Computer-Aided Reuse Tool (CART)

Zina HOUHAMDI
Department of Computer Science, University of Biskra, BP 145, Biskra, 07000, Algeria.
E-mail: z_houhamdi@yahoo.fr

Belkacem ATHAMENA
Department of Automatics, University of Biskra, BP 145, Biskra RP, 07000, Algeria.
E-mail: b.athamena@caramail.com

Keywords: Software reuse, facet classification, object-oriented library, thesaurus, CASE tools.
Received: 17/04/2004

Software reuse has been claimed to be one of the most promising approaches to enhance programmer productivity and software quality. One of the problems to be addressed to achieve high software reuse is organizing databases of software experience, in which information on software products and processes is stored and organized to enhance reuse.

Object-oriented software libraries expand in size more rapidly than other type of software library. This paper presents a simple approach for aiding reuse in software development using object-oriented library. Our approach improves the effectiveness of code searching by reorganizing the library with facet classification scheme and thesaurus. Information in specification models, such as Data Flow Diagrams (DFDs), is extracted through object abstraction and then used as a query input. We are currently implementing a Computer-Aided Reuse Tool (CART) based on the approach.

1. Introduction
Improving software productivity and quality is one of primary emphasis of research in software engineering [12,2]. Many studies have shown that software reuse can improve software productivity and quality significantly. Research on software reuse can be divided into at least three categories: some researchers study how to construct reusable software components [4,6,11,13], some study system frameworks for reusable software asset libraries [8,9,16], other study classification and retrieval strategies for reusable software libraries to achieve effective reuse [3,5,7].

Computer-Aided Software Engineering (CASE) tools usually come with a repository for specification in different development phases. The repository is designed as a browser-like or query-reply tool for search and retrieval of specification contents (elements). These kinds of tools, however, provide less help in reuse than those booting direct search on a real software library. On the other hand, object-oriented libraries probably grow more quickly than any other type of software libraries. Traditional ways of searching for components, such as consulting a users manual, using a browser or examining the source code, not only waste time in searching, but provide little help in understanding particular components. This problem is even more serious when searching through a large object-oriented library, which may comprise millions lines of code or thousands of classes.

In this paper, we present an approach that integrates reuse techniques with an actual object-oriented library to promote reuse in CASE. Our approach boots an automatic mechanism on both classification and retrieval processes of software components. The classification scheme is basically the facet scheme from [10,16,17], and the retrieval mechanism is a query-reply. The classification is performed using the keywords of a thesaurus. The query inputs are generated by extracting information from the entities in specification models, such as a data flow diagram (DFD). A query is instantiated by a user, and its content can be selected by the users or generated automatically. Replies to queries improve reuse by helping users compare entities and components en the library more precisely. We are currently implementing a Computer-Aided Reuse Tool (CART) based on our approach.

Section 2 gives a brief review of the facet scheme. Section 3 discusses the classification process in detail and presents some interesting results of queries. Section 4 presents a design for an automatic code extractor, and section 5 describes the structure of our CART. Section 6 is the conclusion of the paper.

2. The facet scheme
The facet classification scheme was first proposed by Prieto-Diaz in [17]. The facet scheme has many advantages. In particular, it is suitable for collections of similar reusable components that are large, contain groups, and are growing continuously. A typical example is GTE Data Service’s AMP (Asset Management Program), which was introduced in 1991 [16].

A facet is a viewpoint toward software components. Viewpoints may include the functions the components perform, the objects they manipulate, the system types to
whish they belong, and so on. The value for a facet of a component is called facet value. The set of facet values facet descriptor, and a component may be understood through its facet descriptor.

In designing classification scheme, there are at least two factors that must be considered. First, the scheme needs a set of proper facets to represent essential features of a component. This set may be determined by referring to system requirements after classifying all components. Unnecessary or insufficient facets may cause some trouble in the classification process. Unnecessary facets may make the facet values hard to assign or make it impossible to index a component by facet descriptor. Insufficient facets may map many components to the same facet descriptor, so hat ambiguities present when locating and retrieving components. Second, each facet needs a set of proper facet values to represent all possible (distinguished) features, or indices, of components. Usually, the definition of facet values relies on domain analysis and expert knowledge. When the collection of components is large, it is difficult to predefine all possible facet values for further indexing. Thus, the facet scheme must be simplified for a large object-oriented library.

3. A facet scheme for large object-oriented library

3.1. The use of keywords

A word may have more than one meaning and two different words may have the same meaning. A thesaurus is used in vocabulary control. It helps to clarify concepts. Table 1 shows several examples. In the table, the words consume, feed, partake, and chew have the same meaning, eat. On the other hand, the word consume has two different meaning, eat and waste.

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eat</td>
<td>Eat, consume, feed, partake, chew</td>
</tr>
<tr>
<td>Waste</td>
<td>Consume, expend, exhaust, reduce, eat</td>
</tr>
<tr>
<td>Taste</td>
<td>Relish, enjoy, eat, try, sample</td>
</tr>
<tr>
<td>Fewness</td>
<td>Diminish, reduce, lessen, lack, need</td>
</tr>
</tbody>
</table>

Table 1: Partial Thesaurus

Using keywords in a thesaurus as the universe of facet values to simplify classification has the following advantages. First, it saves time in determining the universe of facet values, and thus simplifies the classification domains. When a domain is changed, the universe remains unchanged. A facet value is described with a set of keywords, of which each represents one distinct meaning. A facet descriptor described in this way is space-less, but easier to implement.

From the viewpoint of specification, usually more than one word is needed to describe the features of components on each facet. For example, the information on the function facet may be (copy, from), (copy, replace, all) or (copy, replace, from). Although the primitive for a component is called facet descriptor. The characteristics of a component may be described using its function is copy, these three components are all distinct. As another example, the object type facet of components may have the information (linked, list) or (double, linked, list), indicating that they manipulate lists of different kinds. These cases occur frequently, especially when processing a large collection.

The facet scheme in our approach is centered an automated thesaurus developed from [14]. Our facet scheme consists of three facets:

- Function refers to the function performed. Function names such as store, changeRequest, and updateString are used to generate facet values.
- Object type refers to the template of objects to which the method belongs. Therefore, type names such as set, bag, and textItemEditor are used to represent this facet.
- System type refers to domains which are functionally identifiable, application-independent, and usually include more than one component. The name of a domain, for example, collection-text or graphics-interface, may be used for the value.

Each component of the software (code) library is given (defined) a set of keywords as a facet value. These keywords are generated from information extracted from the above names. One example is the information (linked, list), where linked belongs to the three keywords relations, junction and combination, and list belongs to four keywords numeration, class, list and record. If (linked, list) is used to define a specific facet of a component, the internal representation of the facet is (relations, junction, combination, numeration, class, list, record). The next section describes how to proceed with classification. Section 3.3 describes a retrieval example to illustrate the capabilities of the facet scheme.

3.2. The classification process

Based on the previous discussion, we designed a Library Automation Classification System (LACS) to classify a large object-oriented library. Figure 1 is the overall data flow of our LACS.

In the LACS, each facet has an automatic classification procedure. Each classification procedure is designed according to the properties of components and the needs of each facet. The strategies for generating facet values from a sample object-oriented library, Smalltalk-80, are the following:

- Function facet: The method name can represent the rough meaning of a method, for example, copy, changeRequest, storeone, doesNotUnderstand, storeString and so on. Method names are quite useful in helping users to understand the methods. Thus, our procedure in the LACS first splits the method name and then passes each word through the
thecusaurus to get the corresponding keywords. For example, the name of the method changeRequest can be split into two words, change and request. The value of the function facet of this method is (variance, money, improvement, change, command, request, worship, inquiry).

- Object type facet: This facet refers to the template of objects to which methods belong. An object type is a class name in the library. The class name of the sample library can describe the features of a class. Therefore, the procedure for function facet can be modified slightly to get the values of this facet.

System type facet: The domain that methods belong to are defined in the user manual of Smalltalk-80. The domain names in the user manual are also passed through the thesaurus to get facet values.

1. Query: A user wants to find possible methods of displaying characters. The query is \([\text{display}, \text{character}, (), ()]\), where the empty cell “(“ means “don’t care”.

Reply: The reply displays possible methods in classes. There are 42 possible tuples retrieved. However, some tuples seem irrelevant to the user requirement. This is caused by insufficient strictness in the query constraints. A small amount of extra information (constraints) can reduce the size of reply. The next two queries will illustrate the outcome of using a more narrowly constrained query.

---

### Class Name | Method Name
--- | ---
ArihmeticValue | Sign
Behavior | ShowVariableMenu
Browser | ShowHierarchy
ChangerScanner | ScanClassExpression, ScanExpression
CharacterScanner | StopConditionFor
CompiledCode | SignExtend
ComposedText | DisplayFromCharacter, RightMarginForDisplay
ComposedTextView | DisplayFromCharacter, ShowSelectionBoxOn
GraphicsContext | DisplayCharacterOfIndex, RoundedDisplayCharacterOfInd
IOAccessor | OpenFileWriteOnly
LoopNode | Condition
MenuTracker | DisplayExtraInformation, DisplayExtraInformationFor
Parser | BlockExpression, Expression, PrimaryExpression, TypeExpression
TextCollector | Show
TextLines | RightMarginForDisplay
2. Query: The user has the same requirements as above, and he guesses the system type to be graphics. So, the Reply: There are seven tuples, which satisfy the query constraints. The additional information on the system type facet reduces the size of the reply from 42 to seven tuples.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CharacterScanner</td>
<td>StopConditionFor</td>
</tr>
<tr>
<td>ComposedText</td>
<td>DisplayFromCharacter</td>
</tr>
<tr>
<td></td>
<td>RightMarginForDisplay</td>
</tr>
<tr>
<td>DeviceFont</td>
<td>DisplayCharacter</td>
</tr>
<tr>
<td>GraphicsContext</td>
<td>DisplayCharacterOfIndex</td>
</tr>
<tr>
<td></td>
<td>RoundedDisplayCharacterOfIndex</td>
</tr>
<tr>
<td>TextLines</td>
<td>RightMarginForDisplay</td>
</tr>
</tbody>
</table>

3. Query: If the user guesses that the object type should be graphics, the query may become \((display, character), (graphics), (graphics)\).

Reply: there are two tuples, which satisfy the query constraints. The additional information on the object type facet reduces the reply from seven to two tuples.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphicsContext</td>
<td>DisplayCharacterOfIndex</td>
</tr>
<tr>
<td></td>
<td>RoundedDisplayCharacterOfIndex</td>
</tr>
</tbody>
</table>

Obviously, the replies to queries are determined mainly by the input information (constraints) of each facet. The more precise a user wants a result to be, the more useful information he must provide. This query-reply tool is not complicated, but it still requires that the user type in the details of the query.

4 Object abstraction during specification

4.1 Object abstraction

DFDs have been used widely for system specifications. During requirements analysis, an analyst may not know exactly whether a process in a DFD requires further decomposition. There were some rules to help make such decisions. For example, a process need not be decomposed if its specification can be completely described on a piece of paper or if it has only one input and output. Without enough information, system analysts always decompose processes by experience. This way may cause some processes to be decomposed too much and others too little. Both are undesirable.

Object abstraction of DFDs may provide one kind of help for the above problems. Intuitively, data flows in a DFD can be mapped to objects, data stores to objects, processes to the methods of objects, and external entities to the objects containing original functions as their methods [1]. The specification of an entity, such as a data flow, data store, or external entity, may come from other modeling tools, such as the Entity-Relationship Diagrams (ERD) used in structured analysis (SA) or the object model. During requirements analysis, an analyst may not know exactly whether a process in a DFD requires further decomposition. This way may cause some processes to be decomposed too much and others too little. Both are undesirable.

During requirements analysis, an analyst may not know exactly whether a process in a DFD requires further decomposition. This way may cause some processes to be decomposed too much and others too little. Both are undesirable.

The query information may come directly from the specification, instead of being entered by the user as in section 3.3. An automated tool for processing these queries can help users, when they are working on system analysis, system design, or programming. For example, if the reply indicates that the entity being worked has a matched (or say, qualifies) component in the library, the user may wish to stop decomposition. Therefore, referencing the replied information can speed up the design of the both logical and physical models. In addition, reusing qualified code naturally reduces implementation time.

Figure 2 is a sample DFD from [15]. The process Evaluation bounds violation has the target object blood, pressure temp and pulse. If the system type facet information is Monitoring System, a query \([\text{Evaluation}, \text{bounds}, \text{violation}], (\text{blood}, \text{pressure temp and pulse}), (\text{Monitoring, System})\) will be extracted to see whether the process Evaluation bounds violation has corresponding codes in the library. This query-reply can help an analyst decide whether to decompose the process or not.

![Figure 2: Partial Data Flow Diagram of SA](image)

Another example, shown in figure 3, is an object diagram in Rambaugh’s OOA [18]. The diagram is designed for a diagram editor. Let diagram editor be set as the system type facet information. A query \([\text{box}], ()\) will be extracted to check the existence of code for box. Another example, shown in figure 3, is an object diagram in Rambaugh’s OOA [18]. The diagram is designed for a diagram editor. Let diagram editor be set as the system type facet information. A query \([\text{box}], ()\),
4.2 An Automatic Code Extractor

An automatic Code Extractor (ACE) is an automated tool containing the functions discussed previously. An ACE simplifies the generating action of queries and makes replies more precise by integrating these actions with CASE tools. The more useful the information extracted from the repository is, the more precise the reply derived by an ACE will be. However, each CASE tool may follow a different methodology, such as SA/SD or OOA/OOD. The information extracted different methodologies will be different.

For a SA methodology, DFD is the kernel of SA. The function facet information can be extracted from the name string of a process. The object type facet information is extracted from the data flow name, data store name, or external entity name (which will be specified in the ERD). The system type facet information cannot be extracted from the repository directly. One way to handle this problem is to use an additional tool, like a customizer [19] in Excelerator/IS. To add an additional field to let users input the necessary information. For OOA methodology, the function facet information can be extracted from the service name. The object type facet information can be extracted from the class name or object name. The system type facet information also requires an additional field to let users enter the information.

The role of an ACE within the waterfall model is shown in figure 4. Our current prototype system uses a simple searching strategy, which contains the following steps:

1. Obtain the relevant data from the repository according to the user request and assemble the data into a query.
2. Replace the words in the query by keywords in the thesaurus to produce the real query.
3. Enter the query into classification catalog to get the reply and return it to the user.

If too many components satisfy the query constraints, the system acquires more information from the user interactively. If there is no hit, i.e., no component matches the query (constraints), the user has three choices. One is the user to accept the fact, another is to enter other information for the next query, and the other is to search through the class hierarchy for inherited methods.

Consider the example in figure 2 again. After the step 1, the query \( \{(\text{Evaluation}, \text{bounds}, \text{violation}), (\text{blood}, \text{pressure temp}, \text{pulse}), (\text{Monitoring}, \text{System})\} \) is extracted. After step 2, the keywords of the words in the above query are as shown in the following table and the real query is \( \{(\text{judgment}, \text{measurement}), (\text{limit}, \text{leap}, \text{promise}, \text{circumscription}), (\text{undueness}, ..., \text{impiety})), (\text{nobility}, (\text{rarity}, ..., \text{impulse}), (\text{measurement}, \text{heat}), (\text{regularity}, \text{oscillation})), (\text{inquiry}, (\text{unity}, ..., \text{arrangement}))) \} \).

The searching process of step 3 is through a union operation of the keywords of each word and then an intersection operation of the result of each word in the query. It is the same as \( \{(\text{judgment} \lor \text{measurement}) \cap (\text{limit} \lor \text{leap} \lor \text{promise} \lor \text{circumscription}) \cap (\text{undueness} \lor ... \lor \text{impiety}) \} \land (\text{nobility} \lor (\text{rarity} \lor ... \lor \text{impulse}) \lor (\text{measurement} \lor \text{heat}) \lor (\text{regularity} \lor \text{oscillation})) \lor ((\text{inquiry}) \lor (\text{unity} \lor ... \lor \text{arrangement}))) \} \).

<table>
<thead>
<tr>
<th>Words</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation</td>
<td>Judgment, measurement</td>
</tr>
<tr>
<td>Bounds</td>
<td>Limit, leap, promise, circumscription</td>
</tr>
<tr>
<td>Violation</td>
<td>Undueness, illegality, misuse, non-observation, overstepping, disobedience, impurity, impiety</td>
</tr>
<tr>
<td>Blood</td>
<td>Nobility</td>
</tr>
<tr>
<td>Pressure</td>
<td>Rarity, measurement, adversity, weight, influence, compulsion, propulsion, power, impulse</td>
</tr>
<tr>
<td>Temp</td>
<td>Measurement, heat</td>
</tr>
</tbody>
</table>
After step 3, there are no hits. The user chooses to accept this fact. This means that the process Evaluation bounds violation needs to be decomposed further.

The above searching process can narrow the range of possible components. The components in the reply should include at least one component the user needs if information related to user requests. The thesaurus is used for vocabulary control. The classification catalog is the result of code abstraction with enhanced facet scheme. The LACS described in section 3 is used while constructing the classification catalog. In CART, the role of the LACS is the maintenance of the classification catalog, such as the addition or deletion of software components. The dashed-square in figure 5 is the ACE; it is responsible for locating and retrieving components for users.

With these parts, CART can provide the closest related components in the library once the user (a system analyst, system designer, or programmer) makes a request. For example, when the user is specifying his requirements (or design) with DFD, he can ask the ACE to extract his previous specification as the raw query information. Through the help of automated thesaurus, the information is transformed into the query to search for possible components. We are still in the process of implementing CART. Our sample library is Smalltalk-80 and our CASE tool is Excelerator/IS.

6 Conclusion
This paper presents an approach to facilitate software reuse in case that is based on an object-oriented library. This approach boots the facet scheme, which automates the classification process from a (large) object-oriented library. The reuse process is automated with CASE tools to provide more assistance to the user in each phase of the software development cycle. We are still in the process of implementing CART. The next step in our research will be toward “knowledge-based CART”, where the knowledge may come from use habit. The system may improve the query by acquiring the knowledge while the user is working on the system.

References


