying out the update, the configuration objects are first consulted to determine how the information is to be stored. Modifiers are unaware of particular converters but can be notified by a converter to perform specific actions on the collection. Hence, converters can be easily added and removed without effecting the modifiers. As a result, the repository can be updated in a consistent manner without compromising the integrity of the data stored.

Motivation:

It is important that the data integrity of the collection be maintained throughout its lifetime. One reliable method of achieving data integrity is to restrict the updating procedure to a single point of entry. Hence, the integrity of the data in the collection now depends on this single access point. Therefore, each time the collection format is changed only a single modifier is updated, reducing the number of possible errors.

Structure:

The structure is shown in figure 8.

Participants:

The pattern uses:

- **Modifier:** Provides the interface for the modifier classes.

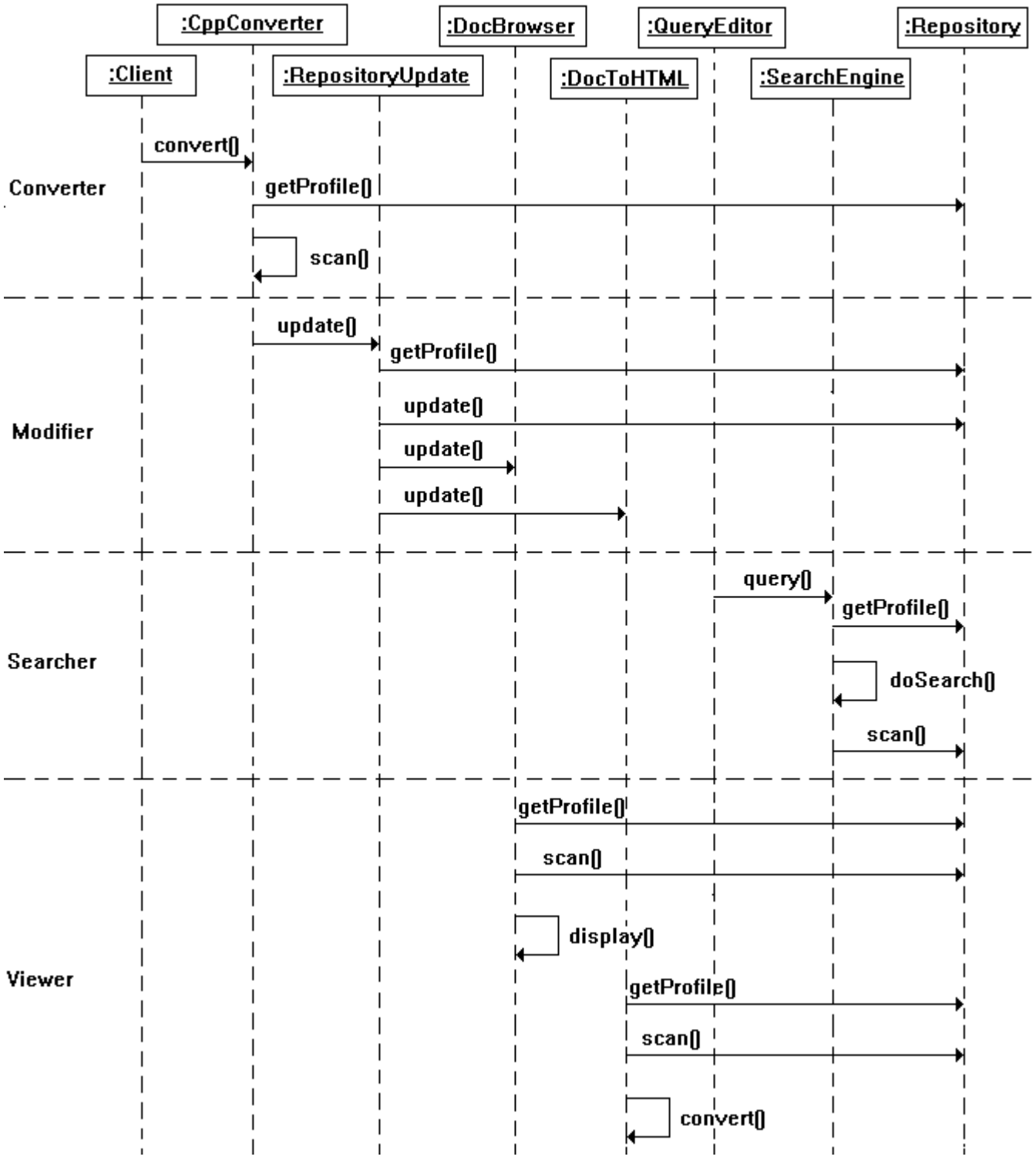

Figure 7: Sequence Diagram for Converters, Modifiers, Searchers and Viewers

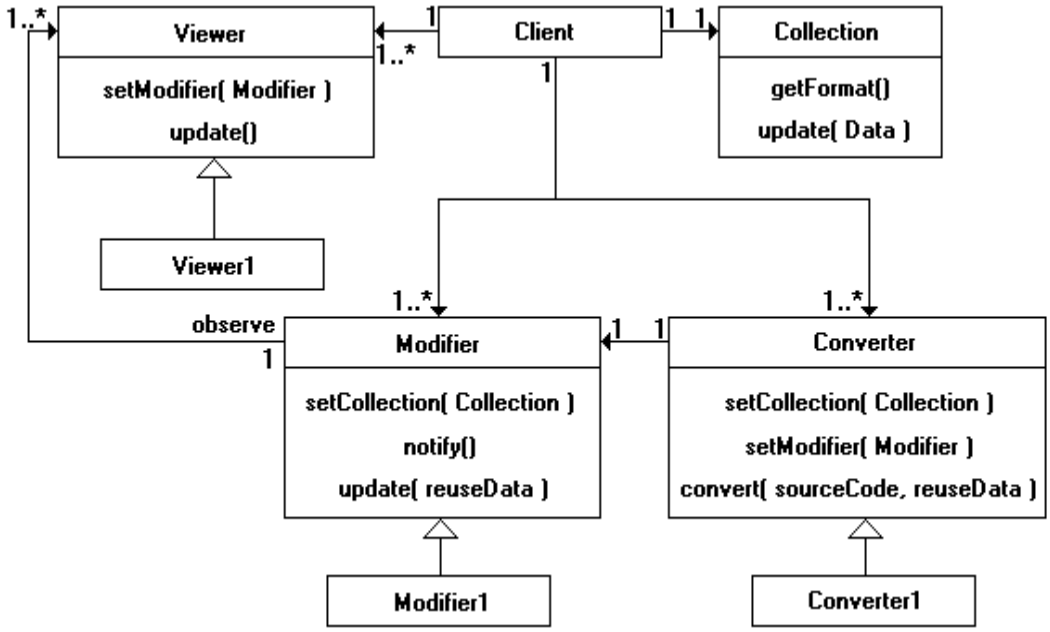

Figure 8: Modifier Design Pattern

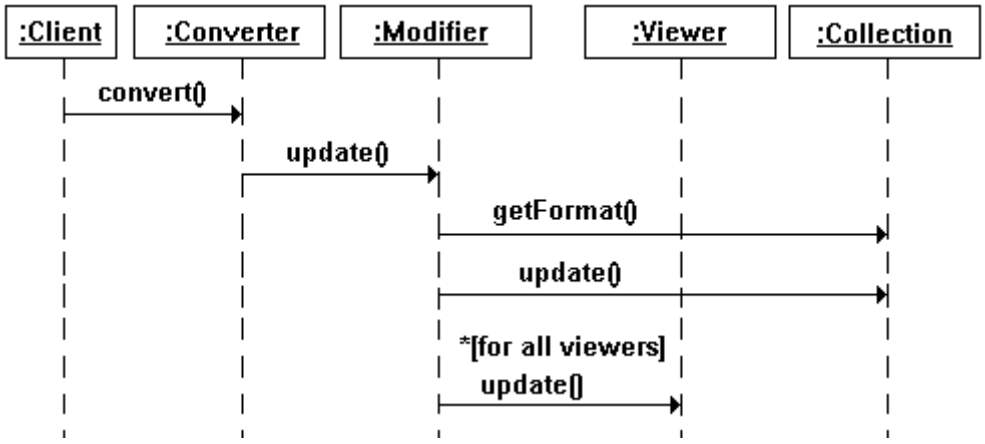

Figure 9: Modifier Sequence Diagram

- **Viewer:** Observes the modifiers for any changes made to the collection.
- **Converter:** Provides the modifiers with new reuse information to be added to the collection.
- **Collection:** Provides the interface to access information stored in the collection.

Collaborations:

The sequencing is shown in figure 9.

- `Converter` sends the new data to `Modifier` via `update`.
- `Modifier` obtains the desired data storage format from `Collection` and then proceeds to update `Collection`.
- When the update is complete, `Modifier` notifies `Viewer` of the changes.

Consequence:

The main benefits of using this pattern are:

- Data integrity is maintained due to the restriction to a single point of entry.
- Data consistency is maintained since the correct storage format can always be enforced.

The main disadvantage of using this pattern is:

- Access to the data is dependent on the flexibility and power of the modifier.
- A poorly designed modifier may allow unauthorized access to the repository thereby becoming a security risk.

Implementation:

Modifiers have the duty of updating the collection with information provided via the converters. Hence, the `Converter` gathers the reuse data and passes it to the `Modifier` via its `update` method. The modifier then updates the collection with the new information and informs the viewers of these changes via their `update` methods.

```
void Modifier::update( Location reuseData )
{
     ReuseDataPackage *data;
     ConfigurationPackage *config;

     // Get the new reuse data
     data = getReuseData( reuseData );

     // Consult the collection to determine how the information
     // is to be formatted and stored
     config = collection->getFormat();

     // Format the data
     formatReuseData( data, config );

     // Add formatted reuse data to the collection.
     collection->update( data );
```

```
        // Notify all viewers of the change.
        for( count = 0; count < getTotalViewers(); count++ )
        {
             ViewerList[ count ]->update();
        }
    }
```

The design pattern uses a variation of the Observer design pattern where the modifier is the subject and the viewers are the observers [8]. This ensures that the modifier informs the viewers of any repository changes. However, the necessity for notification is dependent on the function of the viewer. For example, a viewer that converts the data to the HTML format may be executed at the start of each week, removing the need for any notification. In general, a single modifier is created for a single collection. However, if there is more than one type of collection, such as multiple collections distributed across the Internet, then an Abstract Factory class will be needed for generating the correct modifier class instances.

Known Uses:

A single modifier called RepositoryUpdate, is used to add the reuse information generated by the converters to the ROOC repository [7]. This simplified the design of the converters since they generated all possible information, leaving data selection to the modifiers. Figure 7 contains the sequence diagram of the RepositoryUpdate utility. After receiving an `update` message, the RepositoryUpdate tool obtains a profile of the repository by sending a `getProfile` message to the repository. This allows RepositoryUpdate to determine what information is currently being managed by the repository and therefore to ensure that only the desired attributes and relationships are added.

The information within the ROOC repository is based on an ASCII format. This satisfies the design requirement that the reuse data must be accessible to as many tools as possible. However, this complicated the modifying and indexing of the repository by the RepositoryUpdate tool. To simplify this procedure, a batch mode was implemented, allowing the updating of one or more components in a single round.

6 The Searcher Pattern

Querying of the collection is performed via searchers. On receiving a query, the searchers first consult the configuration objects to determine how the information in the component objects is to be matched. Hence, the collection is searched in a consistent manner by any querying language. For example, when a query is submitted to the search engine, the component objects are initially queried in order to determine their contents. If the component object acknowledges that it has an instance of the specific attribute or relationship then the respective configuration object is asked to perform the match. The results are then returned to the search engine for further processing and display.

Motivation:

Many database systems provide multiple methods of developing queries that use scripting languages

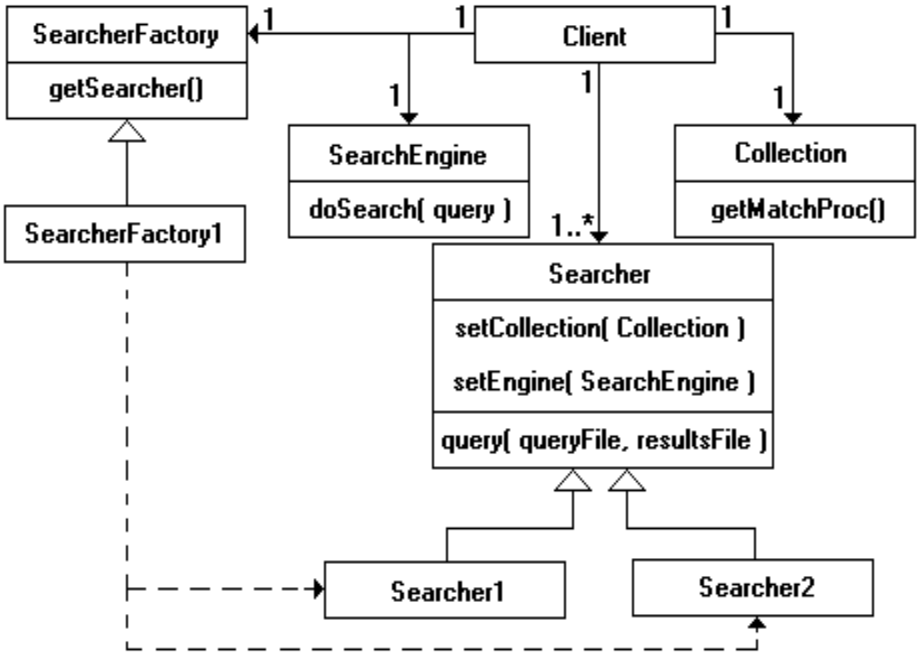

Figure 10: Searcher Design Pattern

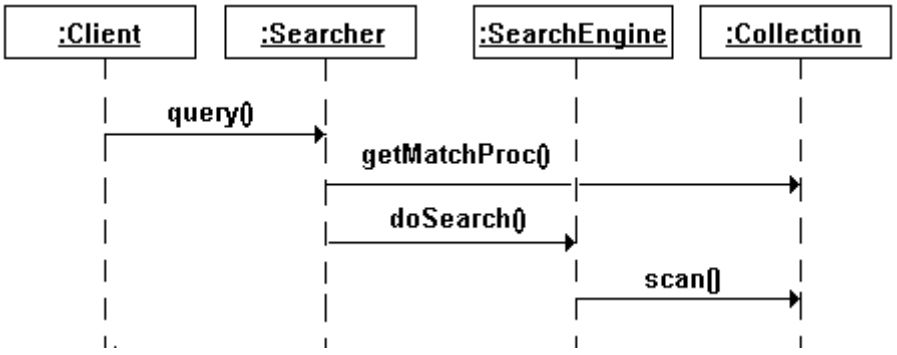

Figure 11: Searcher Sequence Diagram

and graphical user interfaces. In many cases, the multiple forms are converted to a single common form before the search commences. For example, the graphical representation of the query is usually converted into its scripting form. In the case of reuse, several different query languages may be provided to suit different user tastes and experience as well as the current context in which the search is being performed. In addition, the searcher pattern requires all search engines that execute a given query to determine the results in a consistent way. Hence, information that is used for determining the result must be used before the query is performed.

Structure:

The structure is shown in figure 10.

Participants:

The pattern uses:

- **SearcherFactory:** Generates the required searchers based on the given collection type and the query mechanism.
- **Searcher:** Provides the interface for the searcher classes.
- **SearchEngine:** Performs the actual scanning of the collection based on requests from the searcher.
- **Collection:** Provides the interface to access information stored in the collection.

Collaborations:

The sequencing is shown in figure 11.

- `Searcher` receives a query via the `query` message.
- The matching procedure is then obtained from `Collection` by `Searcher`.
- `Searcher` converts the query into a format understood by `SearchEngine` and submits it via `doSearch`.
- `SearchEngine` then scans `Collection` and returns the results to the client.

Consequence:

The main benefits of using this pattern are:

- The repository maintains its query language independence.
- Consistent results are generated independently of the query language and search engine.
- Multiple methods for the exploring of the collection can be used.

The main disadvantage of using this pattern is:

- Query language functionality may be constrained by the underlying matching procedures within the collection.

Implementation:

The user query is passed to the `query` method in the form of a `queryFile` location. The location for the results are given in the `resultsFile` parameter. The actual query is then converted into a form understood by the search engine and passed to the engine for scanning. This conversion is necessary in order to facilitate reuse of the database scanning class, SearchEngine.

```
void Searcher::query( Location queryFile, Location resultsFile )
{
     ConvertedQuery *query;
     QueryResultsPackage *results;

     query = convertToEngineFormat( queryFile );
     results = SearchEngine.doSearch( query );
     displayResults( results, resultsFile );
}
```

Known Uses:

The ROOC system provides a query language and a search engine for querying the repository. The query language is called the **S**cripting Language for **O**bject-**O**riented **C**lasses (S-ROOC) and is an interpreted object-based language, consisting of a set of predefined operators, standard control constructs for iteration and selection, and predefined objects such as a `Report` object for generating reports [6]. A script editor and parser called QueryEditor is provided with the system for the development of S-ROOC programs. All queries are submitted to the search engine, SearchEngine, which in turn scans the repository and produces the desired reports.

The Attribute Objects within the repository provide a set of rules that state how the attributes and relationships within the repository are to be matched during a query. This enables comparable results to be produced for similar queries created with different query languages. In other words, this mechanism ensures consistent results among different search engines. Figure 7 shows the sequence diagram for a ROOC query. The QueryEditor utility sends a query to the search engine via the `query` message. The search engine then obtains the profile in order to determine the required matching procedure. It then scans the repository and returns the results to QueryEditor for viewing by the user. Note that simpler, graphics-based query tools are also provided with the system. These tools allow the creation of queries using more intuitive, graphical methods.

The expressiveness and complexity of the query language is dependent on the amount of work performed by the search engine. Since the matching procedures are determined by the Attribute Object rules, the development time and effort required for the creation of the search engine was considerably reduced.

7 The Viewer Pattern

Views are generated by viewers and describe how collection data is extracted and displayed. The content, type and form of information handled by each viewer depends on its primary function and the available information. To this extent, a viewer may be linked to a modifier where the viewer is the observer and the modifier is the subject. When the modifier makes a change, it notifies the corresponding views to display

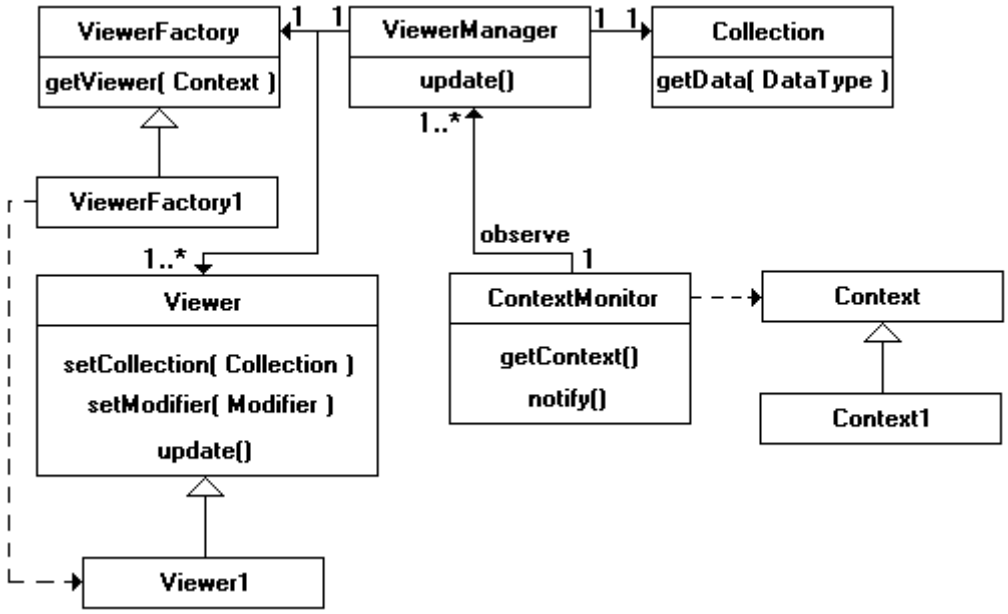

Figure 12: Viewer Design Pattern

the change. This is similar to the Model-View-Controller paradigm in Smalltalk where the model informs the view of any changes to itself. In general, a viewer acts as a filter, transforming the collection data into various forms.

A view is also context sensitive since its use depends on the current context occupied by the developer. For example, a view may provide information that is suitable only during the actual use of a class. Another example consists of a developer attempting to locate one or more classes that conform to specific requirements. In this case, a tree-like view for the exploration of categories of classes or classes sorted by keyword, could be used. The profile determines whether or not the information required by the view is available. If some of the information is missing, the view may be reduced to a subset of its total form. However, if all required information is missing, the view cannot be used.

Motivation:

During the development process, the developer will require various views on the underlying reuse information. The view used is determined by the information needed at that point in time. For example, the user may require that the components be listed in alphabetical order or sorted in specific categories. This pattern allows the context to be determined and the required view generated.

Structure:

The structure is shown in figure 12.

Participants:

The pattern uses:

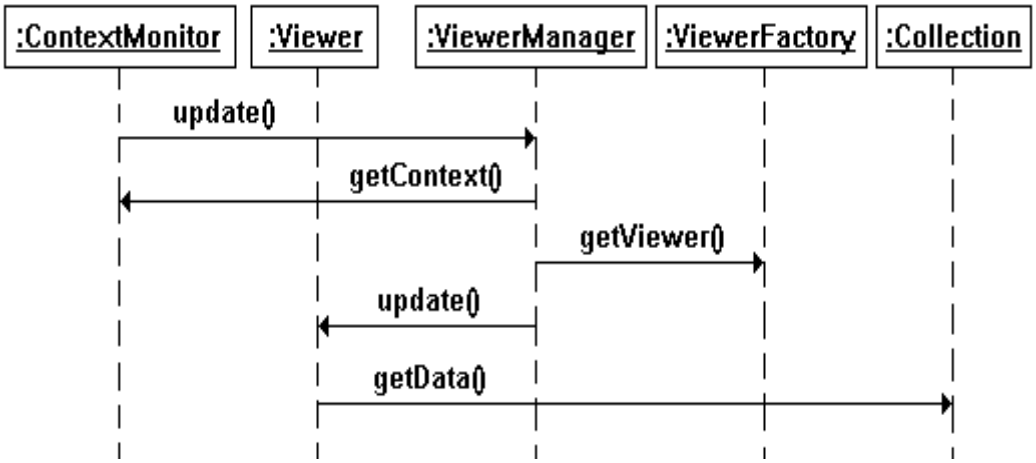

Figure 13: Viewer Sequence Diagram

- **ViewerFactory:** Generates the required viewers based on the context sent via its `getViewer` method.
- **ContextMonitor:** Monitors the interaction of the client with the system to determine what is the current context. If the context changes the `ViewerManager` class is notified via its `notify` method.
- **ViewerManager:** Observes the state of the `ContextMonitor` class for any changes in the context. If the context is changed then a new context is obtained via the `getContext` method of the `ContextMonitor` class.
- **Context:** Contains the details about the current context.
- **Viewer:** Provides the interface for the viewer classes. It also provides the `setModifier` method for informing the viewer of the current modifier.

Collaborations:

The sequencing is shown in figure 13.

- When the context changes, `ContextMonitor` informs `ViewerManager` of the change.
- `ViewerManager` then obtains the new context from `ContextMonitor`.
- `ViewerManager` obtains the appropriate `Viewer` from `ViewerFactory`.
- The new `Viewer` then displays the data relevant to the current context.

Consequence:

The main benefit of using this pattern is:

- Information overload is avoided by displaying only the required information.
- Multiple methods for displaying the data within the collection can be used.

The main disadvantage of using this pattern is:

- The information displayed by a viewer is restricted to the data within the collection.

Implementation:

The `ViewerFactory` class provides the interface for generating the requested viewers.

```
class ViewerFactory
{
     ...
     Viewer *getViewer( Context *aContext );
};
```

The `ContextMonitor` class monitors the interaction of the client with the system. When any changes occur, the `ViewerManager` classes are notified through their `update` methods.

```
void ContextMonitor::monitorClient( void )
{
      ViewerManager *aManager;
      int count;

      while( true )
      {
           if( changeInClient() )
           {
                for( count = 1; count < getTotalManagers(); count++ )
                {
                     aManager = getViewerManager( count );
                     aManager->update();
                }
           }
      }
}
```

The `ViewerManager` class then queries the `ContextMonitor` class via its `getContext` method, for the new context.

```
void ViewerManager::update( void )
{
     CurrentContext = ContextMonitor->getContext();
     ...
}
```

This is then passed to the `ViewerFactory` class via its `getViewer` method and the appropriate viewer is returned.

```
void ViewerManager::update( void )
{
     CurrentContext = ContextMonitor->getContext();
     CurrentViewer = ViewerFactory->getViewer( CurrentContext );
}
```

This design pattern is a variation of the Observer design pattern where the `ContextMonitor` class is the subject and the `ViewerManager` is the observer. As explained in the previous section, the viewers

also monitor the modifiers so that changes to the collection are immediately noticed. Again, for this type of monitoring the most suitable design pattern is the Observer pattern where the viewers are the observers and the modifier is the subject.

The use of `ContextMonitor` ensures that the correct viewer is activated at the appropriate time. However, it is not uncommon for a user to select the desired viewer when needed. In many cases, the reason for using a viewer may be a matter of taste instead of a formal context and so, an inadequate implementation of `ContextMonitor` will be seen as a hindrance.

Known Uses:

Several browsers or viewers are provided with the ROOC system [7]. They performed two basic functions: The global viewing of the data and the conversion of the data into standard formats such as XML. The main ROOC viewer is called DocBrowser and allows the browsing of the underlying data on a class basis. Tools for converting D-ROOC information to HTML, XML and TEX formats are also provided. For example, the DocToHTML tool provides a set of facilities for the creation of index, inheritance and individual class HTML web pages. A viewer can query the repository to determine what information is currently stored within the repository. This means that a viewer does not have to scan the entire repository when determining if the query can be satisfied. Figure 7 shows the sequence diagram for both the DocBrowser and DocToHTML viewers. In both cases, the repository profile is obtained before proceeding with the scan.

The creation of the viewers involved the determination of what information will be required during the reuse process. For example, with the rapid rise in Internet usage, the DocToHTML utility was one of the first tools to be created. Later, a DocToXML tool was created in anticipation of the increase in XML usage.

The Model-View-Controller provides similar functions where any changes to the data are reflected in the views [16]. However, the MVC leaves the user to determine which views to display and hence, there is no need for a context class. Another example is the Eiffel short form which allows Eiffel programs to be displayed without their implementations [14, 15].

8 Concluding Remarks

The Collection-View Model is similar to the Model-View-Controller of SmallTalk. It contains views for the underlying information and these views can be notified when changes to the data take place. The CVM also maintains a distinct separation between the underlying data and the frameworks that access and modify the data. In addition to gathering information, the CVM provides several other frameworks for searching and updating the repository. By separating these activities into various frameworks, the repository remains independent of any programming or querying language.

The separation between the collection itself and the independent frameworks that access and modify the collection offers several advantages for the Collection-View Model:

- *Integrity:* A protective layer via frameworks between the user and the collection.

- *Consistency:* Standardised treatment via configuration objects of class attributes and relationships, and standardised updates via modifiers to the collection.
- *Extensibility:* Programming and query language independence via converters and searchers respectively.
- *Maintainability:* Separation of specific reuse activities into corresponding frameworks.

However, the initial effort and cost of implementing the ROOC system were found to be high. Firstly, a greater coding effort was required since the D-ROOC format requires specialised document comments to be embedded within the code. Another important factor was the standardisation of the reusable attributes and relationships. This determined the final structure and design of the ROOC tools and the repository format. The main risk was that unnecessary information could be captured or important data missed, hence requiring extensive changes to the system. However, it was soon discovered that it was impractical to initially state all of the required reusable attributes and relationships. In fact, the standardisation of the stored reuse information was a continuous process.

In conclusion, software reuse must be entrenched as part of the normal software development process of both smaller and larger software houses and information technology departments. Because the Collection-View Model is designed to support the development of affordable and accessible software reuse systems such as ROOC and does so without the compromise of a weak querying language or an enemic search engine, software reuse gains reach a wider and more productive audience.

9 Acknowledgements

We would like to offer many thanks to Russ Rufer, our PLoP 2K shepherd, for his invaluable guidance during the shepherding process.

References

[1] Barker H. A., Grant P. W., Jobling C. P. and Townsend P., The object-oriented paradigm: A means for revolutionising software development, *IEE Computing and Control*, Vol. 4, No. 1, pp 10-14, Feb. 1993.

[2] Basili V. R., Briand L. C. and Melo W. L., How reuse influences productivity in object-oriented systems, *Communications of the ACM*, Vol. 39, No. 10, pp 104-116, Oct. 1996.

[3] Bergner K. and Rausch A., A componentware development methodology based on process patterns, *Proceedings of the 5th Conference on Pattern Languages of Programs*, 1998.

[4] Campione M., Walrath K. and Huml A., *The Java Tutorial Continued: The Rest of the JDK*, Addison Wesley, 1998.

[5] Depradine C.A., Patrick B.G. and Posthoff C., A document format for the reuse of object-oriented classes, *Proceedings of the Third Conference of the Faculty of Pure and Applied Sciences*, University of the West Indies, pp 26-27, Jan. 1997.

[6] Depradine C.A. and Patrick B.G., A scripting language for the reuse of object-oriented classes, *Proceedings of the Fourth Conference of the Faculty of Pure and Applied Sciences*, University of the West Indies, pp 53-55, Jan. 1999.

[7] Depradine C.A., A Software Tool for the Reuse of Object-Oriented Classes, Ph.D. Thesis, University of the West Indies, Oct. 1999.

[8] Gamma E., Helm R., Johnson R. and Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[9] Goguen J., Nguyen D., Meseguer J., Luqi, Zhang D. and Berzins V., Software component search, *Journal of Systems Integration*, Vol. 6, No. 1, pp 93-134, 1996.

[10] LaLonde W. R. and Pugh J. R., *Inside Smalltallk (Volume 1)*, Prentice Hall, 1990.

[11] Loomis M. E. S., ODBMS myths and realities, *Journal of Object-Oriented Programming*, Vol. 7(4), pp 77-80, July/Aug. 1994.

[12] Loomis M. E. S., Querying object databases, *Journal of Object-Oriented Programming*, Vol. 7(3), pp 56-60, June 1994.

[13] Meyer B., *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice Hall, 1994.

[14] Meyer B., Lessons from the design of the Eiffel libraries, *Communications of the ACM*, Vol. 33(9), pp 68-88, Sept. 1990.

[15] Rist R. and Terwilliger R., *Object-Oriented Programming in Eiffel*, Prentice Hall, 1995.

[16] Rumbaugh J., Modeling models and viewing views: A look at the model-view-controller framework, *Journal of Object-Oriented Programming*, Vol. 7(2), pp 14-20, May 1994.

[17] Stroustrup B., *The C++ Programming Language*, Addison Wesley, 3rd Edition, 1997.