# INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**UMI**

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600


Université d'Ottawa · University of Ottawa

# DEPARS: Design Pattern Recognition System

by

**Te-Wei Sun**

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree of

**Master of**
**Computer Science**

under the Auspices of

the Ottawa-Carleton Institute for Computer Science

Computer Science Department

Faculty of Science

University of Ottawa

Ottawa, Ontario, Canada K1N 6N5

# Abstract

The industry has widely accepted the concept of design patterns to promote quality design reuse in the recent years. However, there are several problems preventing design patterns being used efficiently and effectively. The design pattern recognition system, DEPARS, discussed in this dissertation relieves these problems and promotes design pattern reuse. DEPARS recognizes patterns in object models by matching to templates in the knowledge base. DEPARS arranges the templates in the knowledge base in a hierarchy such that templates close to the root of the hierarchy are the bases of the ones below. The hierarchy reduces DEPARS's matching effort because it narrows the search area. DEPARS provides information about the recognized patterns to designers. This information helps designers to apply appropriate patterns in designs. DEPARS has pattern mining capability. DEPARS recognizes new patterns that may be reusable in the future from existing designs. In addition, DEPARS also facilitates designers verifying the recurrence of proto-patterns by storing the proto-patterns in the knowledge base. Once the proto-patterns are in the knowledge base, DEPARS can recognize them in future designs and hence shows the recurrence of the proto-patterns. The dissertation presents the design and operation of DEPARS. The dissertation also reports and discusses the evaluation results of DEPARS. The evaluation shows promising results indicating that DEPARS is adequate for practical use.

# Table of Content

# Table of Figures

# 1. Introduction

Design patterns have gained popularity in object oriented software design communities since Gamma, et al. introduced them in "Design Patterns" [Gamma et al., 95]. Design patterns offer many benefits such as promoting reuse of quality designs, improving design documentation quality, capturing design experiences for future reuse, facilitating the design of reusable software, increasing vocabularies for design discussion, and educating junior designers about quality designs. In essence, design patterns help designers to get a design "right" faster. However, there are problems in using design patterns and recognizing new design patterns. In order to maximize the use of design patterns, a designer needs to be able to find the right pattern efficiently and effectively from a pattern catalog, including those patterns unknown to the designer. This requires a sophisticated automated browsing mechanism that does not exist currently. A designer requires substantial knowledge of a pattern before he/she can apply it effectively in a design. Learning a new pattern may be a non-trivial task for a designer because of its complexities. Finally, recognizing new patterns from existing designs (also called pattern mining) and verifying the recurrence of proto-patterns[1] require abundant knowledge of the designs and expertise in patterns. This task is sometimes non-trivial even for a seasoned designer.

The design pattern recognition system (DEPARS) can help in relieving the problems stated above and hence promotes design pattern reuse. DEPARS recognizes patterns in the object structures in object model diagrams entered via an object modeling tool. Once DEPARS recognizes a pattern, it provides related information about the pattern to the user. The information includes, but is not limited to, the pattern's intended usage, structure and impact and examples of how the pattern is applied. DEPARS also proposes alternative patterns to users. If DEPARS does not recognize any pattern, it generates a new pattern based on the object model diagram and stores it for future

[1] A pattern that is not yet known to recur is sometimes called a proto-pattern. It is a pattern that has not yet undergo some degree of scrutiny or review by others.

use. DEPARS can also reverse engineer existing designs by recognizing patterns in their object models, which allows the user to understand the object models better and to reuse past experiences in the forms of new patterns.

## 1.1 Software Reuse

Software reuse has been regarded as the key in solving the software crisis. Software reuse is considered to improve software quality and productivity and hence reduces development and maintenance cost. A wide range of researches have been done to increase software reuse [Bott and Ratcliffe, 92].

Reusing software has three essential steps: searching, understanding and adapting [Biggerstaff and Richter, 87]. In order to use a software one must be able to find it first. One of the problems in software reuse is how to organize the collection of reusable software and the related description so that one can retrieve them without much effort. After finding a potentially reusable software, one has to study it in order to determine if it fully or almost fully satisfies the requirements. Not understanding how to reuse a software is another common problem preventing software reuse. DEPARS addresses both of these problems.

Software reuse is evident in different levels of abstraction ranging from code, component, system, to design. Recently, object-oriented design pattern concept has gained popularity. Design patterns not only incorporate all the advantage of object-oriented programming languages for code reuse, but also promote design and document reuse. The patterns solve specific design problems and make object-oriented designs more flexible, elegant and ultimately reusable. They help designers reuse designs by basing new designs on previous experiences [Gamma et al., 95].

## 1.2 Design Patterns

What is a design pattern? According to Gamma, et al. in "Design Patterns", a design pattern is a "core solution to a reoccurring problem in a context" [Gamma et al., 95]. It involves several communicating objects and classes, constructing the key design structure. Each object or class has its own roles and responsibilities in solving the

specific design problem. A design pattern is different from a library class, such as an array or a linked list, which can be reused as is. The most obvious difference is that a design pattern provides a generic solution that needs customizations to solve a design problem in a particular context. Hence a design pattern promotes design reuse instead of code reuse promoted by a class library. In "Design Patterns", Gamma et al. describe patterns using the following template:

- *Name*. The name of the pattern.
- *Intent*. The intent is a short statement that answers the following questions: What does the design pattern do? What are its rationale and intent? What particular design issue or problem does it address?
- *Also Known As*. Other well-known names for the pattern, if any.
- *Motivation*. The motivation illustrates a design problem with an example, and shows how the class and object structures in the pattern solve the problem.
- *Applicability*. What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?
- *Structure*. The structure shows a graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [Rumbaugh et al., 91].
- *Participants*. Participants are the classes or objects participating in the design pattern and their responsibilities.
- *Collaborations*. The collaborations describe how the participants collaborate to carry out their responsibilities.
- *Consequences*. How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

- *Implementation*. What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?
- *Sample Code*. Code fragments illustrate how the pattern may be implemented.
- *Known Uses*. Examples of the pattern found in real systems.
- *Related Patterns*. What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

An example of a pattern is given below. However first we need to explain the notations used to depict the pattern.

### 1.2.1 Object Modeling Technique

The classes of design patterns and their relations are represented by a notation based on the Object Modeling Technique (OMT) with slight modifications. Figure 1a shows the notation for a class. A rectangle with a class name in it represents a class. An aggregation relation between two classes represents the fact that every instance of a class "has" an instance of the other class. An arrow-headed line with a diamond at the base depicts the OMT notation for aggregation relation (see Figure 1b). OMT uses a filled circle to represent a "more than one" relation. In the case of Figure 1b, an instance of the class, Drawing, has more than one instance of the class, Shape. An association relation between two classes represents the fact that every instance of a class "uses" an instance of the other class. An arrow-headed line represents an association relation (see Figure 1c). In this example, an instance of the class LineShape uses an instance of the class Color. An association link in OMT is bi-directional [Rumbaugh et al., 91]. Gamma et al. change it to be directional for clarification. The OMT notation for class inheritance is a triangle connecting a subclass (Shape) to its parent class (LineShape) (see Figure 1e). Finally, a notation added by Gamma et al. is the creation relation. It indicates a class that instantiates another. It is represented by a dashed arrowheaded line. The arrow points to the class that is instantiated (see Figure 1d) [Gamma et al., 95]. In this example, the class CreationTool instantiates the class LineShape.

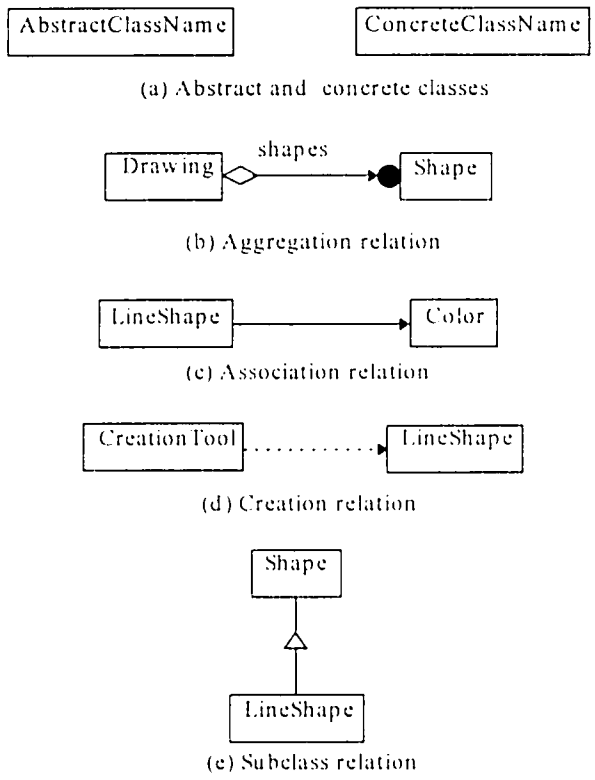



*Figure 1: OMT Notation representing classes and relations*

### 1.2.2 Design Pattern Example - Observer

*Observer*, included in "Design Patterns" [Gamma et al., 95], is a design pattern commonly used in designs of graphical user interfaces. The intent of the pattern is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

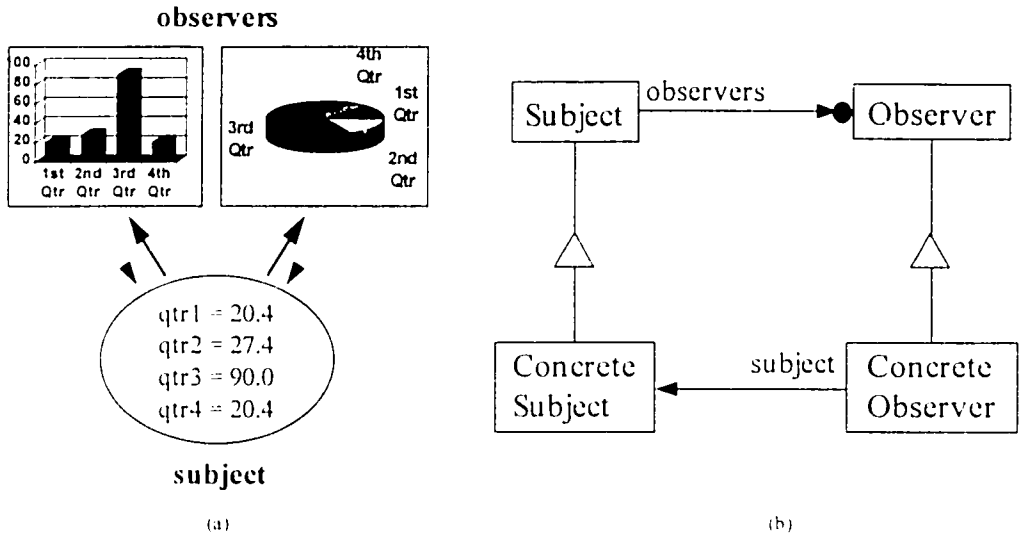



*Figure 2 Observer Pattern*

A common side effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. We do not want to achieve consistency by making the classes tightly coupled, because this reduces their reusability. Using a spreadsheet application as an example, both a bar chart object and pie chart object can depict information in the same application data object using different presentations (see Figure 2a). The chart objects are independent from each other. Thus they can be reused independently. However, they behave as though they know each other. When the user changes the information in the bar chart, the pie chart reflects the changes immediately, and vice versa. The behavior shows that the chart objects are dependent on the data object. Therefore the date object should notify the chart objects any change in its state.

Observer pattern describes how to establish relationships between classes to have that behavior. The pattern consists of four classes, Subject, ConcreteSubject, Observer and ConcreteObserver (see Figure 2b). The key classes are Subject and Observer. A subject may have any number of dependent observers. Subject provides an interface for attaching and detaching Observer objects. All observers are notified whenever the subject undergoes a change in state through the relation named observers. In response, each observer will query the subject to synchronize its state with the subject's state

through the subject relation. This pattern allows a designer to vary subjects and observers independently and still maintaining the consistency between the observers and to add observers without modifying the subject or other observers.

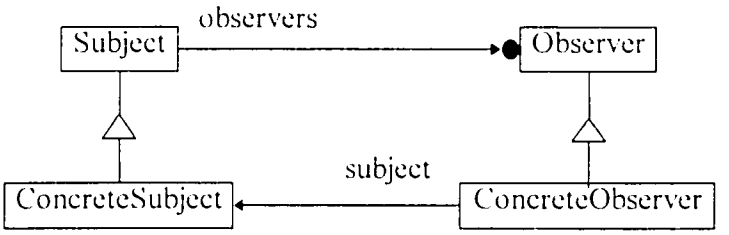
In our spreadsheet application example, the concrete subject is a spreadsheet holding the data and the concrete observers are different types of charts showing different views of the data. Whenever a user changes the data through one of the charts, the chart will request the spreadsheet to update its data. The spreadsheet (the subject) will then broadcast the changes to all dependent charts (the observers) so they can update themselves accordingly.

The following is how Gamma et al. documented Observer pattern in "Design Patterns" using the template:

- *Name*: Observer
- *Intent*: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- *Also Known As*: Dependents, Publish-Subscribe
- *Motivation*: A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
- *Applicability*: Use Observer pattern in any of the following situations:
    - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
    - When a change to one object requires changing others, and you don't know how many objects need to be changed.

- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

- *Structure*:




- *Participants*:
  - Subject
    - knows its observers. Any number of Observer objects may observe a subject.
    - provides an interface for attaching and detaching Observer objects.
  - Observer
    - defines an updating interface for objects that should be notified of changes in a subject.
  - ConcreteSubject
    - stores state of interest to ConcreteObserver objects.
    - sends a notification to its observers when its state changes.
  - ConcreteObserver
    - maintains a reference to a ConcreteSubject object.
    - stores state that should stay consistent with the subject's state.

- implements the Observer updating interface to keep its state consistent with the subject's state.

- *Collaborations*:
  - ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
  - After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

- *Consequences*:
  - The pattern lets you vary subjects and observers independently.
  - You can reuse subjects without reusing their observers, and vice versa.
  - It lets you add observers without modifying the subject or other observers.
  - Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.

- *Implementation*:

  Several issues related to the implementation of the dependency mechanism are discussed in this section.

  1. Mapping subjects to their observers. The simplest way for a subject to keep track of the observers is to store references to them explicitly in the subject. However, such storage may be too expensive when there are many subjects and few observers. Another solution is to trade space for time by using an associative look-up (e.g., a hash table) to maintain the subject-to-observer mapping. Thus a subject with no observers does not incur storage overhead. However, this approach increases the cost of accessing the observers.

  2. Observing more than one subject. It may make sense in some situations for an observer to depend on more than one subject. For example, a spreadsheet may

depend on more than one data source. It is necessary to extend the interface in such cases to let the observer know which subject is sending the notification.

3. Who triggers the update? The subject and its observers rely on the notification mechanism to stay consistent. But what object actually triggers the update? Here are two options:

   - Have state-setting operations on Subject trigger the update after they change the subject's state.
   - Make clients responsible for triggering the update at the right time.

4. Dangling references to deleted subjects. Deleting a subject should not produce dangling references in its observers. One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset their reference to it.

5. Making sure Subject state is self-consistent before notification. It's important to make sure Subject state is self-consistent before triggering the update, because observers query the subject for its current state in the course of updating their own state.

6. Avoiding observer-specific update protocols. Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument. At one extreme, the subject sends observers detailed information about the change, whether they want it or not. This might make observers less reusable, because Subject classes make assumptions about Observer classes that might not always be true. At the other extreme, the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter. This may be inefficient, because Observer classes must ascertain what changed without help from the Subject.

7. Specifying modifications of interest explicitly. You can improve update efficiency by extending the subject's registration interface to allow registering

observers only for specific events of interest. When such an event occurs, the subject informs only those observers that have registered interest in that event.

8. Encapsulating complex update semantics. When the dependency relationship between subjects and observers is particularly complex, an object, ChangeManager, that maintains these relationships might be required. ChangeManager's purpose is to minimize the work required to make observers reflect a change in their subject.

9. Combining the Subject and Observer classes. Class libraries written in languages that lack multiple inheritance (like Smalltalk) generally don't define separate Subject and Observer classes but combine their interfaces in one class. That lets you define an object that acts as both a subject and an observer without multiple inheritance.

- *Sample Code*:

```
class Subject;

class Observer {
public:
      virtual ~Observer();
      virtual void Update(Subject* theChangedSubject) = 0;
protected:
      Observer();
};

class Subject {
public:
      virtual ~Subject();
      virtual void Attach(Observer*);
      virtual void Detach(Observer*);
      virtual void Notify();
protected:
      Subject();
private:
      List<Observer*> *_observers;
```

```
};

void Subject::Attach(Observer* o) {
        _observers->Append(o);
}

void Subject::Detach(Observer* o) {
        _observers->Remove(o);
}

void Subject::Notify() {
        ListIterator<Observer*> i(_observers);
        for (i.First(); !i.IsDone(); i.Next()) {
                i.CurrentItem()->Update(this);
        }
}

class ClockTimer: public Subject {
public:
        ClockTimer();
        virtual int GetHour();
        virtual int GetMinute();
        virtual int GetSecond();
        void Tick();
};

void ClockTimer::Tick() {
        Notify();
}

class DigitalClock: public Widget, public Observer {
public:
        DigitalClock(ClockTimer*);
        virtual ~DigitalClock();
        virtual void Update(Subject*);
        virtual void Draw();
private:
        ClockTimer* _subject;
```

```
};

DigitalClock::DigitalClock(ClockTimer* s) {
        _subject = s;
        _subject->Attach(this);
}

DigitalClock::~DigitalClock() {
        _subject->Detach(this);
}

void DigitalClock::Update(Subject* theChangedSubject) {
        if (theChangedSubject == _subject) {
                Draw();
        }
}

void DigitalClock::Draw() {
        // get the new values from the subject
        int hour = _subject->GetHour();
        int minute = _subject->GetMinute();
        // draw the digital clock
}

class AnalogClock: public Widget, public Observer {
public:
        AnalogClock(ClockTimer*);
        virtual void Update(Subject*);
        virtual void Draw();
        //...
};
```

- *Known Uses*:
    - Smalltalk Model/View/Controller, Smalltalk, ET++, THINK, InterViews, the Andrew Toolkit and Unidraw.
- *Related Patterns*:

- Mediator: By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.
- Singleton: The ChangeManager may use Singleton pattern to make it unique and globally accessible.

### 1.2.3 Using Design Patterns

In order to use a design pattern, the designer has to first select a design pattern to use. There are several approaches described in "Design Patterns":

- Consider how design patterns solve design problems.
- Scan the intents of design patterns.
- Study how patterns interrelate.
- Study patterns of like purpose.
- Examine a cause of redesign.
- Consider what should be variable in your design.

Once a designer selects a pattern, he/she follows the steps listed below to apply a pattern in his/her design:

1. Study the pattern thoroughly.
2. Examine examples of how the pattern has been applied.
3. Choose names for pattern participants that are meaningful in the application context.
4. Define the classes.
5. Define application-specific names for operations in the pattern.
6. Implement the operations to carry out the responsibilities and collaborations in the pattern.

## 1.3 Problems of Using Design Patterns

There are several problems a designer may face when using design patterns: 1. There is a lack of a sophisticated browsing mechanism allowing the designer to search for the right pattern efficiently and effectively. 2. Learning how to apply a pattern and what are the consequences after applying may be non-trivial to some designers because of the pattern's complexities. 3. New patterns are difficult to detect.

Presently, there are three common ways of indexing patterns for the purpose of searching -- by name, purpose and problem category. Searching patterns by their names would be effective if the same patterns are always given the same names and if the designer knows the name of the particular pattern he/she is looking for. However, these two conditions are usually not met at the time a designer is searching for a pattern. First of all, it is common that a pattern has more than one name. For example, the "Observer" pattern is also known as "Dependents" and "Publish-Subscribe". The "Decorator" pattern is also known as "Wrapper". Unless these patterns are indexed under all their names, a designer may not be able to find them. Secondly, finding the right pattern by name is not effective, because the name of a pattern only conveys a very high level of abstraction about the pattern. A designer usually requires more detailed information than that to select a pattern.

In "Design Patterns", Gamma et al. divide patterns into three purpose groups - creational, structural and behavioral. With purpose groups, a designer can narrow the search for a desired pattern by first selecting a purpose group that matches his/her design intention, and then scanning through patterns in the group. This approach is slightly more efficient than the first one. However the designer may still need to scan through several patterns before finding the right pattern. The task could be very time consuming if we have to search through a large collection of patterns. Another down side of this approach is lack of formal way in defining new purpose groups.

Searching a right pattern by problem category is also ineffective considering that there could be many ways to describe the problem a pattern solves. Scanning the descriptions may also be time consuming.

In essence, these three indexing methods prevent effective and efficient searches because there is not a formalized approach in naming a pattern or describing a purpose or a problem.

Once a designer finds a potential design pattern, he/she must study the intended usage of the pattern and the impact after applying the pattern before deciding if the pattern is appropriate for the design. This is difficult sometimes for designers because the pattern is complicated. In addition to the documentation, examples of how to apply the pattern and references to related patterns also help a designer to understand the pattern well.

Creating a reusable pattern solving a general enough design problem requires vast amount of knowledge about object-oriented design and analysis and the problem domain. It may also require previous experiences of solving similar type of problems in order to detect the reoccurring pattern. Sometimes a new pattern goes by unnoticed because the designs using the pattern are too complex for the pattern to stand out.

## 1.4 Solutions Provided by DEPARS

In addition to the three ways of indexing patterns mentioned previously, DEPARS uses a fourth way to index patterns -- indexing by structure. A group of objects and classes and their relationships represent a design pattern's structure. Since each object or class carries its own responsibilities in solving the problem, the pattern structure outlines the solution. Indexing patterns by their structures is similar to indexing patterns by their solutions. The only difference is that notations used to depict the structures are limited. This simplifies the queries used to find a particular pattern.

DEPARS provides information about a pattern it recognizes. The information includes the intended usage, impact and usage examples of a pattern. It also contains references to similar and alternative patterns. The user can update or augment the information. With this information DEPARS is able to recognize a low quality design pattern and suggest the user other alternatives.

DEPARS is capable of simplifying a design by eliminating structural duplications such as multiple subclass branches and multiple identical relations in the object model. During the analysis phase, DEPARS applies matching rules to filter out duplications in the structure of a design. (See Sections 2.2.3.1 and 2.2.3.2 for details.) This helps DEPARS to detect the unique structural characteristics of the object model. DEPARS is also able to recognize combinations of patterns in a design. With these capabilities DEPARS can detect a new pattern containing only unique characteristics. The user can then examine the new pattern to decide its reusability.

In addition, DEPARS allows its users sharing information of a pattern by storing the information in a central location.

## 1.5 Reverse Engineering

Reverse-engineering, sometimes called re-documentation or design recovery, is a process to create a representation of a system at some higher level of abstraction or in some more precise notation [Hall, 92]. Reverse engineering is often needed when the existing documentation is inadequate in providing information required for maintaining the system or building a similar system. Several approaches have been implemented: outlining tools that list the functions contained within the software; cross referencing tools that show which procedures call which other procedures and use which data structures, just to name a few. DEPARS recognizes patterns from existing object model diagrams, providing a higher level of abstraction than the object model diagrams themselves. New patterns detected in the existing object model diagrams allow past experiences to be reused in the future.

## 1.6 Contributions of the Thesis

1. Organizing design patterns by their structures. As mentioned earlier in Section 1.3, design patterns are currently organized by their names, purposes or problem categories. Organizing design patterns by their structures provides a fourth alternative for searching.

2. Arranging design patterns in a hierarchy. Patterns, connected by deltas, are arranged so that a pattern always expands the one before. This narrows the search path for finding a pattern and hence speeds up DEPARS's analysis process.
3. A tool that recognizes design patterns in object models.
4. A tool that promotes proper usage of design patterns by actively displaying pattern information
5. A tool that facilitates pattern mining and proto-pattern verification. DEPARS facilitates pattern mining by detecting new patterns that are not in the knowledge base and expanding its knowledge base accordingly. Once a design pattern is captured in the knowledge base, DEPARS may detect it if it is used in object models. Users can use this information to verify the usefulness of a proto-pattern.
6. A tool that promotes design pattern reuse.
7. Empirical evaluations demonstrate that DEPARS is an effective pattern reuse tool.

## 1.7 Organization

In Chapter 2 other related works on software reuse and reverse engineering are discussed. Chapter 3 explains the architecture, design and implementation of DEPARS. Chapter 4 shows the result of the empirical evaluations and discusses the result's implication. Finally, Chapter 5 summarizes the dissertation and discusses the limitations of DEPARS and the future work for expanding DEPARS.

# 2. Design Pattern Recognition System

In order to demonstrate the stated benefits of DEPARS, I have implemented a working prototype. The prototype is developed in IBM Smalltalk. It has a complete API (Application Programming Interface) for integrating with an object-modeling tool, such as Argos by Versant.

## *2.1 DEPARS Design Rationales*

### 2.1.1 Problems of Recognizing Design Patterns

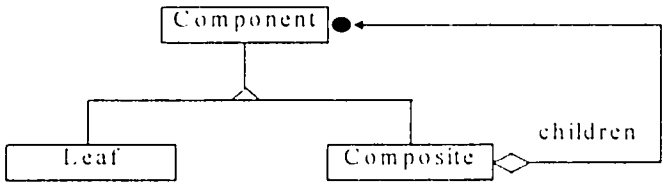



*Figure 3 Composite Pattern Consists of Component, Leaf and Composite*

Design patterns are design concepts, not rigorous mathematical definitions. A design pattern provides a generic solution that needs customizations to solve a design problem in a particular context. This means the actual implementations of a design pattern may not always be exactly the same. However, they still all share some common characteristics. For example, the Composite pattern described in "Design Patterns" composes objects into tree structures to represent part-whole hierarchies and lets clients treat individual objects and compositions of objects uniformly (see Figure 3). Composite pattern consists of three participants: Component, Leaf and Composite. Component is an abstract class that defines the uniform interface for objects in the one-to-many composition, i.e. the tree. Both Leaf and Composite are its subclasses. Leaf represents leaf objects in the composition. Composite represents objects in the composition with children, i.e. a subtree. It has a one-to-many aggregation relation to Component, which indicates a composite object may have several leaf objects or composite objects as its children. These participants and relations are the key of

Composite pattern and they are traceable in all actual implementations of Composite pattern. To recognize a design pattern, DEPARS must be able to recognize the design pattern's common characteristics, i.e. the pattern's key participants and relations, in object models. (For simplicity we will refer to these participants and relations as components henceforth.) Recognizing key components in object models is not easy. We need to solve several problems before we can achieve this goal. First of all, we must identify the key components of a design pattern. DEPARS needs them in order to know what to find in actual implementation. Second, even given the key components of a design pattern, exact matches of the key components and object models do not always guarantee to recognize the design pattern or identify all the objects and relations implementing the design pattern. We may need to generalize the content of an object model to increase the accuracy of matching. For example, it is reasonable to expect an aggregation relation in the object model matching an association relation of a design pattern, because an aggregation relation is considered to be a specialized association relation [Rumbaugh et al., 91]. Sometimes a class in a design pattern can be implemented as a class with several subclasses. It is also reasonable to expect a class and its subclasses in the implementation matching a class in the design pattern. However, if the class in the design pattern has subclasses, the subclasses in the object model should match the subclasses instead (see Figure 4). The other problem involves matching a part of a design pattern repeatedly. For example, the design pattern Iterator provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It is usually implemented with more than one pair of concrete aggregates and concrete iterators. All these pairs of concrete aggregates and concrete iterators should match the same Iterator pattern (see Figure 5). Finally, there may be more than one way to match an object model to a design pattern. DEPARS needs to be able to explore different possibilities and determine the best match.

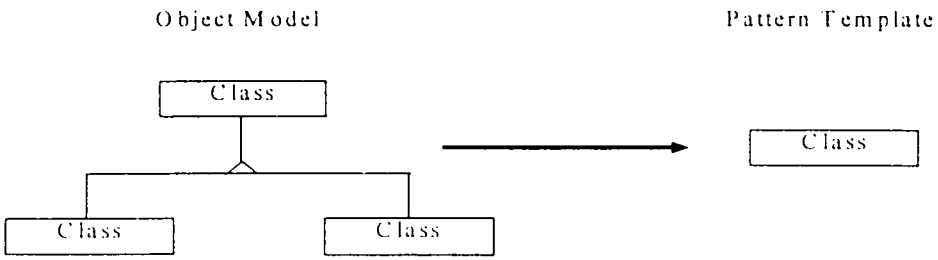



*Figure 4: A class hierarchy can be generalized in two different ways*

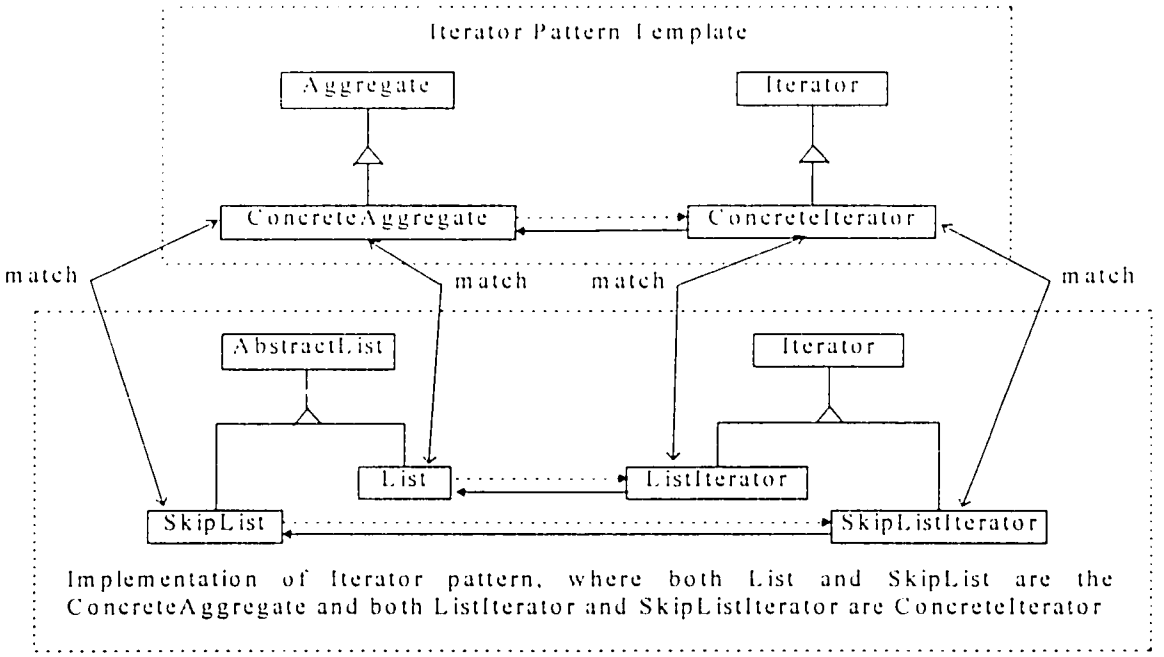



*Figure 5: The two pairs, List - ListIterator and SkipList - SkipListIterator, both match the pair ConcreteAggregate - ConcreteIterator in the Iterator template.*

### 2.1.2 Solutions to Problems of Recognizing Design Patterns

#### *2.1.2.1 Identifying Key Components*

The key components of an existing design pattern tell DEPARS what to look for in object models in order to recognize the design pattern. Identifying the key components is not a task for every designer. It often requires substantial amount of experience and knowledge about a design pattern. Fortunately, for most of the design patterns available today their key components are already a part of the documentation, since these design patterns are documented using the template proposed by Gamma et al. (see Section 1.2).

For those new design patterns whose key components are yet to be documented, DEPARS's new design pattern detection capability can facilitate the task. The detail of new design pattern detection capability is discussed later.

#### *2.1.2.2 Matching Key Components*

There are several scenarios where an exact match of the key components of a design pattern to object models is not sufficient and may lead to incorrect recognition. During the matching process, DEPARS applies several matching rules to increase its matching tolerance. See Sections 2.2.3.1 and 2.2.3.2 for details about the matching rules.

#### *2.1.2.3 Different Ways of Matching*

Sometimes a template may match an object model in several different ways. For example, given a template with three concepts A, B and C, both B and C inherit from A (see Figure 6). Matching the template to an object model containing identical structure, i.e. three concepts D, E, and F, and both E and F inheriting from D, may result in different matches. We can match the template to the object model by mapping A to D, B to E and C to F, or A to D, B to F and C to E. DEPARS keeps track of different ways of matching and adopts the one that causes the maximum number of components in the object model to be included in design patterns. See Section 2.2.3.3 for details on mapping.

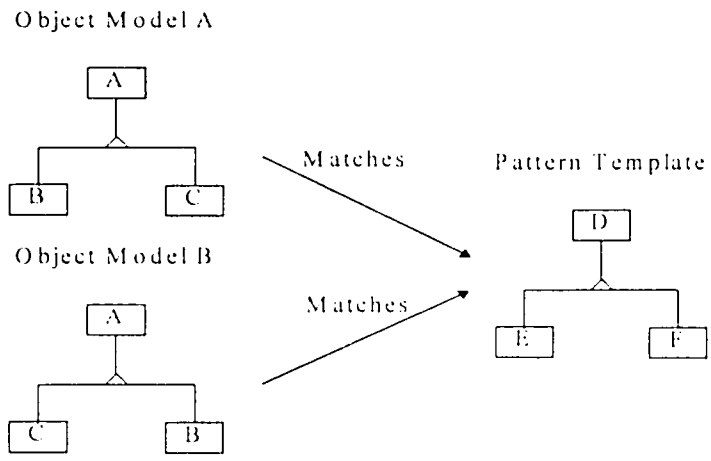



*Figure 6 Both object models A and B match the pattern template*

## 2.2 Architecture and Design of DEPARS

DEPARS's architecture consists of two parts, the ontology and the analyzer. The ontology is the hierarchical knowledge base of DEPARS. It stores known design patterns and sorts them by their structures. The ontology guides DEPARS during the design pattern recognition process. It also helps to shorten the time required to recognize a design pattern by narrowing the search area. The analyzer matches the given object models to the design patterns stored in the ontology and keeps track of the matching results.

### 2.2.1 Ontology

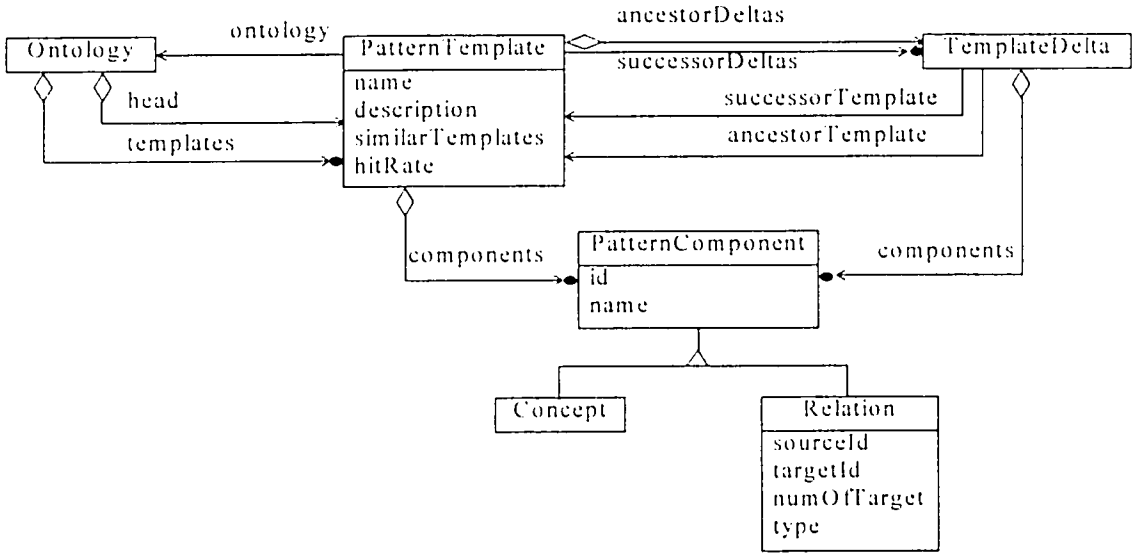



*Figure 7 The object model of the ontology*

The ontology represents DEPARS's knowledge of design patterns. The ontology consists of pattern templates (see Figure 7). Each pattern template represents a design pattern known to DEPARS. To maximize the reuse of design patterns, it is the best that the ontology to be shared by more than one user. Integrating DEPARS with an object oriented database system (OODBS) allows the users to share the same up-to-date knowledge base, especially useful when there are proto-patterns in the ontology. Proto-patterns are potential new patterns whose reusability has not yet been clearly demonstrated. DEPARS records the number of times it recognizes a template in object models. This number may help in determining the recurrence rate of a proto-pattern. The ontology also helps to see how templates relate to each other. When the key components of a design pattern are not identified correctly or completely, DEPARS may create other templates based on the design pattern. It is common for a proto-pattern not to have complete or correct key components. By examining the differences between the newly created templates and the design pattern, a user may be able to identify the missing or incorrect key components of the proto-pattern and modify it accordingly.

A pattern template stores a design pattern's name, structural information, intended usage, impacts and references to similar and alternative design patterns. The structural information consists of the key components of the design pattern, i.e. the concepts and their relations (see Section 2.1.2.1). All the components in a pattern template must be connected, since the collaborations among concepts are the essence of a design pattern. It is safe to assume that disjoined concepts or relations should not exist in a design pattern.

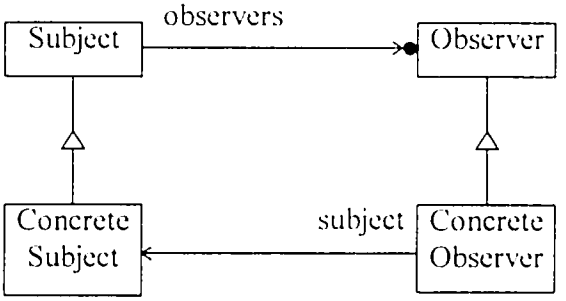



*Figure 8: The Observer pattern structure*

Each component in a pattern template is assigned a unique number, called an id. The id is used by DEPARS during the recognition process to map a component in a template to a component in an object model. The mapping allows DEPARS to know how a pattern template is matched to an object model. Each template component stores its name given by the creator of the pattern template. For example, the concepts in Observer pattern are named Subject, ConcreteSubject, Observer and ConcreteObserver respectively. The relation between Subject and Observer is named observers and the relation between ConcreteObserver and ConcreteSubject is named subject (see Figure 8). The name of a component is only used as a part of the documentation of the design pattern.

There are two types of components, concept and relation. A concept represents an object or a class in a design pattern. A relation represents the relation between two objects. A relation stores the unique id of the source concept, the unique id of the target concept, the cardinality of the relation and the type of the relation. Because a

relation refers to its connecting concepts by their ids, the maintenance effort of the ontology is greatly reduced. The indirect references also allow templates be modified independent from each other. This is especially important when DEPARS integrates with an object oriented database system (OODBS). When modifying an object stored in an OODBS, the object and the objects related to it have to be locked first to preserve data integrity. If the relations use direct references to their connecting concepts, modifying a relation may result in the whole ontology to be locked. In a multi-user environment, this means no other user will be able to access the ontology.

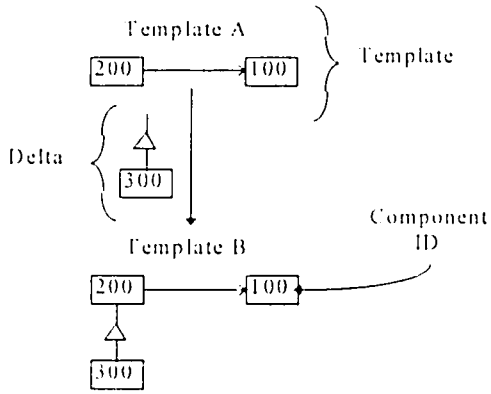



*Figure 9: Template A connects to Template B by a delta*

The pattern templates stored in the ontology are linked by deltas. A delta is the structural differences between two connecting templates. For example, in Figure 9 template A consists of two concepts with unique ids 100 and 200 respectively and an association relation. Template B, in addition to what template A already has, contains a concept with unique id 300 and a subclass relation connecting from the concept 200 to the concept 300. The differences between template A and template B are the concept 300 and the subclass relation. A delta consists of maximum of one concept and exactly one relation because all components of a pattern template must be connected by relations. Therefore while an additional relation may connect two existing concepts, an additional concept will always need an additional relation to connect it to the rest of the pattern template. The components stored in a delta are the same type of components stored in a pattern template. A template may connect to

several different templates via different deltas. The deltas are sorted based on their components.

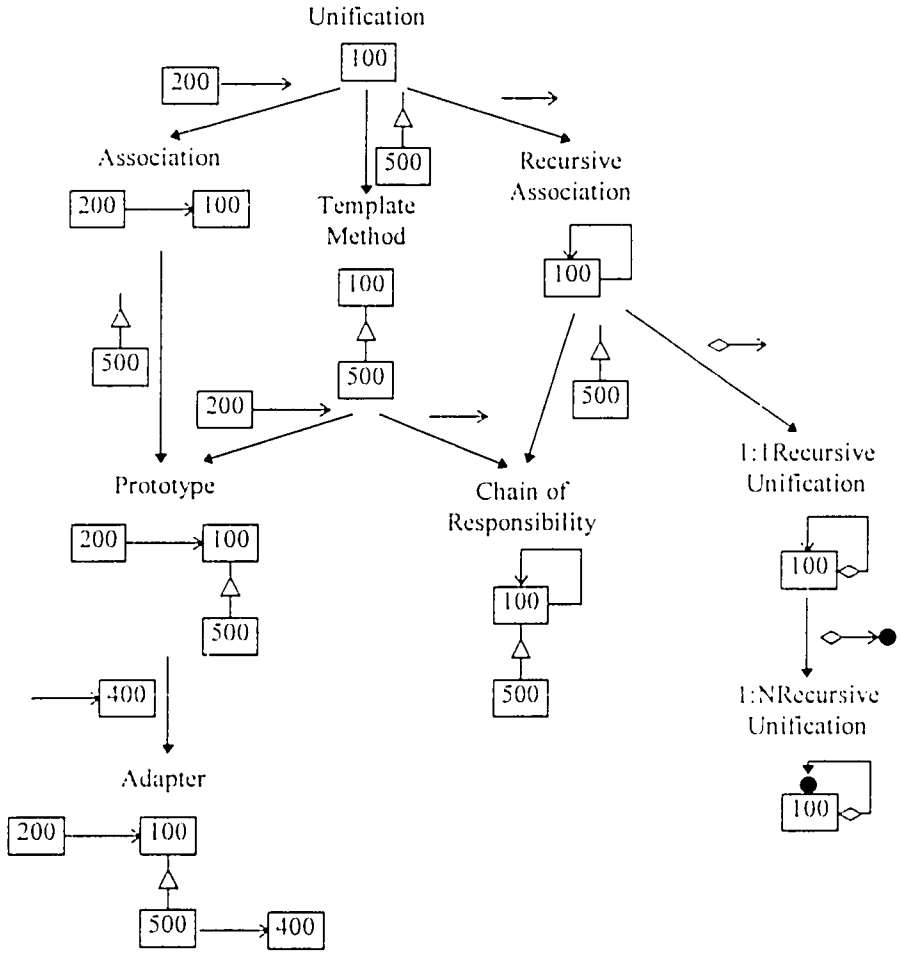



*Figure 10: An example of the ontology.*

The templates stored in the ontology are stored in a hierarchy based on their structures from the simplest to the most complicated (see Figure 10). The ontology starts with a template with only one concept and expands downwards as templates become more complicated. This arrangement allows DEPARS to do incremental matching. This means that if a template is matched, to match the next template connected by a delta only requires the delta to be matched. This arrangement also helps to narrow the search area for the target pattern. For example, if the Association template does not match the object model, 1:1 Connection template and Strategy template will not

match the object model neither, since the Association template is the precondition for matching these two templates (see Figure 10). Given this knowledge, DEPARS does not attempt to match these templates to the object model and the search effort is reduced.

### 2.2.2 Analyzer

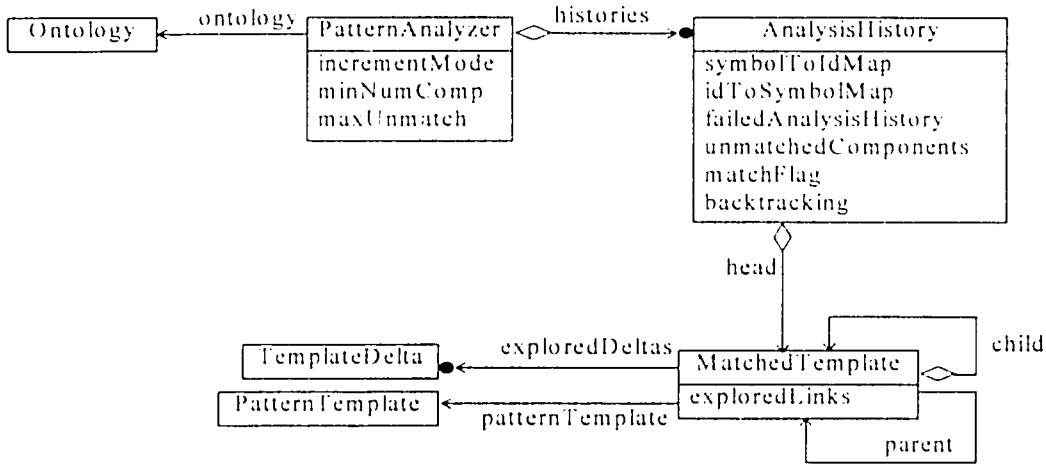



*Figure 11 The object model for the analyzer*

The analyzer is the control center of DEPARS. It is responsible for interfacing with an object-modeling tool, matching templates stored in the ontology to object models and keeping track of the matching results. It provides methods such as addConcept, addRelation, deleteConcept and deleteRelation, which are called by an object-modeling tool whenever a concept or relation is added or deleted in an object model. These methods all require the added/deleted component to be passed in as a parameter. DEPARS needs that information to perform appropriate actions. The analyzer also provides a method, matchNow, allowing the object modeling tool to demand DEPARS to start recognizing design patterns. The method is required since DEPARS can be operated in the non-incremental mode. In this operating mode, DEPARS does not analyze object models until it is instructed to do so. The analyzer comprises analysis histories and matched templates, which are delegated the responsibilities of matching templates and storing matching results (see Figure 11).

Section 2.1.2.3 mentioned how an object model can be matched differently against design patterns. In order to keep track of these different matching results, the analyzer stores these results in analysis histories. An analysis history represents a design pattern recognized by DEPARS. An analysis history maintains two mapping tables. One table maps a component of an object model to a component of a pattern template. The other table does the converse, mapping a component of a pattern template to that of an object model. The analyzer uses the information in these mapping tables to know how a design pattern is matched. In fact, DEPARS only needs one of the mapping tables to get the same information. The other table is added to improve performance. An analysis history also keeps track of the components of an object model that are not matched to components of a pattern template. New components added to an object model are stored in this unmatched component list of an analysis history. Hence DEPARS always attempts to match these components first. Since each analysis history represents a design pattern recognized in an object model, multiple analysis histories represent the combination usage of design patterns. Recognizing individual design pattern in each analysis history improves the scalability of the ontology. The analyzer only requires one template for each design pattern to recognize it in an object model, even when the design pattern overlaps with other design patterns. The only exception is discussed in Section 5.1.

### 2.2.3 Analysis Process

The analysis process has two goals. 1. Identify all participants of a recognized design pattern. 2. Cover the object model with a minimum number of design patterns. If the combinations of existing design patterns cannot cover the entire object model, the analyzer creates new design patterns to cover it.

DEPARS's analysis process is basically taking a component arrangement, traversing the ontology and trying to recognize a pattern template one delta at a time. A component arrangement is a permutation of the mapping between the components in an object model and those in a pattern template. Whenever a delta fails to match components in the object model, the analyzer tries other deltas along the traversed

path. If none of the deltas matches, a different component arrangement is used and the same process is repeated. If all these attempts fail, a new pattern template is created.

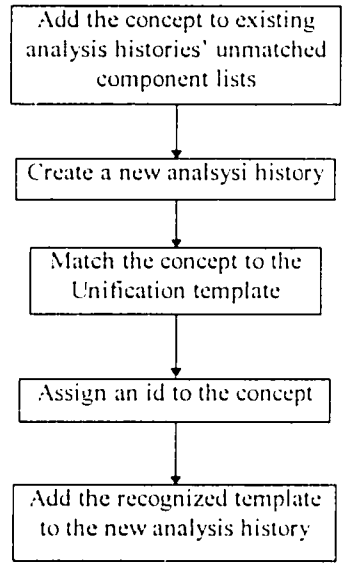



*Figure 12 Flow chart for adding a concept*

To start the analysis process, a user has to add a new concept or a new relation in his/her object model. Let's start by adding a new concept. When a new concept is added to the object model, the object-modeling tool informs the analyzer of the adding of the new concept. The analyzer adds the new concept to the unmatched component list of its analysis histories, if it has any analysis history. The reason this step is required is explained later. The analyzer also creates a new analysis history. The analysis history matches the new concept to the only component of the Unification template, which is the first template in the ontology. The analysis history creates a new instance of MatchedTemplate to reference the Unification template. This way the next time the analyzer tries to match new components in the object model, it can start from the Unification template. The analysis history maps the id of the only concept in the Unification template to the new concept added by the user. This indicates the new concept is a participant of the Unification pattern.

An analysis history serves several purposes. 1. By creating an analysis history for every concept added to an object model and matching the concept to the root template

of the ontology allows the analyzer to recognize a design pattern even when it is entered in an order different from the path stored in the ontology. Without doing this, a design pattern can only be recognized if it is entered in the order of the deltas comprising the design pattern. That means for the analyzer to recognize the Adapter pattern in an object model, that a user would have to enter the components in the object model in the following order:

- Concept 100
- Concept 200
- Association relation from Concept 200 to Concept 100
- Concept 500
- Subclass relation from Concept 100 to Concept 500.
- Concept 400
- Association relation from Concept 500 to Concept 400 (see Figure 13).

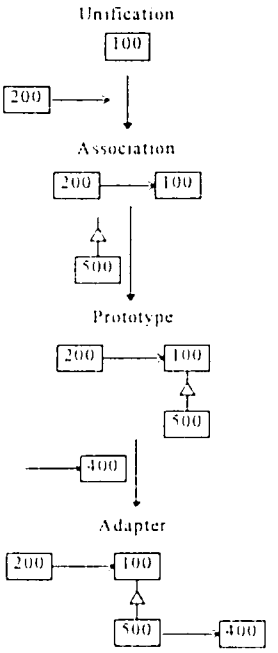



*Figure 13: The path in the ontology leads to the Adapter pattern*

2. By matching each of the concepts to the Unification template makes all possible permutations of the mapping between the components in an object model and those in a template reachable.

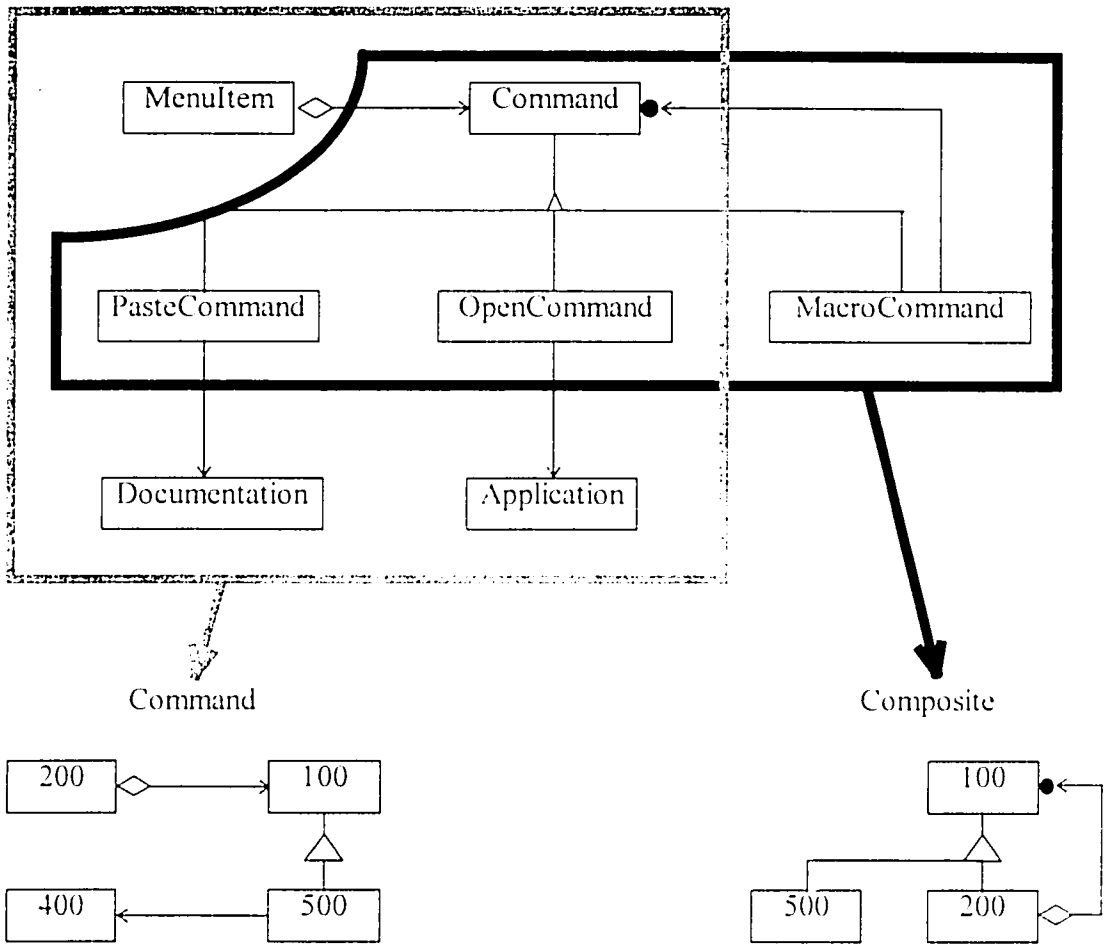



*Figure 14: The above object model uses both Command pattern and Composite pattern. Command concept is where both patterns overlapped. In this case, the two patterns cannot be recognized by two separate analysis histories. Instead, a new pattern template has to be created to capture this combination.*

3. Because an analysis history is created for each concept in the object model, it is possible for the analyzer to recognize more than one design pattern used in combinations in the object model. The only type of design pattern combination cannot be recognized is when the design patterns are joined at the concept of the Unification pattern, i.e. Concept 100 in the ontology (see Figure 14). This limitation is compensated by treating these design pattern combinations as new design patterns and including them in the ontology.

4. An analysis history reduces the effort of matching. An analysis history maintains a linked list of templates in the order they are recognized. When the user enters new

components in the object model, instead of rematching everything from square one, the last recognized template is used as the starting point. The analyzer assumes the new components do not invalidate the templates recognized previously and attempts to match the components to the deltas of the last recognized template directly. Of course there are times the new components will not match the deltas. The algorithm dealing with these situations is discussed in details later.

5. The linked list of recognized templates stored in an analysis history provides the explanation on how the analyzer recognizes a design pattern.

The analyzer uses the ontology to guide its search. Instead of trying to match all known templates to the object model, only the ones derived from the recognized template are examined. This is achieved by matching deltas of a recognized template. As mentioned in Section 2.2.1, the templates in the ontology are connected by deltas. A delta may be a new relation or a relation and a new concept added to its source template. Or it may be a relation that specializes an existing relation in its source template. A delta must have one relation, because that is the only way to add a concept that is connected to a design pattern. This means that the user must add a new relation in his/her object model in order to advance to the next template in the ontology. Whenever the user adds a new component in the object model, it is stored in the unmatched component lists of all existing analysis histories. When the analyzer tries to match a delta, it tries to match the delta to the unmatched components of the selected analysis history first. Because very often a delta represents the new components added to its source template rather than a specialized relation in its source template, by matching the delta to unmatched components increases the chance of a good match. Unmatched components are stored instead of calculated for better speed performance.

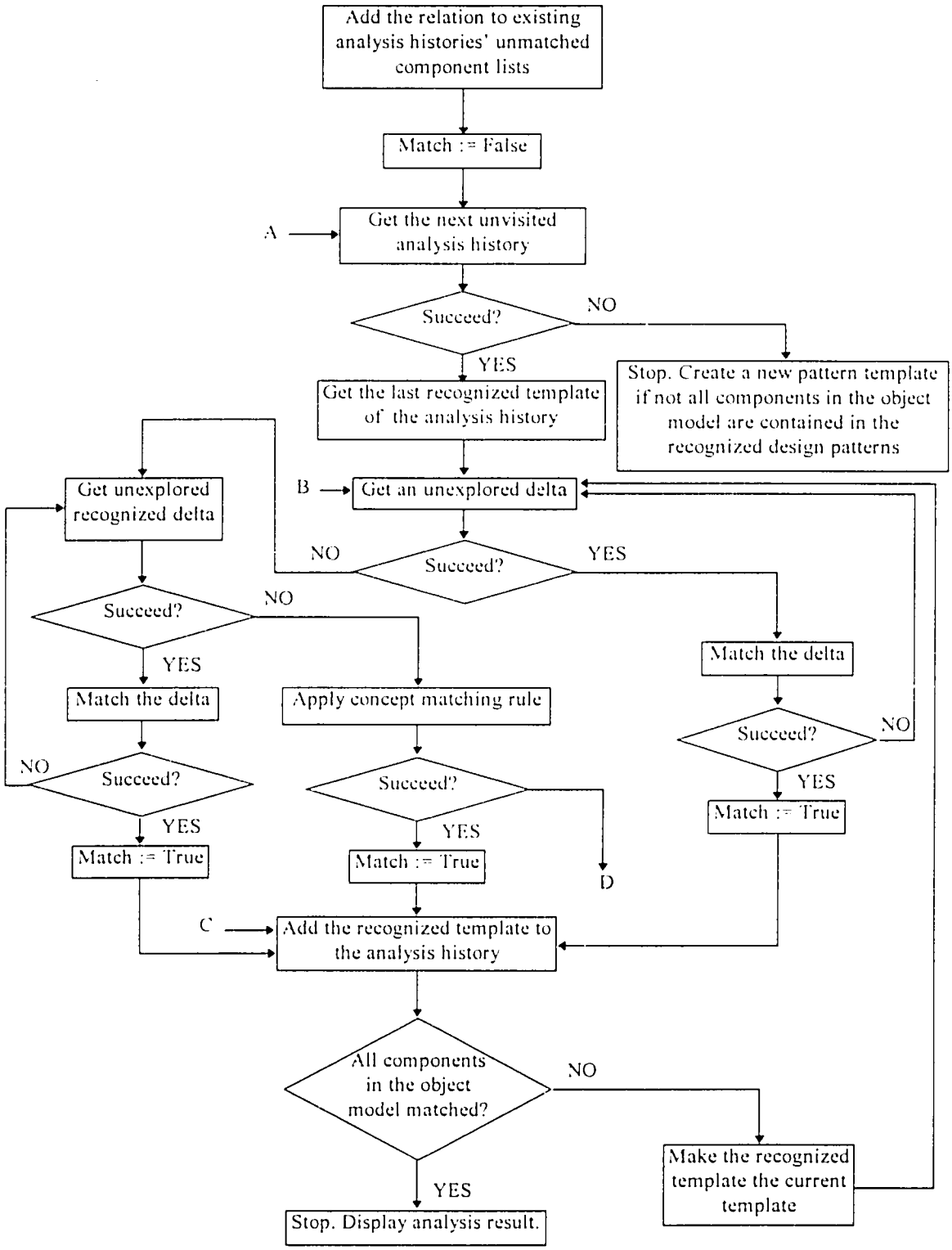



*Figure 15: Flow Chart for Adding a Relation.*

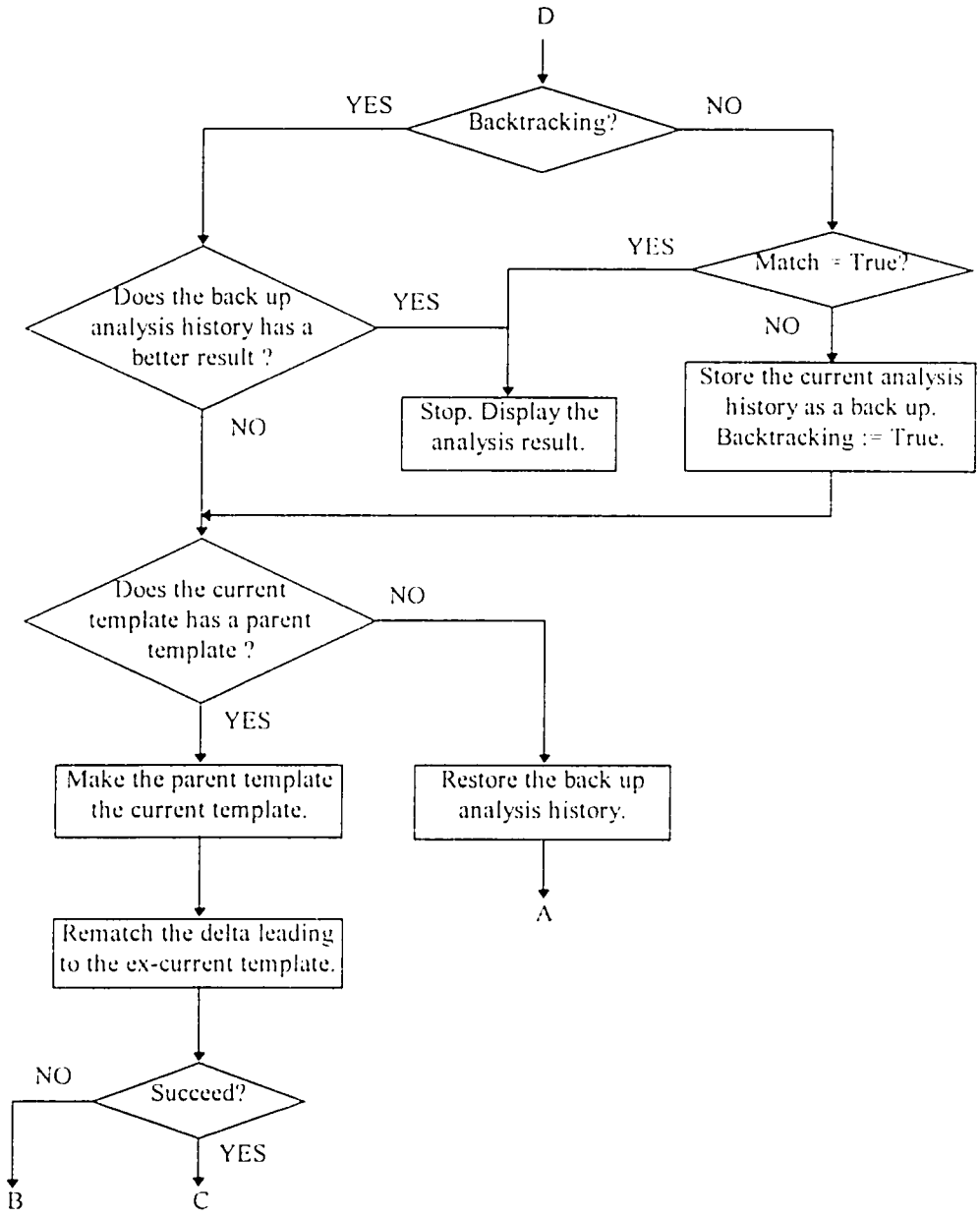



*Figure 16: Flow Chart for Adding a Relation (Continued).*

The analysis process is more complicated when a relation is added. After the user adds a new relation in his/her object model, the object-modeling tool informs the analyzer that the new relation is added. The analyzer adds the new relation to the unmatched component list of all its existing analysis histories. The analyzer explores

each analysis history. The new relation may allow more complicated design patterns to be recognized by different analysis histories.

For each existing analysis history, the first attempt is to find out if the design pattern recognized by the analysis history is only a part of a more complicated design pattern. A more complicated design pattern usually implies more participants and relations are involved. This is aligned with the second goal of the analysis process because the larger the design patterns the fewer of them are required to cover the object model. In practice, it is also more desirable to recognize a complicated design pattern than a simple design pattern. Simple design patterns, called meta patterns [Pree, 95], like Unification, Template Method, Association, 1:1 Connection, 1:N Connection, 1:N Recursive Connection, Recursive Association, 1:1 Recursive Unification and 1:N Recursive Unification are the basic building blocks of object models and design patterns. Recognizing these design patterns in an object model has very limited value because they are usually parts of other design patterns and they offer little design abstraction. To find out if a more complicated design pattern exists in the object model, the analyzer tries to match the unmatched components to the deltas of the last recognized pattern template. Because the pattern templates in the ontology are arranged so the templates higher in the hierarchy (closer to the root) are simpler than the ones lower in the hierarchy (closer to the leaves), by matching to a delta, which leads to a template lower in the hierarchy, the analyzer manages to recognize a more complicated design pattern.

To match exactly a delta to the components of an object model is quite straightforward. The relation in the delta is tried first since either its source concept or destination concept must have been recognized in the previous matching because the relation in a delta must be connected to a concept in the delta's ancestor template. The delta can never be reached unless the ancestor template has been recognized. A delta relation is considered matched if there is a relation in the object model that has the same source concept, destination concept, type and cardinality. To check if a relation in the object model has the same source concept as the relation specified in the delta, DEPARS looks up the id of the source concept stored in the delta relation in the

analysis history's mapping tables. If the id is mapped to the same concept as the source concept of the relation in the object model, the delta relation has the same source concept as the relation. The same is done for the destination concept except replacing all occurrences of source concept with destination concept. Of course, when the delta involves a new concept, the new concept's id is not mapped to any concept in the object model. When this is the case, a relation, instead of matching both the delta relation's source concept and destination concept, only needs to match either the source concept or the destination concept, depending whether the new concept is a destination concept or a source concept. The prospective relation is then checked to see if it is connected to an unmatched concept in the object model. If so, the concept is mapped to the new concept in the delta. Both the relation and the concept are considered matched. If the relation is not connected to an unmatched concept, it fails to match the delta relation.

The relation of a delta is matched against the following types of relations in the listed order: 1. relations already assigned the same id as the delta relation, 2. unmatched relations, and 3. relations with a special subclass id. The first case takes care of the scenario where a delta relation specializes an existing relation in the ancestor template. The second case allows a more complicated design pattern to be recognized by matching unmatched relations. The third case only applies if the delta relation is a subclassing relation. Sometimes a concept is matched to its parent concept by applying the matching rules (see Section 2.2.3.2 for details). And the subclassing relation connecting these two concepts is assigned a special id. So when a delta with a subclassing relation is encountered, the relations with special ids are re-matched. If they match the subclassing relation of the delta, they are assigned the delta relation's id.

#### *2.2.3.1 Matching Rules for Relations*

Exact matching is not very useful since most of the matching requires some reasonable degree of tolerance. Several matching rules are applied to increase the matching tolerance. There are two types of matching rules, one for matching a

relation, the other for matching a concept. The matching rules for matching a concept will be explained later.

There are four types of relations: association, aggregation, creation and subclassing. Among them, associations and aggregations are often associated with cardinalities. Aggregations are considered to be a specific type of association [Rumbaugh et al., 91]. The cardinalities of relations are specialized in a rather interesting way. A one-to-one relation is considered to be more general than a one-to-many relation simply because often it is used in an object model diagram when the cardinality of the relation is unknown or not cared. For example, in the Adapter pattern described in "Design Patterns" the relation between a Client and a Target is a one-to-one relation. However, in the example of an actual implementation of the Adapter pattern the Client (DrawingEditor) has a one-to-many relation to the Target (Shape) instead (see Figure 17). The analyzer employs a hierarchy of different types of relations to increase the tolerance when matching a relation. The hierarchy starts with an imaginary type of relation, connection, as the most generic type of relation. Creation and one-to-one association follow connection. One-to-many association and one-to-one aggregation follow one-to-one association. Both one-to-many association and one-to-one aggregation are followed by one-to-many aggregation, which is the most specific type of relation. A relation with a fixed cardinality greater than one is considered to be more specific than a one-to-one relation but more general than a one-to-many relation. A subclassing relation is only considered during concept matching. Therefore it is not included in the hierarchy (see Figure 18).

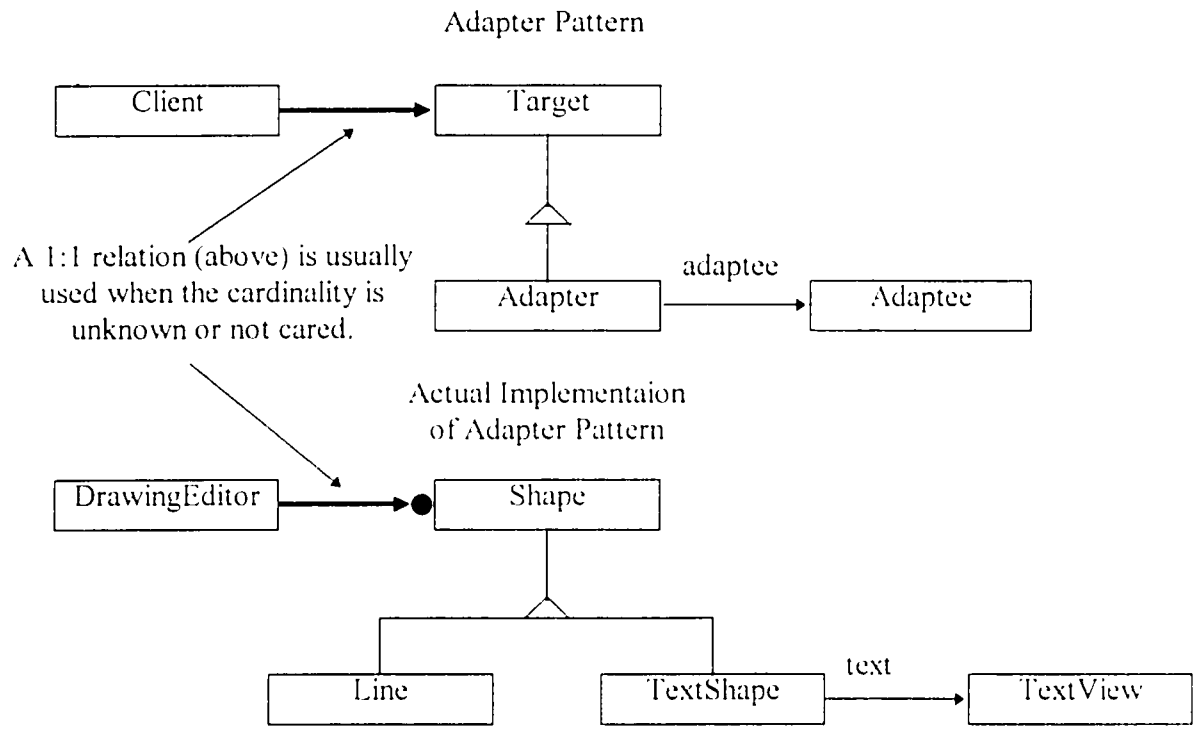



*Figure 17: A one-to-one relation is considered to be more generic than a one-to-many relation is simply because often it is used in an object model diagram when the cardinality of the relation is unknown or not cared*

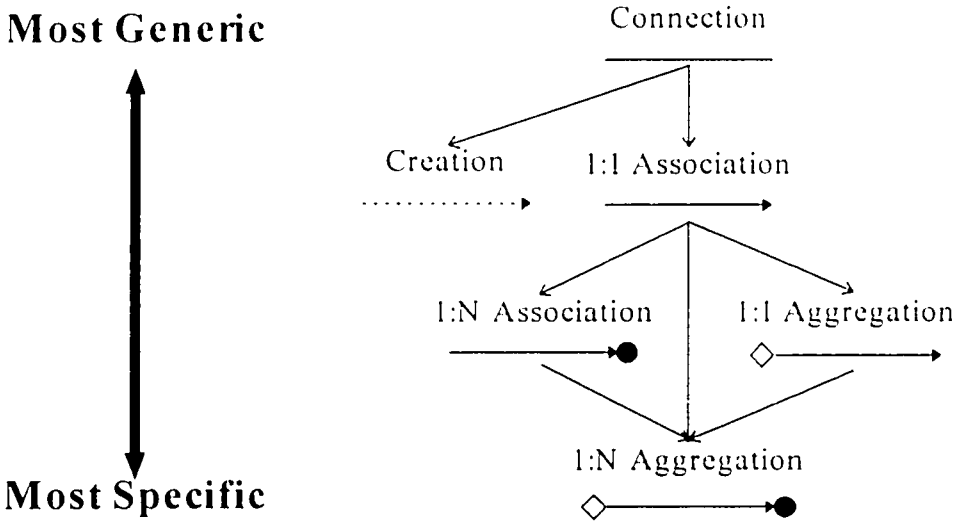



*Figure 18: Relation Hierarchy: Connection is the most generic relation. 1:N Aggregation is the most specific relation.*

Instead of exactly matching two relations' type and cardinality, the analyzer uses the relation hierarchy to increase the matching tolerance. If the relation in the object

model is more specific in terms of type and cardinality (lower in the relation hierarchy) than the delta relation, the type and cardinality of the two relations match. For example, a one-to-many association in the object model would match a one-to-one association in the delta in terms of type and cardinality. However, a one-to-one association in the object model would not match a one-to-many association's type and cardinality in the delta.

If a delta matches, its destination template is added to the analysis history. The analysis history is marked to indicate another design pattern has been recognized. If the delta does not match, the other deltas of the last recognized template are tried. If all these deltas fail to match, the deltas constructing the last recognized templates are tried. These deltas are matched one by one in the order of the template list stored in the analysis history. If none of these deltas matches, concept-matching rules are attempted. The steps are repeated until that no other components in the object model can be matched, provided the analysis history is marked. Otherwise, the analyzer starts to backtrack the analysis history.

#### *2.2.3.2 Matching Rule for Concepts*

The concept-matching rule allows a concept to be generalized and be seen as its parent concept. After all the deltas fail to match, the analyzer goes through any unmatched subclassing relations whose parent concepts have already been assigned ids. It generalizes all the child concepts connected by these relations and assigns them their parents' ids. The subclassing relations are also assigned special ids to indicate they are matched via the concept matching rules (see Figure 19).

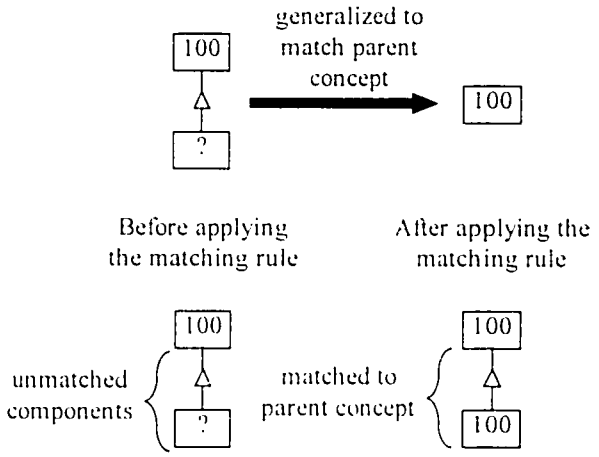



*Figure 19: Child concepts can be generalized to match their parent concept, so they are seen as the parent concept.*

Child concepts can also be generalized to their sibling concept, so they are seen as their sibling concept. Given an object model with a two level concept hierarchy, i.e. a parent concept with one immediate layer of child concepts, both the parent concept and one of the child concepts (the sibling concept) are assigned unique ids. The unmatched child concepts in the object model can be generalized and mapped to the sibling concept (see Figure 20). This generalization step is performed during the delta-matching phase.

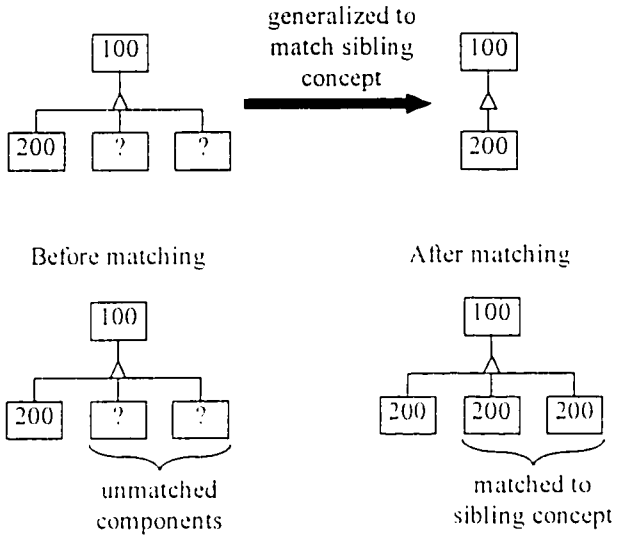



*Figure 20: Child concepts can also be generalized to their sibling concept, so they are seen as if they are their sibling concept*

### *2.2.3.3 Backtracking*

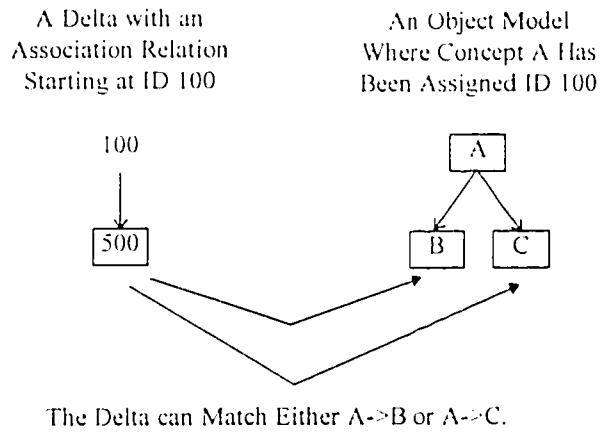



*Figure 21: The delta ->500 can match either A->B or A->C.*

When these steps fail, the analyzer backtracks on the ontology path it has pursued so far. The purpose of backtracking is to explore other paths in the ontology in the hope it may lead to a pattern template that has a better coverage on the object model. Before backtracking, the current analysis history is stored as a back up. In case the backtracking process does not yield a better result, the original analysis history is restored. This saves the analyzer from redoing the matching that has been done

previously. The first step of the backtracking process is to undo the latest matching. The latest matching is the matching of the delta that leads to the last recognized template in the analysis history. If there is no such a delta, i.e. the analysis history has only recognized the root template - Unification, backtracking fails to find a better result. The back up analysis history is restored. After undoing the last match, the last matched delta is re-matched. This is because sometimes there is more than one way to match a delta. For example, given an object model with three concepts, A, B and C. An association relation, A->B, connects from concept A to concept B. An association relation, A->C, connects from concept A to concept C. There is a delta with an association relation connecting to an un-mapped concept. The source concept of the association relation is mapped to the concept A. The delta can be matched by either the association relation A->B or the association relation A->C (see Figure 21). Both ways of matching meet the condition specified in the delta, an association relation from the concept A to another concept. We want to ensure that different permutations of matching the delta are tried before entirely abandoning the delta. If the delta is matched, the destination template of the delta is put back in the analysis history. The analysis process resumes with the deltas of the template. If the last matched delta cannot be re-matched in any other way, the analyzer tries to match the other deltas of the ancestor template; i.e. the template recognized right before the last recognized template. Like before, if any of these deltas are matched, its destination template is stored in the analysis history and the analysis process resumes with the destination template's deltas. However, if the deltas are not matched, the next latest matched delta is undone and re-matched. The analyzer repeatedly undoes matched deltas until an alternative path is found or the entire path has been undone. In the latter case, the back up analysis history is restored because backtracking fails to find a better result.

Regardless if a delta has been matched or not in an analysis history, the analyzer continues to explore other un-visited analysis histories. This is because an object model may contain more than one design pattern. The design patterns may be used in isolation or combinations. Exploring all analysis histories allows the participants of all design patterns to be recognized properly.

#### *2.2.3.4 Creating a New Pattern Template*

After exploring all analysis histories, the next step is to determine if the object model has been completely covered by design patterns. As mentioned earlier, Unification, Template Method, Association, 1:1 Connection, 1:N Connection, 1:N Recursive Connection, Recursive Association, 1:1 Recursive Unification and 1:N Recursive Unification are building blocks of object models and design patterns. They are so basic that they alone can always cover an object model. Therefore in order to get more realistic object model coverage, these design patterns should be excluded and not considered a part of the object model coverage. The analyzer stores a parameter, minNumComp, allowing users to specify the minimum number of participants in a design pattern for it to be taken into account in the object model coverage. The default value of minNumComp is set to three to exclude the above mentioned design patterns. The analyzer collects the mapping between components in the object model and the pattern templates exceeding the minimum number of participants from each analysis history. (Each analysis history represents a design pattern.) It then uses this information to identify the components in the object model that are not covered by any design patterns exceeding the minimum number of participants.

After identifying the unmatched components in the object model, the next step is to create a new pattern template to cover these components. The analyzer stores a parameter, maxUnmatch, which allows users to set the maximum number of components not involved in any of the recognized design patterns in an object model. If the number of the remaining unmatched components is lower than that is specified in the parameter, the analyzer will not create a new pattern template. The default value of maxUnmatch is set to zero. This ensures new pattern template is always created to cover the unmatched components in the object model. In order for an unmatched component to be covered in the new pattern template, it has to be connected to a recognized design pattern. This is because the new pattern template is created as an extension to the design pattern. Before creating a new pattern template, the analyzer has to choose a design pattern. The new pattern template is then built on top of the design pattern. In order to cover the object model with the minimum

number of design patterns, the design pattern with the most matched components, i.e. the last pattern template in an analysis history, is chosen first. If no unmatched components are connected to this design pattern, the design pattern with the second most matched components, i.e. the last pattern template in a different analysis history, is chosen and so on. If none of the unmatched components is connected to any of the recognized design patterns, no new pattern template will be generated. After a design pattern has been selected, the analyzer scans through the unmatched components and identifies the ones connecting to the design pattern. These components can be relations and concepts connecting to these relations, if any. As mentioned previously, pattern templates are connected by deltas. The same applies to the new pattern template. Because each delta can only hold a relation or a relation and a concept, more than one delta may be created. That means some intermediate pattern templates may be created along with the target pattern template. The analyzer tries to abstract the key components of the new pattern template by applying rules that generalize the connected yet unmatched components.

After creating a new pattern template, the analyzer matches the unmatched components to the new pattern template to reduce the number of unmatched components. Since the new pattern template is created based on the unmatched components, at least one unmatched component will be matched to it. The creation process is repeated until all unmatched components are covered or no more new pattern template can be created.

#### *2.2.3.5 Creation Rules*

The simplest case in creating a new pattern template is when the unmatched relation and concept, if any, in an object model are connected to a concept that has been uniquely assigned an id (see Figure 22). This means no other concepts in the object model are assigned the same id. In this case, the unmatched relation and concept, if any, are the delta between the new pattern template and the recognized design pattern. The analyzer then creates a delta with components sharing the same properties, such as name, type, cardinality, source concept and target concept for relations and name

for concepts, as the unmatched relation and concept. A new pattern template is also created using the same information. The same delta and template creation steps are performed in the following cases too.

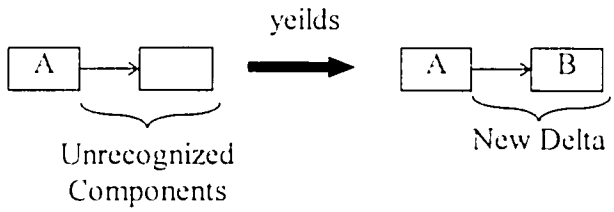



*Figure 22: When the unmatched relation and concept, if any, in an object model are connected to a concept that has been uniquely assigned an id, in this case, A, a delta is created based on the unmatched relation and concept.*

The second case is when the unmatched relation and concept, if any, in an object model are connected to a concept that has been assigned the same id as its parent concept (see Section 2.2.3.2 for details on matching rules for concepts) (see Figure 23). In other words, the unmatched components are connected to a child concept that is not considered a key component in the recognized design pattern. (That is why it is assigned the id of its parent concept.) Because of the unmatched components the child concept now has a different characteristic from its parent concept and can no longer be generalized to it. To create this new pattern template requires creating two other intermediate pattern templates. First of all, a new pattern template is created to include the child concept as its key component. A second pattern template is created to include the unmatched components as its key components. In order to distinguish child concepts without connection to other components from the child connecting to these unmatched components, a third pattern template is created to include a second child concept as its key components. For example, given an object model with concepts A, B, C and D, concept A is the parent concept. Both concepts C and D are child concepts of concept A. Concept B is connected to concept C by an association relation. If we were to match the object model to the templates we just created, it would be matched as follows: concept B is matched to the first delta. Concept C is matched to the second delta and concept D is matched to the third delta. The order of the deltas being created is important for better performance. By creating the deltas

identifying the two child concepts as key components one after the other allows more than one way of matching. Although the design pattern would eventually be recognized, yet the analyzer may need to backtrack the analysis history in the process thus reducing its matching efficiency.

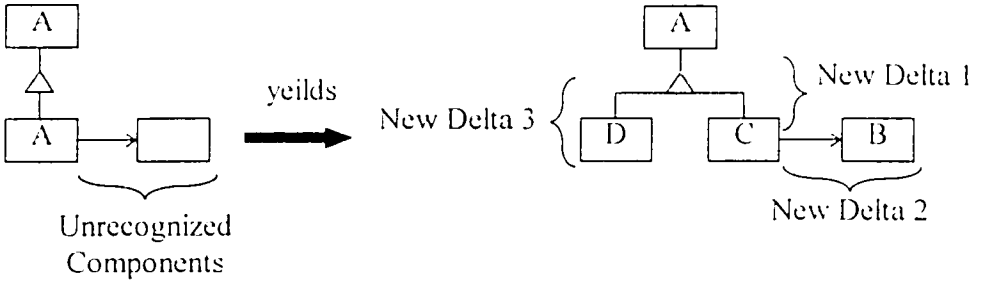



*Figure 23: When unrecognized components are connected to a child concept, three deltas are created to ensure the characteristics of the child concept is captured.*

The third case is similar to the second case, where the unmatched relation and concept, if any, are connected to a parent concept, instead of a child concept (see). Since the child concept is not a key component and the unmatched components are not connected to it, it does not have a unique characteristic that needs to be captured in the new pattern template. Therefore we only need to consider the unmatched components. In this case, a new pattern template is created to include the unmatched components as key components.

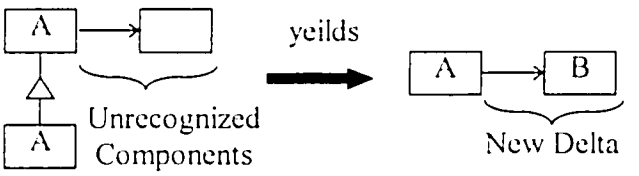



*Figure 24: When unrecognized components are connected to a parent concept, a new delta is created based on the unrecognized components.*

The fourth case is when more than one child concept in the object model are assigned the same id and the unmatched relation and concept, if any, are connected to one of them. Unlike the second case, the child concepts are not assigned the same id as their parent concept. This means the child concept is a key component of the recognized

design pattern. In this case, the analyzer does not need to create an intermediate pattern template just to include the child concept as a key component. The rest of the steps is similar to that of the second case. A new template is created to include the unmatched components as key components, and is followed by another new template that includes a second child concept as its key component.

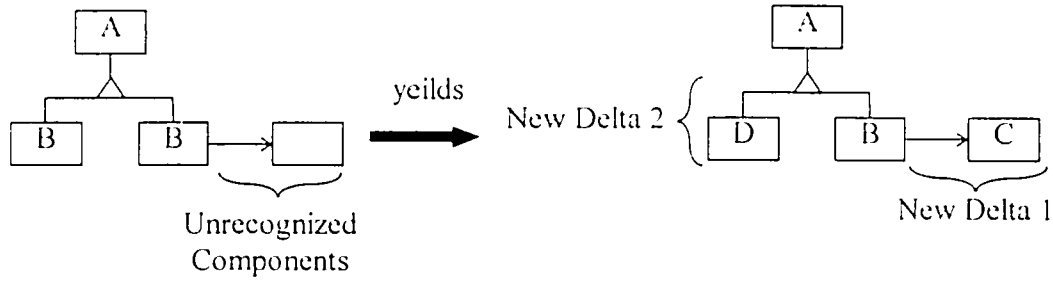



*Figure 25: When the unrecognized components are connected to a child concept that is a key component, two deltas are created. One delta identifies the unrecognized components. The second delta is added to preserve the characteristics of the child concept connecting to the unrecognized components.*

### 2.2.4 Deleting a Component

As the user is entering his/her object model, modifying what has already being entered is inevitable. The basic editing operations supported by existing object modeling tools are deleting concept, deleting relation, changing relation properties like name, type, source concept, destination concept and cardinality and changing concept name. Changing the names of concepts and relations does not influence the outcome of DEPARS's analysis process. Changing other relation properties like type, source concept and destination concept can always be achieved by deleting the relation and adding it back in with the new properties. Naturally it is not as efficient as just updating the related information. However, because of time constraint the implementation of this operation is omitted.

#### *2.2.4.1 Deleting a Concept*

It is assumed that in order to delete a concept, its relations, inward or outward must be deleted first. This means by the time a user is able to delete a concept, the concept must be disjoined from other concepts and has no recursive relations to itself. In that

case, deleting a concept does not have much impact on the outcome of the analysis process. When a concept is deleted, one of the analysis histories will become empty because the concept is the only component in the Unification pattern. As mentioned previously, the Unification pattern is the first pattern to be recognized and included in an analysis history. The empty analysis history is then deleted.

#### *2.2.4.2 Deleting a Relation*

Deleting a relation may invalidate a recognized design pattern, if the relation is the only component matching to a key component of the design pattern. When this occurs, the analyzer has to update the analysis histories to reflect the change.

After a relation is deleted from the object model, the first thing the analyzer does is to remove the relation from the mapping tables of the analysis histories. Sometimes the deleted relation is not the only component that requires updates. Deleting a relation usually invalidates the matching to the other components, such as the concept connected by the relation. Without the relation, the concept may become disjoined from the rest of the object model. The easiest way to identify and update these components accordingly is through pattern templates. The analyzer stores the pattern template that assigns the id of an object model component and the id itself in the object-model-component-to-pattern-template-component (OMC-PTC) mapping table indexed by the component. The mapping table also keeps the history of the id assignment, since it is possible that new ids are assigned to a component during the course of the analysis process. The component is identified by its most recently assigned id. For example a child concept may initially be assigned the same id as its parent concept. It is then assigned a new id after matching a delta that makes it a key component of a design pattern. The analyzer looks up the mapping table to find out the template that assigns the deleted relation's id. It then goes through the mapping table and "unmatches" any component that has been assigned an id by the same template. "Unmatching" a component involves going through each analysis history and deleting the id assignment record from the analysis history's OMC-PTC mapping table. If the component has only been assigned an id once, then its entries in both

mapping tables are removed and it is added back to the unmatched component list of the analysis history. If the component has been assigned more than one id, then the last id assignment is deleted from the assignment history stored in the OMC-PTC mapping table. The component is now identified by its next most recently assigned id. Sometimes the unmatching process unmatches components that should not have been unmatched. To fix this, the analyzer runs the analysis process after the unmatching process. This way we do not need a fancy algorithm to identify the exact components that require unmatching. We simply unmatch any potential components that may be affected by the deletion and rematch them afterwards.

### 2.2.5 Walk-through

To demonstrate how DEPARS works in practice, we will walk through two scenarios. In the first scenario, a user enters a part of the object model for a graphical user interface toolkit. The object model allows properties like borders or behaviors like scrolling to be added to any user interface component (for details please refer to [Gamma et al., 95] pages 175-177). The Decorator pattern is used in this object model (see Figure 26). We will use the non-incremental mode, i.e. DEPARS does not start analysis process until it is told to do so, to make the walk-through simple.

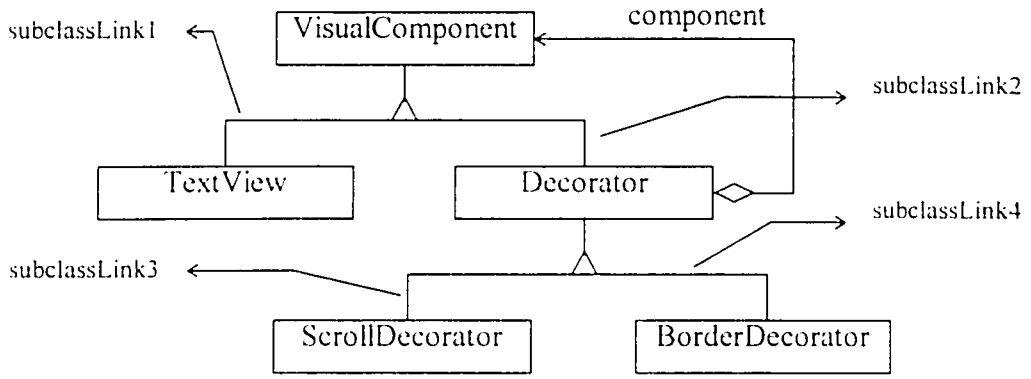



*Figure 26. The object model entered by the user in the first walk-though. The object model uses Decorator pattern. The subclass relations are named so they can be referred to later on in the walk-through.*

### *2.2.5.1 Scenario 1*

We start by entering the object model one component at a time. Every time a concept is created, DEPARS creates a corresponding analysis history for the concept and adds the concept to any existing analysis history's unmatched component list. Every time a relation is created, DEPARS simply adds it to the existing analysis histories' unmatched component list (see Figure 27). After completely entering the object model, DEPARS is instructed to start the analysis process. The complete knowledge base is used (see Appendix A).

Analysis History 1

| Id | Component | Template |
|---|---|---|
| 100 | VisualComponent | Unification |

VisualComponent

Analysis History 2

| Id | Component | Template |
|---|---|---|
| 100 | TextView | Unification |

TextView

Analysis History 3

| Id | Component | Template |
|---|---|---|
| 100 | Decorator | Unification |

Decorator

Analysis History 4

| Id | Component | Template |
|---|---|---|
| 100 | ScrollDecorator | Unification |

ScrollDecorator

Analysis History 5

| Id | Component | Template |
|---|---|---|
| 100 | BorderDecorator | Unification |

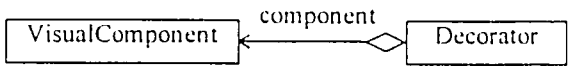
BorderDecorator

*Figure 27: Every time a concept is created, DEPARS creates a corresponding analysis history for the concept. Id is the component in a template that matches to the component in the object model. Component is the component in the object model. Template is the template that assigns the id to the component. The figures on the right of each analysis history show the current pattern recognized by each analysis history.*

The analyzer starts with the analysis history for VisualComponent. VisualComponent is given the id 100 because it matches to the only concept in the Unification template. The analyzer begins matching the first delta obtained from the Unification template. The delta has an association relation from concept 200 to concept 100 and a new concept 200. The analyzer finds the aggregation relation named "component" matches the delta relation and Decorator matches the delta concept. Decorator is now given the id 200. The relation is assigned the id 1200. And the template Association is appended at the end of the analysis history (see Figure 28). The object model is not yet covered by design patterns. The analyzer continues its analysis.

Analysis History 1

| Id | Component | Template |
|---|---|---|
| 100 | VisualComponent | Unification |
| 200 | Decorator | Association |
| 1200 | component | Association |




Analysis History 2

| Id | Component | Template |
|---|---|---|
| 100 | TextView | Unification |

TextView

Analysis History 3

| Id | Component | Template |
|---|---|---|
| 100 | Decorator | Unification |

Decorator

Analysis History 4

| Id | Component | Template |
|---|---|---|
| 100 | ScrollDecorator | Unification |

ScrollDecorator

Analysis History 5

| Id | Component | Template |
|---|---|---|
| 100 | BorderDecorator | Unification |

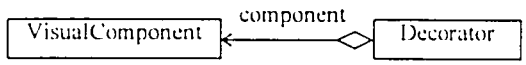
BorderDecorator

*Figure 28: DEPARS matches the first delta with Analysis history 1. Now Analysis history 1 recognizes the Association pattern in the object model.*

The analyzer gets the next delta from the Association template. The delta consists of an aggregation relation from concept 200 to concept 100. The relation 1200 matches the delta relation. The relation is assigned the id 1200 again, since the delta only specializes an existing relation. The template 1:1 Connection is appended at the end of the analysis history (see Figure 29). The object model is not yet covered by design patterns. The analyzer continues its analysis.

Analysis History 1

| Id | Component | Template |
|---|---|---|
| 100 | VisualComponent | Unification |
| 200 | Decorator | Association |
| 1200 | component | 1:1 Connection |




Analysis History 2

| Id | Component | Template |
|---|---|---|
| 100 | TextView | Unification |

TextView

Analysis History 3

| Id | Component | Template |
|---|---|---|
| 100 | Decorator | Unification |

Decorator

Analysis History 4

| Id | Component | Template |
|---|---|---|
| 100 | ScrollDecorator | Unification |

ScrollDecorator

Analysis History 5

| Id | Component | Template |
|---|---|---|
| 100 | BorderDecorator | Unification |

BorderDecorator

*Figure 29: DEPARS matches another delta. Analysis history 1 now recognizes the 1:1 Connection pattern in the object model.*

The analyzer gets its next delta from the 1:1 Connection template. The delta consists of a concept 600 and a creation relation from concept 600 to concept 100. No match is found for this delta. The analyzer gets the next delta. The delta consists of a 1:N aggregation relation from concept 200 to concept 100. Relation 1200 does not match the delta because it is a 1:1 aggregation relation. The analyzer tries the next delta. The delta consists of a concept 500 and a subclass relation from concept 100 to concept 500. The analyzer matches the subclass relation from VisualComponent to TextView to the delta relation and TextView to concept 500. The Strategy template is appended to the end of the analysis history (see Figure 30). The object model is still not covered by design patterns. Therefore the analysis process continues.

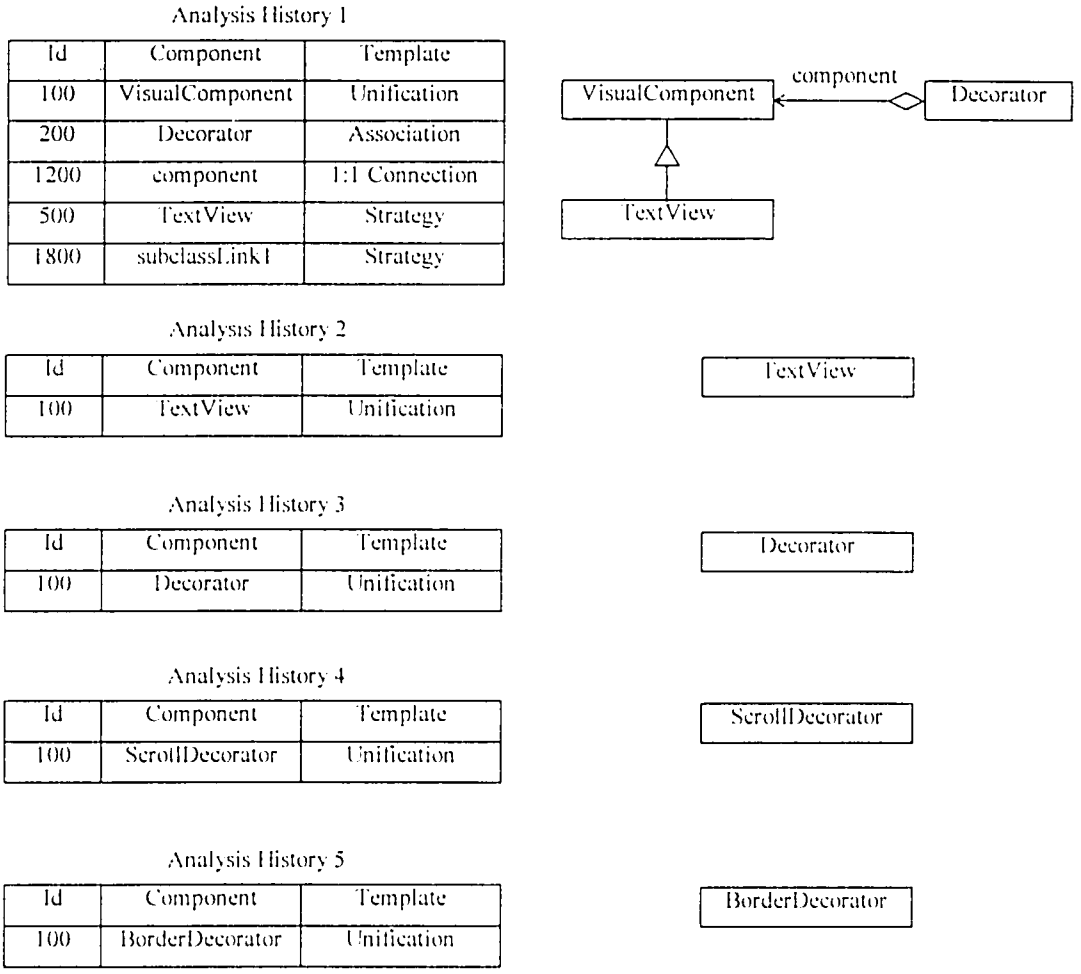


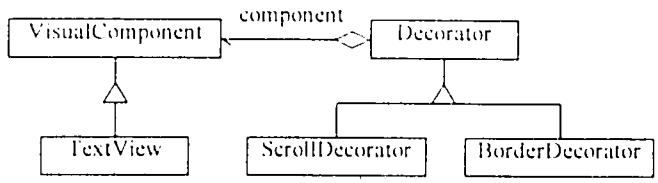

*Figure 30: DEPARS matches the next delta. Analysis history 1 now recognizes the Strategy pattern in the object model.*

The analyzer gets the next delta from the Strategy template. The delta consists of a concept 400 and a connection relation from concept 500 to concept 400. The analyzer cannot find any component matching this delta and moves on to the next delta. The next delta consists of a 1:N aggregation relation from concept 200 to concept 100. Again the only relation that may match the delta is relation 1200, yet it is a 1:1 aggregation relation. Therefore it fails to match the delta. The next delta consists of a concept 400 and a subclass relation 1900 from concept 200 to concept 400. The analyzer finds that both ScrollDecorator and BorderDecorator match concept 400 and the subclass relations from Decorator to ScrollDecorator and from Decorator to

BorderDecorator match subclass relation 1900. The analyzer assigns both ScrollDecorator and BorderDecorator the id 400 and their subclass relations the id 1900. The Bridge template is appended to the end of the analysis history (see Figure 31). There is still a subclass relation in the object model that is not covered by design patterns. Therefore the analysis process continues.

Analysis History 1

| Id | Component | Template |
|---|---|---|
| 100 | VisualComponent | Unification |
| 200 | Decorator | Association |
| 1200 | component | 1:1 Connection |
| 500 | TextView | Strategy |
| 1800 | subclassLink1 | Strategy |
| 400 | BorderDecorator | Bridge |
| 1900 | subclassLink4 | Bridge |
| 400 | ScrollDecorator | Bridge |
| 1900 | subclassLink3 | Bridge |




Analysis History 2

| Id | Component | Template |
|---|---|---|
| 100 | TextView | Unification |

TextView

Analysis History 3

| Id | Component | Template |
|---|---|---|
| 100 | Decorator | Unification |

Decorator

Analysis History 4

| Id | Component | Template |
|---|---|---|
| 100 | ScrollDecorator | Unification |

ScrollDecorator

Analysis History 5

| Id | Component | Template |
|---|---|---|
| 100 | BorderDecorator | Unification |

BorderDecorator

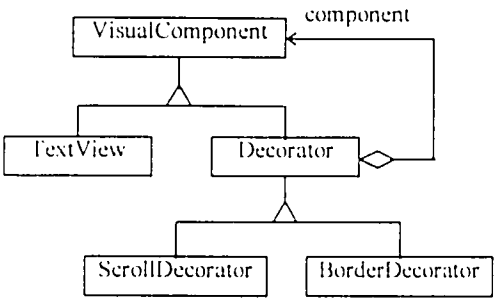
*Figure 31: Analysis history 1 now recognizes the Bridge pattern in the object model.*

The analyzer gets the next delta from Bridge. The delta consists of a subclass relation from concept 100 to concept 200, which matches to the subclass relation from

VisualComponent to Decorator. The relation is assigned the id 1300. The Decorator template is appended to the end of the analysis history (see Figure 32). The analysis process is completed because all components in the object model are covered by the Decorator pattern.

Analysis History 1

| Id | Component | Template |
|---|---|---|
| 100 | VisualComponent | Unification |
| 200 | Decorator | Association |
| 1200 | component | 1:1 Connection |
| 500 | TextView | Strategy |
| 1800 | subclassLink1 | Strategy |
| 400 | BorderDecorator | Bridge |
| 1900 | subclassLink4 | Bridge |
| 400 | ScrollDecorator | Bridge |
| 1900 | subclassLink3 | Bridge |
| 1500 | subclassLink2 | Decorator |




Analysis History 2

| Id | Component | Template |
|---|---|---|
| 100 | TextView | Unification |

TextView

Analysis History 3

| Id | Component | Template |
|---|---|---|
| 100 | Decorator | Unification |

Decorator

Analysis History 4

| Id | Component | Template |
|---|---|---|
| 100 | ScrollDecorator | Unification |

ScrollDecorator

Analysis History 5

| Id | Component | Template |
|---|---|---|
| 100 | BorderDecorator | Unification |

BorderDecorator

*Figure 32. Analysis history 1 now recognizes the Decorator pattern in the object model. Because all components in the object model are now covered by a design pattern, the analysis process terminates here.*

### *2.2.5.2 Scenario 2*

In the second scenario, a user enters a part of a server object model designed by Doug Schmidt [Schmidt, Dec 96]. The object model serves as a framework for developing an efficient, robust, extensible and reusable communication server (See Figure 33). The object model consists of four different networking design patterns. The design patterns are Reactor, Connector, Acceptor and Service Configurator (See Section

3.1.1). The challenge here for DEPARS is to recognize the four design patterns in this object model using its knowledge base. (See Appendix A for the location of these networking design patterns in the knowledge base.) This time we will run DEPARS in the incremental mode. That means DEPARS will analyze the data as soon as the user enters it.

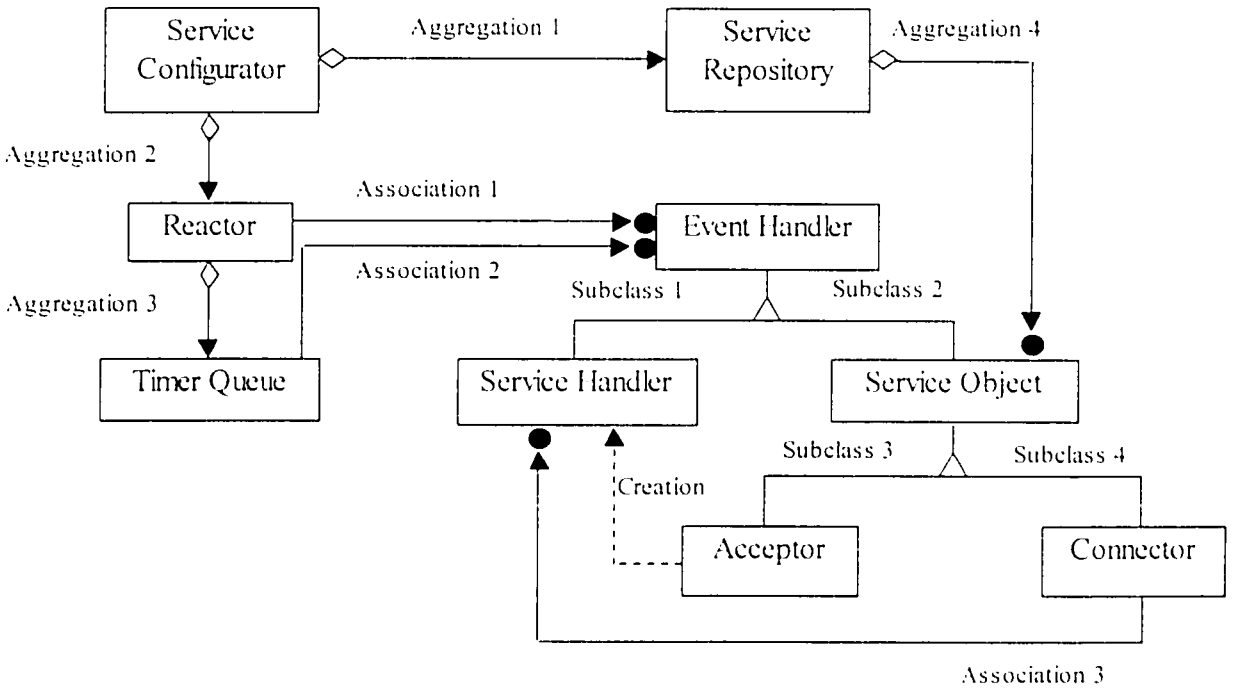


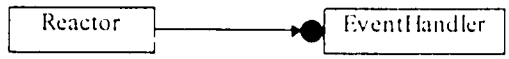

*Figure 33: The object model entered by the user in the second walk-through. The object model consists of four different design patterns. Relations are named for easy reference in the walk-through.*

We start the scenario by entering Reactor. After Reactor is entered, DEPARS creates an analysis history for the object. We will call it AHR. Reactor is now assigned the id 100 by Unification pattern. We then enter the second object, EventHandler. DEPARS creates the second analysis history and EventHandler is assigned the id 100 by Unification pattern in that analysis history. We will call this analysis history AHEH. AHR remains unchanged at this point. We proceed by adding a one-to-many association relation from Reactor to Event Handler. DEPARS tries to match the new data to the next pattern in the knowledge base. It goes though each analysis history

and tries to match a delta of the last recognized pattern. DEPARS cannot match any deltas in AHR. It then continues with AHER. DEPARS matches the association relation to Association pattern and then 1:N association pattern. Reactor is assigned the id 200 and the association relation is assigned the id 1500 in AHEH. DEPARS rearranges the analysis histories so the analysis history with higher number of matched objects will be matched first next time.

AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |




AHR

| Id | Component | Template |
|---|---|---|
| 100 | Reactor | Unification |

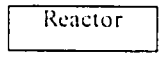
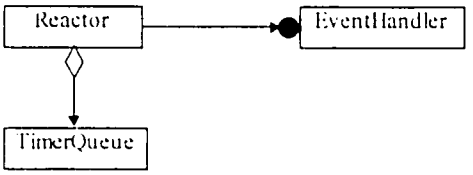



*Figure 34: After matching the one-to-many association relation between Reactor and Event Handler, DEPARS recognizes 1:N Association pattern in AHEH.*

We proceed by entering TimerQueue and the aggregation relation Aggregation 3. DEPARS matches the new data to PreReactor pattern in AHEH. A third analysis history AHT is created for Timer Queue. DEPARS recognizes Aggregation pattern in AHT. AHR remains unchanged.

AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |




AHT

| Id | Component | Template |
|---|---|---|
| 100 | TimerQueue | Unification |
| 200 | Reactor | Association |
| 1500 | Aggregation3 | 1:1 Connection |


AHR

| Id | Component | Template |
|---|---|---|
| 100 | Reactor | Unification |

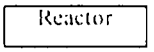
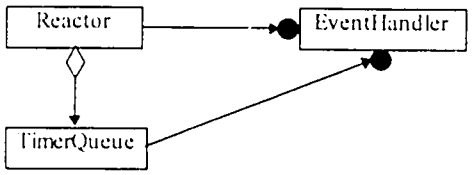



*Figure 35: DEPARS recognizes PreReactor pattern in AHEH and 1:1 Connection pattern in AHT after the user enters TimerQueue and Aggregation3 in the object model.*

We proceed by entering the one-to-many association relation between TimerQueue and EventHandler. DEPARS recognizes PreReactor2 pattern in AHEH. The other two analysis histories remain unchanged.

AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |
| 2600 | Association2 | PreReactor2 |




AHT

| Id | Component | Template |
|---|---|---|
| 100 | TimerQueue | Unification |
| 200 | Reactor | Association |
| 1500 | Aggregation3 | 1:1 Connection |


AHR

| Id | Component | Template |
|---|---|---|
| 100 | Reactor | Unification |

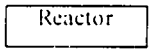
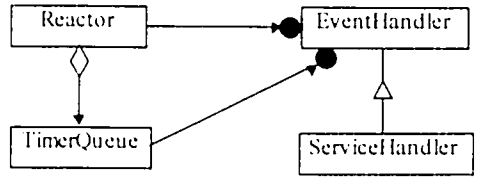



*Figure 36. DEPARS recognizes PreReactor2 pattern after the user enters Association2.*

We proceed again by entering ServiceHandler and the subclass relation from EventHandler to ServiceHandler. DEPARS recognizes Reactor pattern in AHEH. DEPARS also creates the fourth analysis history AHSH for ServiceHandler. Both AHT and AHR remain unaffected by the new data.

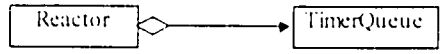
AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |
| 2600 | Association2 | PreReactor2 |
| 1800 | Subclass1 | Reactor |
| 500 | ServiceHandler | Reactor |



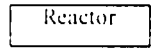

AHT

| Id | Component | Template |
|---|---|---|
| 100 | TimerQueue | Unification |
| 200 | Reactor | Association |
| 1500 | Aggregation3 | 1:1 Connection |




AHR

| Id | Component | Template |
|---|---|---|
| 100 | Reactor | Unification |




AHSH

| Id | Component | Template |
|---|---|---|
| 100 | ServiceHandler | Unification |

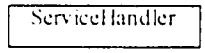


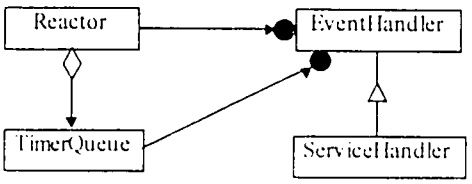

*Figure 37: DEPARS recognizes Reactor pattern after the user enters ServiceHandler and Subclass1.*

AHT, AHR and AHSH remain very much inactive in the remaining of the walk through. We will stop showing these analysis histories in the following figures to save space.

We proceed by entering ServiceConfiguration and ServiceRepository. DEPARS creates analysis histories AHSC and AHSR for ServiceConfiguration and ServiceRepository respectively. We then enter the aggregation relation between ServiceConfiguration and ServiceRepository. All existing analysis histories except AHSR remain unchanged.

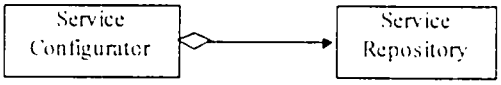
AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |
| 2600 | Association2 | PreReactor2 |
| 1800 | Subclass1 | Reactor |
| 500 | ServiceHandler | Reactor |




AHSR

| Id | Component | Template |
|---|---|---|
| 100 | ServiceRepository | Unification |
| 200 | ServiceConfigurator | Association |
| 1500 | Aggregation2 | 1:1 Connection |




AHSC

| Id | Component | Template |
|---|---|---|
| 100 | ServiceConfigurator | Unification |

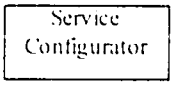


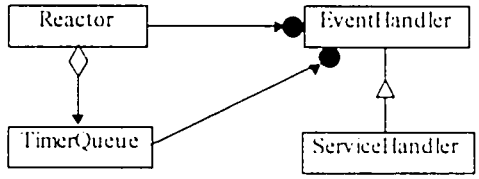

*Figure 38: DEPARS adds two new analysis histories after the user enters ServiceConfigurator and ServiceRepository. DEPARS recognizes 1:1 Connection in AHSR after the user enters Aggregation2.*

We proceed by entering ServiceObject and the one-to-many aggregation relation from ServiceRepository to ServiceObject. DEPARS creates an analysis history AHSO for ServiceObject. After Aggregation4 is entered, DEPARS recognizes Association pattern, then 1:1 Connection pattern, then 1:N Connection pattern and finally 1:N Connection 2 pattern in AHSO.

AHSR and AHSC will not be shown in the future figures due to lack of activities.

AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |
| 2600 | Association2 | PreReactor2 |
| 1800 | Subclass1 | Reactor |
| 500 | ServiceHandler | Reactor |




AHSO

| Id | Component | Template |
|---|---|---|
| 100 | ServiceObject | Unification |
| 200 | ServiceRepository | Association |
| 1500 | Aggregation4 | 1:N Connection |
| 2900 | ServiceConfigurator | 1:N Connection2 |
| 2800 | Aggregation1 | 1:N Connection2 |

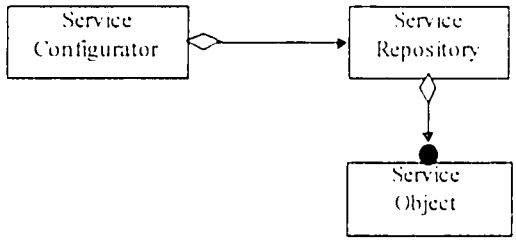
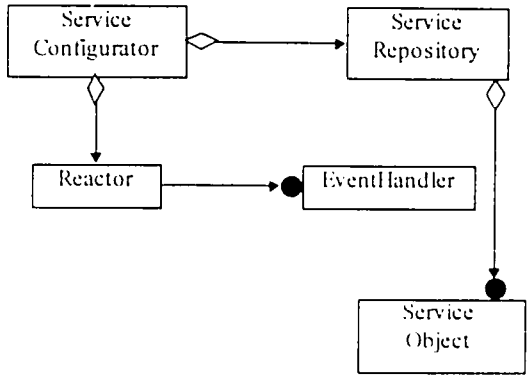



*Figure 39: DEPARS recognizes Association, 1:1 Connection, 1:N Connection and 1:N Connection2 in AHSO*

We proceed this time by adding an aggregation relation from ServiceConfigurator to Reactor. AHEH remains unchanged by all the past few steps because the new data is disjoint from the pattern it recognizes. In the mean time, DEPARS recognizes PreSvcConfig1 pattern and PreSvcConfig2 pattern in AHSO after the aggregation relation is entered.

AHSO

| Id | Component | Template |
|---|---|---|
| 100 | ServiceObject | Unification |
| 200 | ServiceRepository | Association |
| 1500 | Aggregation4 | 1:N Connection |
| 2900 | ServiceConfigurator | 1:N Connection2 |
| 2800 | Aggregation1 | 1:N Connection2 |
| 3100 | Reactor | PreSvcConfig1 |
| 3000 | Aggregation2 | PreSvcConfig1 |
| 3300 | EventHandler | PreSvcConfig2 |
| 3200 | Association1 | PreSvcConfig2 |




AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |
| 2600 | Association2 | PreReactor2 |
| 1800 | Subclass1 | Reactor |
| 500 | ServiceHandler | Reactor |

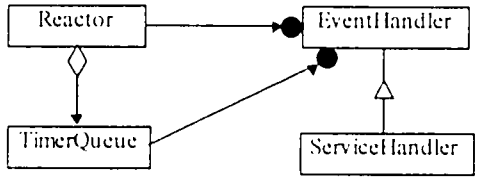
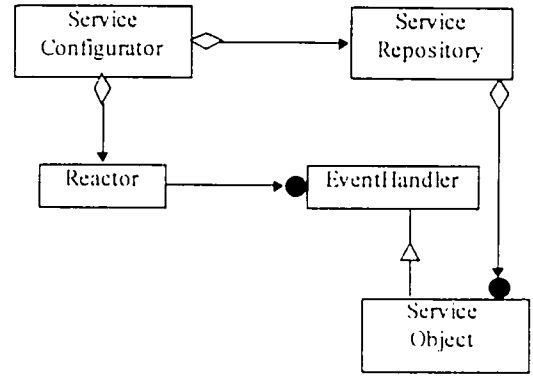



*Figure 40: DEPARS recognizes PreSvcConfig1 and PreSvcConfig2 patterns in AHSO after the user enters Aggregation2.*

We proceed again by entering Subclass2, the subclass relation from EventHandler to ServiceObject. DEPARS recognizes Service Configurator pattern in AHSO. The new data is also recognizes as a duplicated branch in Reactor pattern in AHEH.

AHSO

| Id | Component | Template |
|---|---|---|
| 100 | ServiceObject | Unification |
| 200 | ServiceRepository | Association |
| 1500 | Aggregation4 | 1:N Connection |
| 2900 | ServiceConfigurator | 1:N Connection2 |
| 2800 | Aggregation1 | 1:N Connection2 |
| 3100 | Reactor | PreSvcConfig1 |
| 3000 | Aggregation2 | PreSvcConfig1 |
| 3300 | EventHandler | PreSvcConfig2 |
| 3200 | Association1 | PreSvcConfig2 |
| 3400 | Subclass2 | ServiceConfigurator |




AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |
| 2600 | Association2 | PreReactor2 |
| 1800 | Subclass1 | Reactor |
| 500 | ServiceHandler | Reactor |
| 1800 | Subclass2 | Reactor |
| 500 | ServiceObject | Reactor |

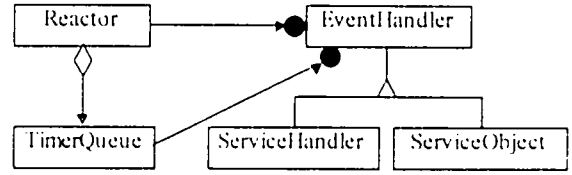
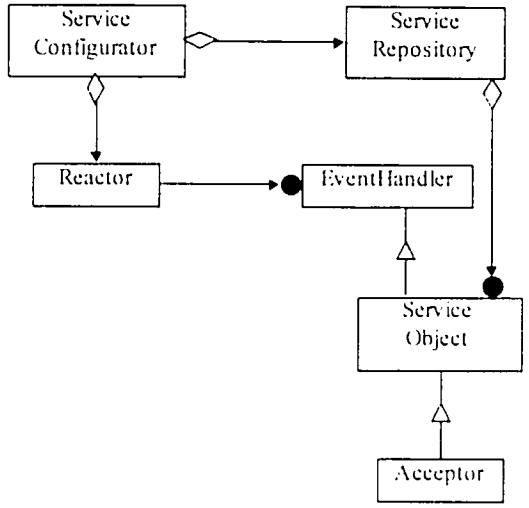



*Figure 41: DEPARS recognizes Service Configurator pattern in AHSO after Subclass2 is added to the object model. DEPARS recognizes Subclass2 and ServiceObject as a duplicated branch in Reactor pattern in AHEH.*

We continue by adding Acceptor and the subclass link from ServiceObject to Acceptor. Again, DEPARS creates an analysis history for the new object added to the object model. DEPARS recognizes Subclass3 and Acceptor as a part of Service Configurator 2 pattern in AHSO. DEPARS also recognizes Subclass3 and Acceptor as a part of Reactor pattern in AHEH by generalizing Acceptor as a kind of ServiceObject.

AHSO

| Id | Component | Template |
|---|---|---|
| 100 | ServiceObject | Unification |
| 200 | ServiceRepository | Association |
| 1500 | Aggregation4 | 1:N Connection |
| 2900 | ServiceConfigurator | 1:N Connection2 |
| 2800 | Aggregation1 | 1:N Connection2 |
| 3100 | Reactor | PreSvcConfig1 |
| 3000 | Aggregation2 | PreSvcConfig1 |
| 3300 | EventHandler | PreSvcConfig2 |
| 3200 | Association1 | PreSvcConfig2 |
| 3400 | Subclass2 | ServiceConfigurator |
| 3700 | Acceptor | ServiceConfigurator2 |
| 3800 | Subclass3 | ServiceConfigurator2 |




AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |
| 2600 | Association2 | PreReactor2 |
| 1800 | Subclass1 | Reactor |
| 500 | ServiceHandler | Reactor |
| 1800 | Subclass2 | Reactor |
| 500 | ServiceObject | Reactor |
| 501 | Subclass3 | Reactor |
| 500 | Acceptor | Reactor |

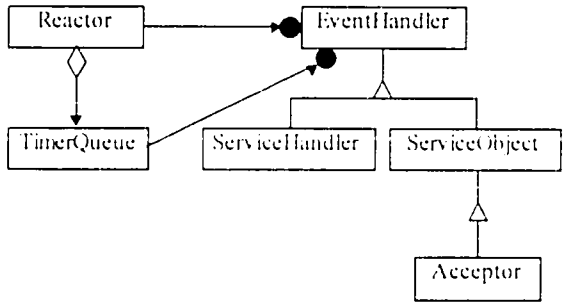


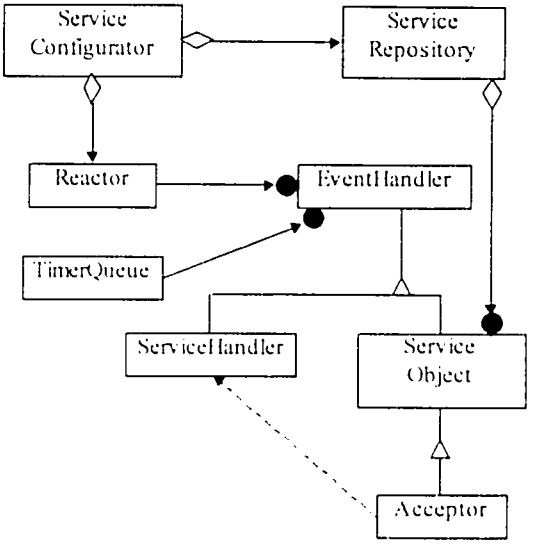

*Figure 42: DEPARS recognizes Service Configurator2 in AHSO. DEPARS generalizes Acceptor and Subclass3 as ServiceObject in AHEH.*

We continue again by adding the creation relation from Acceptor to ServiceHandler. This time, DEPARS cannot recognize the new data in any of its analysis histories even after applying the matching rules. Although Acceptor pattern exists in the knowledge base, it is not considered as a favorable pattern comparing to Reactor, because we can match more components with Reactor pattern. Since both Reactor and

Acceptor patterns originate from the same object, i.e. EventHandler, DEPARS can only recognize one of them and create a new template to cover the unrecognized components (See Section 2.2.3). In this case the new template is created based on Service Configurator 2 pattern, since it is currently the pattern that recognizes the most components and the creation link is connected to the pattern. After applying the matching rules, DEPARS recognizes TimeQueue and ServiceHandler as a duplicated part of the Service Configurator pattern. When DEPARS creates the template which will cover the creation relation, it also needs to create a template to assign ServiceHandler a unique id. This is necessary because the creation link makes ServiceHandler unique. In this case, DEPARS creates the first template including the creation link and ServiceHandler. It then creates another template to include the subclass link from EventHandler to ServiceHandler. DEPARS names the first new template ServiceConfig2+ and the second template ServiceConfig2++. DEPARS then tries to match the new templates to the existing object model.

AHSO

| Id | Component | Template |
|---|---|---|
| 100 | ServiceObject | Unification |
| 200 | ServiceRepository | Association |
| 1500 | Aggregation4 | 1:N Connection |
| 2900 | ServiceConfigurator | 1:N Connection2 |
| 2800 | Aggregation1 | 1:N Connection2 |
| 3100 | Reactor | PreSvcConfig1 |
| 3000 | Aggregation2 | PreSvcConfig1 |
| 3300 | EventHandler | PreSvcConfig2 |
| 3200 | Association1 | PreSvcConfig2 |
| 3400 | Subclass2 | ServiceConfigurator |
| 3700 | Acceptor | ServiceConfigurator2 |
| 3800 | Subclass3 | ServiceConfigurator2 |
| 3900 | ServiceHandler | ServiceConfig2+ |
| 4000 | Creation | ServiceConfig2+ |
| 4100 | Subclass2 | ServiceConfig2++ |




AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |
| 2600 | Association2 | PreReactor2 |
| 1800 | Subclass1 | Reactor |
| 500 | ServiceHandler | Reactor |
| 1800 | Subclass2 | Reactor |
| 500 | ServiceObject | Reactor |
| 501 | Subclass3 | Reactor |
| 500 | Acceptor | Reactor |

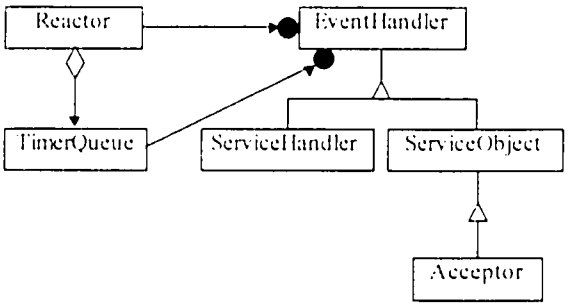


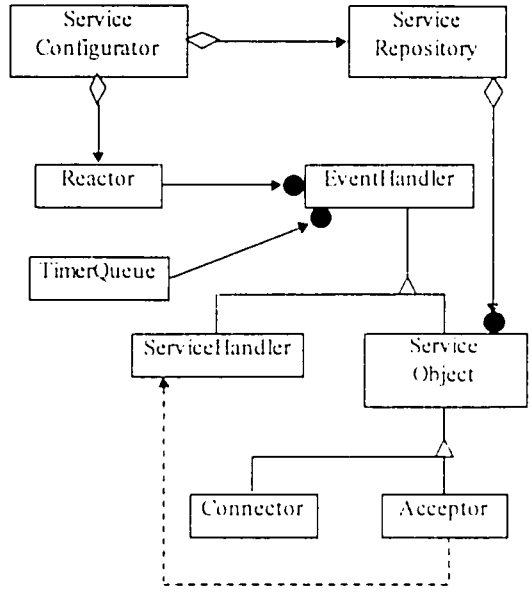

*Figure 43: DEPARS creates new templates because it cannot match Creation anywhere. The first template assigns Creation and ServiceHandler in AHOS unique ids. The second template assigns Subclass2 a unique id in AHOS.*

We proceed by entering the last object, Connector. Again, DEPARS creates a new analysis history for this object. We then enter the subclass link from ServiceObject to

Connector. DEPARS recognizes Subclass4 and Connector as a duplicated part in ServiceConfig2++ pattern in AHOS. It also recognizes Subclass4 and Connector as a duplicated part in Reactor pattern in AHEH.

AHSO

| Id | Component | Template |
|---|---|---|
| 100 | ServiceObject | Unification |
| 200 | ServiceRepository | Association |
| 1500 | Aggregation4 | 1:N Connection |
| 2900 | ServiceConfigurator | 1:N Connection2 |
| 2800 | Aggregation1 | 1:N Connection2 |
| 3100 | Reactor | PreSvcConfig1 |
| 3000 | Aggregation2 | PreSvcConfig1 |
| 3300 | EventHandler | PreSvcConfig2 |
| 3200 | Association1 | PreSvcConfig2 |
| 3400 | Subclass2 | ServiceConfigurator |
| 3700 | Acceptor | ServiceConfigurator2 |
| 3800 | Subclass3 | ServiceConfigurator2 |
| 3900 | ServiceHandler | ServiceConfig2+ |
| 4000 | Creation | ServiceConfig2+ |
| 4100 | Subclass2 | ServiceConfig2++ |
| 3700 | Connector | ServiceConfigurator2 |
| 3800 | Subclass4 | ServiceConfigurator2 |




AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |
| 2600 | Association2 | PreReactor2 |
| 1800 | Subclass1 | Reactor |
| 500 | ServiceHandler | Reactor |
| 1800 | Subclass2 | Reactor |
| 500 | ServiceObject | Reactor |
| 501 | Subclass3 | Reactor |
| 500 | Acceptor | Reactor |
| 501 | Subclass4 | Reactor |
| 500 | Connector | Reactor |

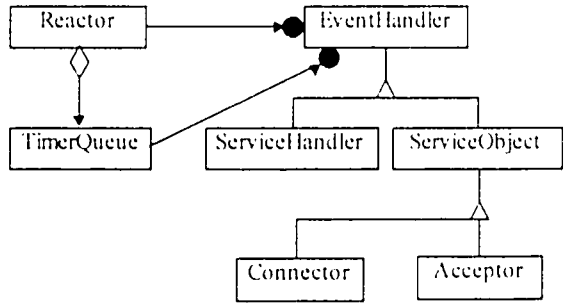
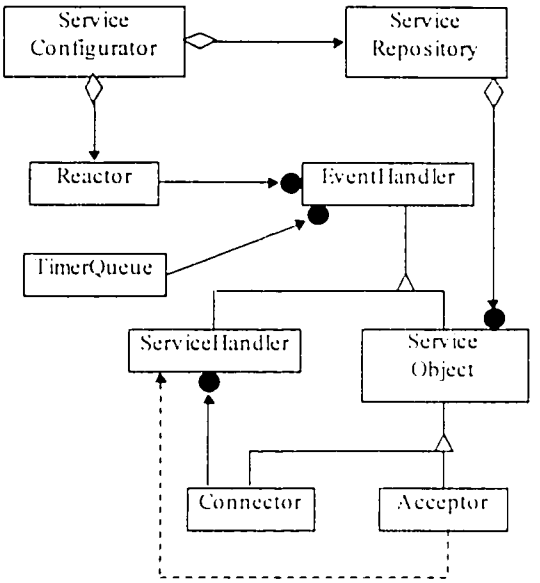



*Figure 44: DEPARS recognizes Connector and Subclass4 as a duplicated part in both AHOS and AHEH.*

Finally, we enter the one-to-many association relation between Connector and ServiceHandler. Again DEPARS cannot recognize this relation. And because the relation is not covered by any template with more than three components, DEPARS creates a new template to cover it. ServiceConfig++ is chosen to be the base of the new template because it has the best component coverage. DEPARS creates a new template named ServiceConfig2+++. After the creating the template, DEPARS tries to match the object model to the new template. DEPARS stops here because all components in the object model are covered by one or more patterns with more than three components. Since only AHSO and AHEH contain patterns with more than three components, DEPARS concludes that the object model consists of Service Config2+++ pattern and Reactor pattern.

AHSO

| Id | Component | Template |
|---|---|---|
| 100 | ServiceObject | Unification |
| 200 | ServiceRepository | Association |
| 1500 | Aggregation4 | 1:N Connection |
| 2900 | ServiceConfigurator | 1:N Connection2 |
| 2800 | Aggregation1 | 1:N Connection2 |
| 3100 | Reactor | PreSvcConfig1 |
| 3000 | Aggregation2 | PreSvcConfig1 |
| 3300 | EventHandler | PreSvcConfig2 |
| 3200 | Association1 | PreSvcConfig2 |
| 3400 | Subclass2 | ServiceConfigurator |
| 3700 | Acceptor | ServiceConfigurator2 |
| 3800 | Subclass3 | ServiceConfigurator2 |
| 3900 | ServiceHandler | ServiceConfig2+ |
| 4000 | Creation | ServiceConfig2+ |
| 4100 | Subclass2 | ServiceConfig2++ |
| 3700 | Connector | ServiceConfigurator2 |
| 3800 | Subclass4 | ServiceConfigurator2 |
| 4200 | Association3 | ServiceConfig2+++ |




AHEH

| Id | Component | Template |
|---|---|---|
| 100 | EventHandler | Unification |
| 200 | Reactor | Association |
| 1500 | Association1 | 1:N Association |
| 800 | TimerQueue | PreReactor |
| 3500 | Aggregation3 | PreReactor |
| 2600 | Association2 | PreReactor2 |
| 1800 | Subclass1 | Reactor |
| 500 | ServiceHandler | Reactor |
| 1800 | Subclass2 | Reactor |
| 500 | ServiceObject | Reactor |
| 501 | Subclass3 | Reactor |
| 500 | Acceptor | Reactor |
| 501 | Subclass4 | Reactor |
| 500 | Connector | Reactor |

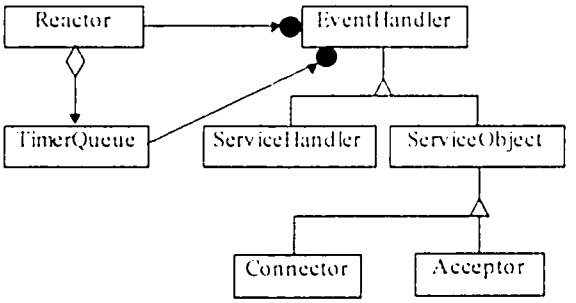



*Figure 45: DEPARS creates a new template to include Association3 and concludes that the object model is covered by ServiceConfig2+++ pattern and Reactor pattern.*

# 3. Related Work

The work of DEPARS was inspired by the concept of design pattern. Design patterns had rooted since 1970s, but only gained popularity in Object Oriented community in the recent years. Section 3.1 discusses related researches of design patterns. DEPARS promotes design reuse not only by recognizing known design patterns but also by detecting new patterns for future reuse. Section 3.2 discusses other systems that also promote design reuses. DEPARS uses a knowledge-based approach to retrieve design patterns. It recognizes existing design patterns by matching components in an object model diagram to that in its knowledge base template and then retrieves information of the recognized design pattern. Several other knowledge-based design systems are discussed in Section 3.3. DEPARS detects new patterns as the designer is working on his/her new design or from an existing design. The process of extracting a higher level of abstraction like design patterns from a low-level abstraction like object models is considered reverse engineering. Section 3.4 introduces reverse-engineering technique and discusses some examples of reverse-engineering tools. Finally, Section 3.5 discusses the automatic code generation tool by F.J. Budinsky, et al. and Frame by Paul Bassett, since they do not fit in the above categories.

## *3.1 Design Patterns*

The concept of design pattern originated from the work of Christopher Alexander in the 1970's. The design patterns he discovered are patterns in buildings and towns instead of software. Alexander published the book "A Pattern Language" in 1977 [Alexander et al., 77], and the second book "The Timeless Way of Building" in 1979 [Alexander, 79]. Both books presented a way of representing architectural knowledge as a series of problems, constraints on each problem, a solution to the problem along with a template or example that can be followed to solve the problem.

In 1987, Kent Beck and Ward Cunningham started to apply the concept of design pattern in software engineering and developed a simple pattern language for

designing graphical user interfaces in Smalltalk. The work was presented in OOPSLA '87.

Erich Gamma and Richard Helm continued the work. Together with Ralph Johnson and John Vlissides, they collected a catalog of common design patterns, which was published in "Design Patterns" [Gamma et al., 95]. Most of the design patterns in "Design Patterns" are captured in DEPARS's knowledge base. (See Appendix A).

### 3.1.1 Networking Design Patterns

Douglas Schmidt et al. have identified several networking design patterns using the template suggested by Gamma et al. The patterns are: Acceptor, Connector, Asynchronous Completion Token, Thread-Specific Storage, and Service Configurator. These design patterns are captured in DEPARS's knowledge base. (See Appendix A). The following is the summary of these networking design patterns extracted from various articles written by Schmidt et al.

#### *3.1.1.1 Reactor*

Reactor pattern decouples event demultiplexing and event handler dispatching from the services performed in response to events [Schmidt, Dec 96].

Use the pattern if:

- There is a need to demultiplex multiple types of events from multiple sources of events efficiently within a single thread of control
- There is a need to extend application behavior without requiring changes to the event dispatching framework.

Figure 46 illustrates the structure of Reactor pattern.

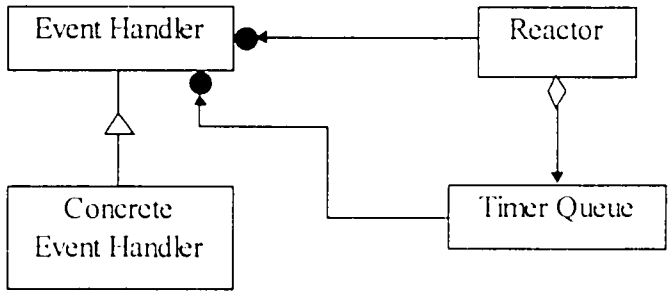



*Figure 46 Structure of Reactor pattern.*

Reactor defines an interface for registering, removing, and dispatching Concrete Event Handler objects. An implementation of this interface provides a set of application-independent mechanisms. These mechanisms perform event demultiplexing and dispatching of application-specific event handlers in response to events (such as input, output, signal and timer events).

Event Handler specifies an abstract interface used by the Reactor to dispatch callback methods defined by objects that register to events of interest. Each Concrete Event Handler selectively implements callback method(s) to process events in an application-specific manner.

### *3.1.1.2 Acceptor*

Acceptor pattern decouples the passive initialization of a service from the tasks performed once a service is initialized. This enables the creations of reusable, extensible and efficient network services [Schmidt, 95].

Use the pattern if:

- The behavior of a network service does not depend on the steps required to passively initialize a service;
- Connections may arrive concurrently from different peers, but blocking or continuous polling for incoming connections on any individual peer is inefficient.

Figure 47 illustrates the structure of Acceptor pattern.

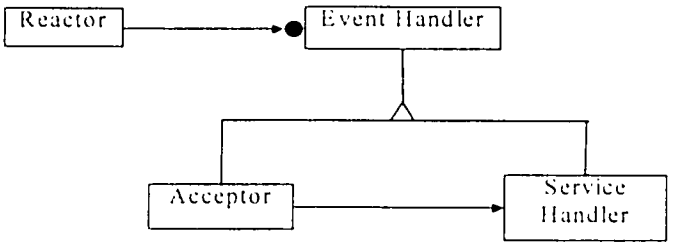



*Figure 47: Structure of Acceptor pattern.*

Reactor defines an interface for registering, removing, and dispatching Event Handler objects. It provides a set of application independent mechanisms that perform event demultiplexing and dispatching of application-specific event handlers in response to events.

Event Handler specifies an interface that the Reactor uses to dispatch callback method defined by objects that are pre-registered to handle events. These events signify conditions such as a new connection request or the arrival of data from a connected peer.

Service Handler provides a generic interface for processing services.

Acceptor implements the generic strategy for passively initializing network services.

### *3.1.1.3 Connector*

Connector pattern decouples the active initialization of a service from the tasks performed once a service is initialized. This enables the creations of reusable, extensible and efficient network services [Schmidt, 96].

Use the pattern if:

- The behavior of a network service does not depend on the steps required to actively initialize a service;
- The application must establish a large number of connections with peers connected over long-delay networks.

Figure 48 illustrates the structure of Connector pattern.

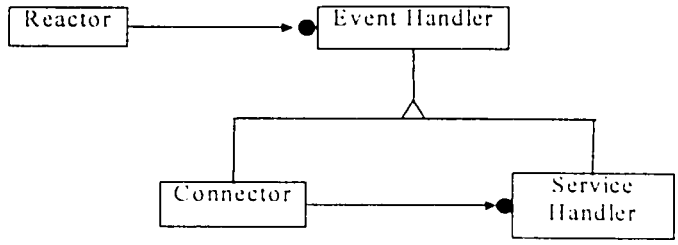



*Figure 48: Structure of Connector pattern.*

Service Handler defines a generic interface for a service. It contains a communication end point that encapsulates an I/O handle. This end point is used to exchange data between the Service Handler and its connected peer.

Connector connects and activates a Service Handler. It implements the strategy for actively connecting the Service Handler with its remote peer.

Reactor handles the completion of connections that were initialized asynchronously. It allows multiple Service Handlers to have their connections initiated and completed asynchronously by a Connector configured within a single thread of control.

Event Handler specifies an interface that the Reactor uses to dispatch callback method defined by objects that are pre-registered to handle events. These events signify conditions such as a new connection request or the arrival of data from a connected peer.

### *3.1.1.4 Asynchronous Completion Token*

Asynchronous Completion Token pattern efficiently associates state with the completion of asynchronous operations [Harrison et al., 96].

Use the pattern if:

- The service performs client requests asynchronously. If the operations are synchronous, there may be no need to provide an explicit hook for regaining state since the state can be implicit in the activation record where the client blocks.

- The notification provided by the service upon the completion of an asynchronous operation is not sufficient for the client to uniquely identify the operation.
- The service knows nothing about the application-specific nature of the clients.

Figure 49 illustrates the structure of Asynchronous Completion Token pattern.

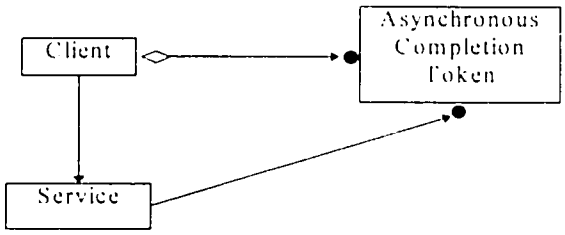



*Figure 49: Structure of Asynchronous Completion Token pattern.*

Client performs requests for asynchronous operations on the Service. It requires application-specific state, along with the completion notification, to correctly handle asynchronous events.

Service provides some type of asynchronous task to Clients.

Asynchronous Completion Token is given by Clients to Services to be returned on completion of asynchronous operations.

### *3.1.1.5 Thread-Specific Storage*

Thread-Specific Storage pattern's intent is to allow multiple threads to use on logical access point to retrieve thread local data without incurring locking overhead for each access [Harrison and Schmidt, 97].

Use the pattern if:

- The application contains multiple preemptive threads of control that can execute concurrently in an arbitrary scheduling order, and
- Each thread of control contains a sequence of operations that share data common only to that thread, and

- That data must be accessed through a global object that is "logically" shared with other threads, but is "physically" unique for each thread.

Figure 50 illustrates the structure of Thread-Specific Storage pattern.

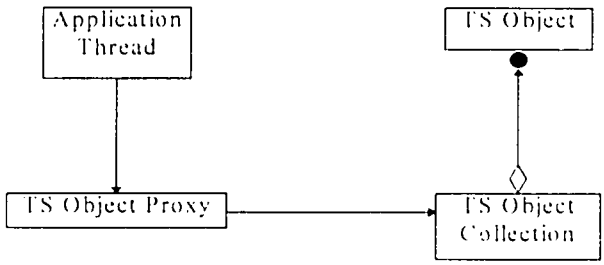



*Figure 50: Structure of Thread-Specific Storage Pattern*

Application Threads use Thread-Specific Object Proxy to access the thread-specific storage in Thread-Specific Objects.

Thread-Specific Object Proxy provides an interface to a Thread-Specific object. It is responsible for providing access to a unique object for each calling thread.

Thread-Specific Object Collection contains a set of all Thread Objects belonging to a particular thread. The collection maintains a mapping of keys to Thread-Specific Objects for indirect access from a Thread-Specific Proxy.

Thread-Specific Object is a particular thread's instance of a thread-specific object.

### *3.1.1.6 Service Configurator*

Service Configurator pattern is intended to decouple the behavior of services from the point in time at which service implementations are configured into an application or system [Jain and Schmidt, 97].

Use the pattern when:

- Services must be initiated, suspended, resumed, and terminated dynamically, and
- The implementation of a service may change, but its configuration with respect to related services remains the same and/or the configuration of a group of collocated services may change, but their implementations remain the same, or

- An application or system can be simplified by being composed of multiple independently developed and dynamically configurable services, or
- The management of multiple services can be simplified or optimized by configuring them using a single administrative unit.

Figure 51 illustrates the structure of Service Configurator pattern.

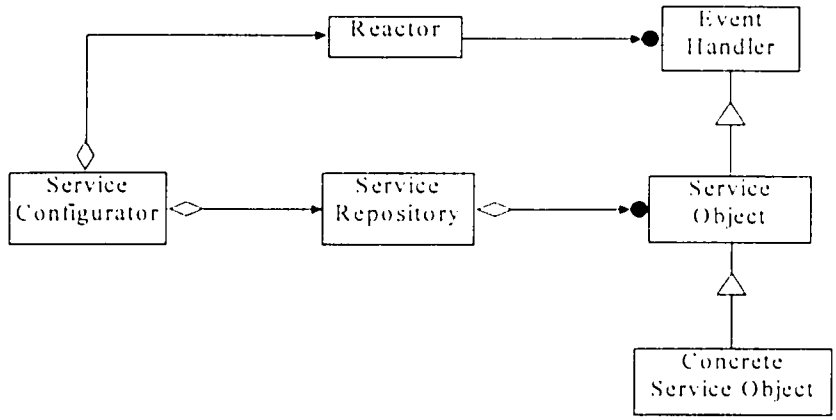



*Figure 51 Structure of Service Configurator pattern*

Service Object specifies the interface containing hook methods used by a Service Configurator-based application to dynamically configure the Service Object.

Concrete Service Object implements the service's hook methods and other service-specific functionality.

Service Repository maintains a repository of all the services offered by a Service Configurator-based application. It allows administrative entities to centrally manage and control the behavior of the configured services.

## 3.2 Design Reuse Systems

The reuse of analysis and design is believed to be much more powerful than the reuse of code. As James M. Neighbors has pointed out by examining the cost of the analysis and design phases in the software life cycle and the cost of correcting a mistake made in these phases, it is not difficult to see the importance of successful design reuse

[Neighbors, 84]. The examples of researches on design reuse include [Cheng et al., 94] and [Hunt and Lee, 91].

### 3.2.1 VisualSpecs

VisualSpecs, developed by Betty H.C. Cheng, et al., enables a user to perform object-oriented analysis graphically using the OMT notations [Cheng et al., 94]. With the object model the user generated after the analysis, VisualSpecs generates a formal specification of the object model. The specification can then be systematically analyzed for completeness and consistency prior to implementation. VisualSpecs's graphical environment facilitates the development of reliable software using formal methods, enables automated processing of requirements and design information, and promotes software design reuse based on graphical notations. OMT notations are visually oriented and easy to use. Similar to VisualSpecs, DEPARS also takes the advantages of using OMT. Cheng claims that VisualSpecs promotes software design reuse because it utilizes OMT. Cheng believes OMT notations allow the user to comprehend an object model easily and quickly and hence he/she may apply similar design in new software. DEPARS, on top of using OMT notations, can detect reoccurring patterns in object models automatically and provides information related to the patterns to users. With the information, users can learn about patterns quickly and use them effectively in new designs.

### 3.2.2 MADOC

The MADOC system, developed by John Hunt and Mark Lee, assists its user in designing electrical circuits [Hunt and Lee, 91]. The MADOC system allows the user to: specify the design requirements, retrieve existing designs, analyze the functionality of an existing or modified design, analyze structurally the electrical behaviour of any design, modify or create a design of an electrical circuit and record a design. Like DEPARS, the MADOC system stores existing designs as cases and uses knowledge-based approach in finding a similar design.

## 3.3 Knowledge-Based Design Systems

A knowledge-based design system uses two distinct kinds of knowledge: design knowledge, which defines the space of possible designs, and domain knowledge, which serves to evaluate, analyze and modify designs [Faltings et al., 91]. There are five types of techniques applied commonly to knowledge-based systems: constraint satisfaction methods, logical reasoning methods, decomposition methods, plan-based methods and case-based methods [Hunt and Lee, 91]. Constraint satisfaction methods find a solution by propagating constraints through the design. The automatic automobile power transmission design system, by Bernard A. Nadel and Jiang Lin [Nadel and Lin, 91], and the telephone switch system configuration expert system by Alois Haselbock and Markus Stumptner [Haselbock and Stumptner, 91] are both examples of knowledge-based systems applied constraint satisfaction methods. Decomposition methods break a required design down into a number levels of simpler subdesigns. The subdesigns are then analyzed and applied appropriate actions. They are then recomposed into higher level of designs and eventually into the whole design [Hunt and Lee, 91]. R1/XCON, a configuration design system, by McDermott is an example of a knowledge-based system applied decomposition methods [McDermott, 82]. Plan-based methods involve using design plans that contain compiled knowledge of how a design can be decomposed into subdesigns, how the constraints on the design can be transformed into constraints on the subdesigns and how the solutions of the subdesigns can be recomposed into a solution for the desired design [Hunt and Lee, 91]. AIR-CYL by Brown and Chandrasekaran is an example of a design system using plan-based methods [Brown and Chandrasekaran, 89]. Case-based methods retrieve existing designs from the database, adapt the design(s) to the current requirements and store the new design back into the database of existing designs. The adaptation process is usually controlled by heuristic knowledge, however the evaluation of the adapted designs may be performed heuristically or using first principles analysis [Hunt and Lee, 91]. Research into case-based methods includes [Hunt and Lee, 91], [Goel and Chandrasekaran, 89], [Pu and

Reschberger, 91], [Mahera and Zhang, 91] and [Dyer et al., 86]. DEPARS also uses this technique to recognize patterns in object model diagrams.

## 3.4 Reverse Engineering

One of the major tasks in reverse engineering is to develop or extract higher level description of a software system from its lower level source code description [Rich and Waters, 88] [Sneed and Jandrasics, 89]. One of DEPARS's features is detecting design patterns in existing object models. Since design patterns are regarded as a higher level of abstraction than object models, this process is also considered a reverse-engineering process. There are three types of common reverse-engineering tasks: re-documentation, restructuring and design recovery. Re-documentation is the production of a semantically equivalent representation of the target system at whatever level of abstraction is being addressed. Typically documentation tools are used on existing source code to produce data flow diagrams, data models, structure charts or text-based process definitions [Frazer, 92]. Examples of re-documentation tools are [Jandrasics, 81], [Antonini et al., 87], [Foster and Munro, 87] and [Boldyreff and Zhang, 92]. Restructuring is the transformation from one representation of a system into another without altering the system meaning or functionality. Most commonly it is applied to source code in order to make it conform better to structured programming principles [Frazer, 92]. An example of this type of research is done by [Arnold, 86]. Design recovery is the process whereby an approximation of the design of a target system is achieved. It recreates design abstractions from a combination of all known system information (i.e. source code, documentation, etc.). It attempts to produce the design elements which could reasonably be expected to have been produced as deliverables during the forward engineering process [Frazer, 92]. Examples of this area of research are: [Arango et al., 85], [Biggerstaff, 89], [Chen and Ramamoorthy, 86], [Ambras et al., 88], [Parnas and Clements, 86], [Rich and Waters, 88], [Ward, 88], KT [Brown, 97], and AutoSpecs [Cheng and Gannod, 90]. DEPARS also falls into this category. We will focus our discussion on KT and AutoSpecs since they are similar to DEPARS.

### 3.4.1 KT

KT, a design pattern detection tool developed by Kyle Brown offers similar functionality in the area of design reverse-engineering as DEPARS [Brown, 97]. KT detects three design patterns, Composite, Decorator and Template Method discussed in "Design Patterns" by Gamma et al. It takes object models, dynamic model, e.g. dynamic flow of messages, class definitions and method definitions as input from the user. Some user interactions may be required as KT tries to derive instance variable type later on in the detection process. In order to detect Composite and Decorator patterns, KT examines each class in an object model one by one. For each class, each of its relations is traversed. If the relation points to an ancestor class, the class is a participant of either a Composite or Decorator pattern. The class is a participant of a Composite pattern if the relation is a one-to-many relation. The class is a participant of a Decorator pattern if the relation is a one-to-one relation. To detect Template Method, KT scans for virtual methods by looking in each class's method dictionary. It then scans the method dictionary for methods that calls any of the virtual methods found in the first scan. These methods are the template methods. KT applies these two search algorithms to detect Composite, Decorator and Template Method pattern. As a result, the tool generates a list consisting of names of classes that participated in a Composite or Decorator pattern, names of template methods and virtual methods, total number of occurrences of Composite pattern, total number of occurrences of Decorator pattern and total number of template methods and virtual methods.

DEPARS, similar to KT, detects design patterns in existing designs, but it uses a different approach. Unlike KT, DEPARS requires only object models as its input. This substantially cuts down the effort required from the user for using DEPARS. DEPARS recognizes patterns by matching the structure of the object model to that of the design templates with the guidance of the knowledge base and matching rules. Unlike KT, the algorithm of DEPARS is very simple, fast and flexible. It can recognize patterns with unique structures with reasonable accuracy in a time hardly noticeable by the user. DEPARS learns new pattern structure automatically without the need of a new search algorithm. KT, on the other hand, performed poorly in

detecting the pattern Chain of Responsibility even with a search algorithm Brown designed specifically to detect the pattern. DEPARS provides not only the names of the classes involved in a pattern, it also shows how the classes form the pattern. The information regarding to the pattern is also shown to the user as references. It is with the pattern information we actually have a higher level of abstraction than the existing object model. Simply listing the occurrences of a certain patterns in an object model does not help those designers who do not know the patterns.

### 3.4.2 AutoSpec

AutoSpec is a formal specifications abstraction tool developed by Betty H.C. Cheng and Gerald C. Gannod [Cheng and Gannod, 90]. Autospec abstracts formal specifications from program code. The abstraction process incorporates domain-specific information supplied interactively by the user, as necessary. The input to AutoSpec are programs written in Dijkstra's imperative programming language, which contains type declarations in the header, sequential statements and no references to global variables. AutoSpec outputs the specifications in the form of predicate logic assertions, which annotates the original program code. The form of predicate logic assertion varies depending on the type of the statement. There are three types of statements: assignment, alternative and iterative. For example, given an assignment statement of the form x:= e; and a precondition *U*, where *U* is a logical expression; the following predicate logic expressions are inserted:

```
{U}              /* precondition */
x := e;
{x = e ∧ U}      /* postcondition */
```

For every logical grouping of annotated code, AutoSpec verifies that the code satisfies the generated specifications.

Like AutoSpec, DEPARS utilizes symbolic methods in the process. DEPARS abstracts design patterns from existing designs for reuse, yet AutoSpec abstracts specifications from existing program code for verification.

## 3.5 Other Related System

### 3.5.1 Automatic Code Generation Tool

The automatic code generation tool by F.J. Budinsky, et al. automates the implementation of design patterns [Budinsky et al. 96]. The tool provides on-line hyper text references of design patterns for the user to browse and select. After the user selects a design pattern, the tool guides him/her through several templates prompting for information which it uses to generate a custom implementation of the pattern. The user supplies application-specific information for a given pattern, normally application-specific names for the participants in a pattern along with choices for the design trade-offs. Based on the input, the tool generates automatically all the pattern-prescribed code, such as class declarations and definitions that implement the pattern, currently in C++. The user then adds this code to the rest of his/her application, often enhancing it with other application-specific functionality.

Both DEPARS and the tool are based on the concept of design pattern. The tool focuses on assisting users implementing a particular design pattern, while DEPARS focuses on reusing design patterns. Similar to the tool, DEPARS provides on-line references of design patterns. In addition, DEPARS also assists its user in searching the desired pattern while the tool does not.

### 3.5.2 Frame

Frame is a "parts-oriented" programming language developed by Paul Bassett [Bassett, 97]. Software developed in Frame is built with "parts" or, like Bassett calls them, "frames". Each application has a root frame, which may be composed of several other frames. Each of these frames may in turn consist of other frames and so on, forming a tree-like hierarchy. A frame, stored in text format, typically contains these elements:

- a name and a short description.
- parameters with default values.

- a function definition, and/or a data structure definition.

- references to other frames as sub-components and commands specifying how to adapt these frames. A frame may consist of one or more of other frames, but cannot consist of itself directly or indirectly. For example, a frame named "plane" consists of other frames like "tail", "wings", "engines", "fuselage", etc. "Fuselage" in turn consists of "shells" and "cabin". However, "fuselage" cannot consist of "fuselage" or "plane". Lack of support of recursion is certainly a major limitation of Frame.

A frame reuses other frames by including them in its composition. A whole or a part of a frame may be reused depending on the command used by the adapting frame. The commands specify how the definition in the reused frame is to be instanced, selected, extended, modified, deleted and/or iterated. In essence, frames are like macros in other functional programming languages like C. During construction time, Frame processor replaces references to frames with the frames' definitions as "in-line" code. The output file, a file with expanded code, is then fed to a compiler to produce executables.

Frame focuses its reuse on source code level. As mentioned earlier, other frames higher in the frame hierarchy can reuse a piece of code in a frame. DEPARS promotes reuse of design patterns. As Neighbors pointed out in [Neighbors, 84], reuse of design is far more powerful in terms of effort, time and money.

### 3.5.3 Design Apprentices Using Critiquing Approach

Design apprentices using critiquing approach present to users a reasoned opinion about a design. Design apprentices recognize deficiencies in a design and communicate the observations to users. Some of the design apprentices also offer suggestions on how to improve the design. JANUS is a design apprentice using critiquing approach [Fischer and Giergensohn, 90] [Fischer et al., 90]. JANUS helps users in designing residential kitchens. JANUS has knowledge of "good" and "bad" kitchen designs. JANUS observes as users entering their designs. With the knowledge

JANUS can critique "bad" designs and offer suggestions to users on improving the designs.

DEPARS's knowledge base stores not only the design patterns we encourage to reuse. It can also store the design patterns that are considered "bad" designs and are therefore discouraged for reusing. The knowledge base stores a pattern's name, intent, motivation and structure, for example. (See Section 1.2 for details). More importantly, it can also store comments and alternative designs for a design pattern. DEPARS recognizes a "bad" design pattern the same way it recognizes a "good" design pattern. DEPARS then shows the user the comments and the alternative designs of the "bad" design pattern.

# 4. Empirical Evaluation

In order to show the effectiveness of the design pattern recognition in practice, DEPARS is tested with two different sets of object models. The first test set contains a total of 20 object models. The object models resemble the patterns stored in DEPARS's knowledge base. That means, if it recognizes a pattern in one of these object models, the pattern should match exactly one of the patterns in the knowledge base. This set of object models shows how DEPARS may perform in the ideal situation. The second test set contains 20 object models. These object models include the designs of a WYSIWYG, "What-You-See-Is-What-You-Get", document editor. They also include the designs of a web server designed by Doug Schmidt and the designs of DEPARS. This set of object models shows how DEPARS may perform in practice.

There are five controllable parameters. Each may influence the outcome of DEPARS's analysis process. The parameters are: data entry sequence, operating modes, knowledge base, minNumComp and maxUnmatch. We mentioned minNumComp and maxUnmatch in Section 2.2.3.4. MinNumComp sets the minimum number of components a pattern template must have in order to be considered significant towards the coverage of an object model. MinNumComp is set to the default value, three, for the evaluation. MaxUnmatch sets the maximum number of unmatched components allowed in an object model. It is also set to the default value, zero, for the evaluation. The first three parameters are described in details later. Several tests are run with different combinations of these three parameter values to show the impact of these parameters on the analysis results.

In order to find out how the data entry sequence may affect the results, each object model in the first test set is entered twice in different sequences. The first time, each object model is entered in the same order as one of the ontology paths. The second time, each object model is entered in the reverse order of the path. For the second test set, the components in the object models are entered in no particular order.

DEPARS is capable of running in two different operating modes, incremental and non-incremental. In incremental mode, it analyzes an object model as soon as the user enters a relation. In non-incremental mode, it analyzes an object model only when it is instructed by the user to do so. In order to show how the operating mode may influence the outcomes, the test sets are evaluated twice in the two modes. When testing DEPARS in the non-incremental mode, it is instructed to analyze an object model after it is completely entered.

In order to show how effective DEPARS is in recognizing known patterns, it is evaluated with a complete knowledge base (see Appendix A). The knowledge base contains all the patterns, including design patterns from Schmidt et al., found in the test sets. DEPARS is expected to recognize these patterns in the given object models. The evaluation is run twice in the two different operating modes to show the impact of the data entry order.

In order to show how effective DEPARS is in detecting new patterns, it is evaluated with a minimum knowledge base (see Appendix B). The knowledge base does not contain any of the patterns in the test sets. DEPARS is expected to detect these patterns and store them in the knowledge base. Again, this evaluation is executed twice in the two different operating modes.

## 4.1 Results of the Evaluation

### 4.1.1 Results of Known Pattern Recognition Tests

The first round of evaluations shows how accurately DEPARS recognizes patterns. This means DEPARS is tested with a complete knowledge base, so the patterns in the test sets are already known to DEPARS. In order to recognize a design pattern with perfect accuracy, DEPARS must identify all the components involved in the design pattern. The accuracy is the percentage of the number of correctly matched components against the total number of involved components.

The first four tests evaluate DEPARS's recognition performance under the ideal environment. This means the design patterns in the test set should match exactly to

the templates in the ontology. The first test set is used for all four tests. For the first two tests, the design patterns in the test set are entered in the same sequence as that in the ontology. Test 1 is run in the incremental mode. Test 2 is run in the non-incremental mode. As expected, DEPARS recognizes all the patterns with 100% accuracy in both tests. The results of the two tests indicate that DEPARS can recognize the exact patterns stored in its knowledge base entered in the fashion that requires the minimum search effort in both operating modes.

Tests 3 and 4 use the same test set as the first two tests, but the design patterns in the test set are entered in the reverse order of ontology paths. Test 3 is run in the incremental mode and Test 4 is run in the non-incremental mode. As expected, DEPARS recognizes all the patterns with 100% accuracy in both tests. The results of the two tests indicate that the entry order has little impact on the outcome of DEPARS's recognition process.

The next two tests evaluate how DEPARS performs in practice. In these two tests, the second test set is used instead of the first test set. Both tests are conducted with the complete knowledge base. The components of the object models in the test set are entered in no particular order. Test 5 is run in the incremental mode and Test 6 is run in the non-incremental mode. As expected, DEPARS had a higher accuracy in recognizing patterns when running in the non-incremental mode. Nevertheless, the accuracy of DEPARS running in the incremental mode is still quite satisfying. It only failed to match patterns with 100% accuracy in one object model out of 15. The object model is recognized with 89% accuracy.

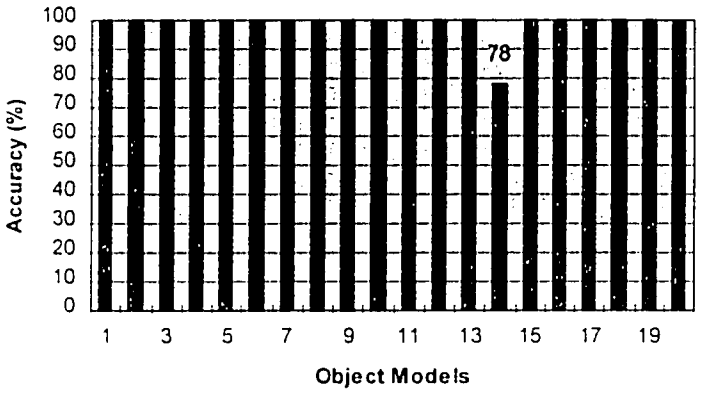



*Table 3 TEST 5 Test Set 2. Incremental Mode. Complete Knowledge Base. Components Entered in no Particular Order*

In the case of object model 14 in Test 5, DEPARS is expected to recognize the Proxy pattern in the object model. The component DEPARS needed in order to recognize the Proxy pattern is not entered until the end. DEPARS pursues another path that is considered to be the best at the time and erroneously recognizes the Adapter pattern instead. However since both Proxy and Adapter share many common structural characteristics[2], DEPARS scores a 78% accuracy by recognizing the Adapter pattern instead of the Proxy pattern.

[2] Both the structures of the Proxy pattern and the Adapter pattern are derived from the structure of the Pluggable Adapter, i.e. the structures of the Proxy pattern and the Adapter pattern extend from that of the Pluggable Adapter.

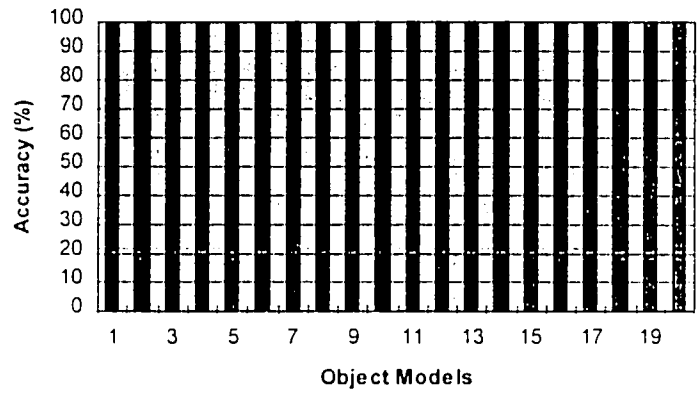



*Table 4. TEST 6 Test Set 2. Non-Incremental. Complete Knowledge Base. Components Entered in no Particular Order*

Test 6 is run in the non-incremental mode. The result is very satisfactory. All object models in the second test set are recognized with 100% accuracy.

### 4.1.2 Results of New Pattern Detection Tests

The next six tests evaluate DEPARS's accuracy in detecting new design patterns. DEPARS is run with the minimum knowledge base, so the design patterns to be detected are unknown to DEPARS. In these tests, DEPARS is required to detect the new pattern in an object model, create a template to store the new pattern and match the components in the object model to the new template. DEPARS fails to detect a new pattern if the new template created by DEPARS does not have the expected templates as the pre-conditions or if the new template contains erroneous components. The components that do not fulfill these conditions are considered missed in the analysis process. Those that do fulfill the conditions are considered matched. The accuracy is calculated based on the number of matched components in the total number of components in an object model.

The first four tests are run with the first test set. Each object model in the first test set consists of one design pattern with its key components only. They simulate the ideal input. In order to discover if the entry order of object model components may impact the detection outcome, the design patterns are entered in the order of ontology paths in Tests 7 and 8 and in the reverse order of ontology paths in Tests 9 and 10. To

compare the effectiveness of the two operating modes, incremental and non-incremental, Tests 7 and 9 are executed in the incremental mode while Tests 8 and 10 are executed in the non-incremental mode.

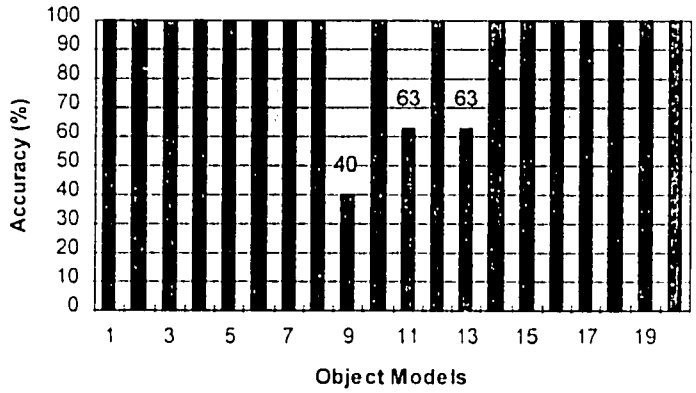



*Table 5: TEST 7: Test Set 1: Incremental Mode: Minimum Knowledge Base: Components Entered in the Order of Ontology Paths.*

Table 5 shows the results of DEPARS detecting new patterns in the first test set using the minimum knowledge base and running in incremental mode. The components in the object models are entered in the order of ontology paths. DEPARS creates the expected templates 13 times out of 16 for the new patterns that it detects. DEPARS manages to score at least 40% accuracy in those object models it failed to detect and create the expected templates.

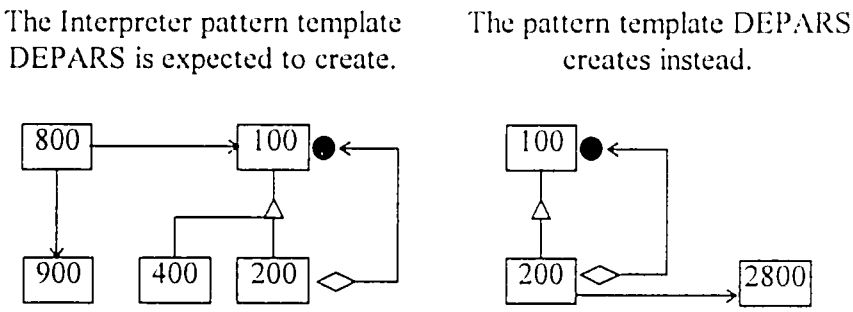



*Figure 55 Instead of creating the expected Interpreter pattern template, DEPARS creates the template on the right.*

In the case of object model 9, DEPARS is expected to detect and create the Interpreter pattern template. In this pattern template, concept 200 has a subclass relation and a

one-to-many aggregation relation with concept 100 and concept 800 has an association relation to concept 100. DEPARS over-generalizes concept 800 and matches it to concept 200. This causes DEPARS to create the new template after the erroneous pattern template and fail to generate the expected outcome (see Figure 55).

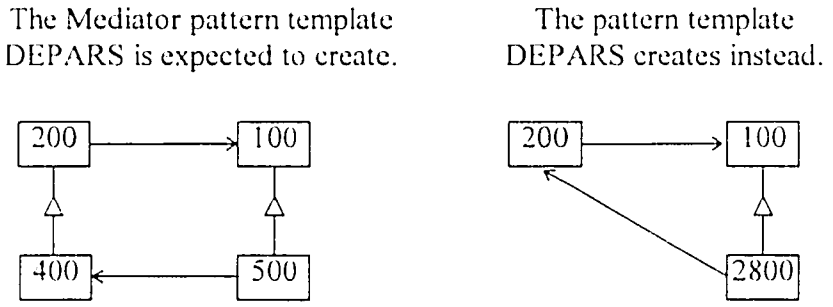



*Figure 56: DEPARS should have included the child concept of concept 200 in this template. The association relation from concept 2800 should also goes to the child concept instead of concept 200.*

In the case of object model 11, DEPARS is expected to detect and create the Mediator pattern template. According to the creation rules, if a child concept has a relation to either itself or other concepts, it becomes a key component and should be assigned a unique id, so it can be distinguished from its parent concept. However the rules do cover the case when the other concept has a relation to the child concept. When this occurs, the child concept should also become a key component and be assigned a unique id. This causes DEPARS not to create a vital delta which assigns a unique id to one of the child concepts in the Mediator pattern (see Figure 56). Hence DEPARS only scores 63% accuracy in this case.

In the case of object model 13, the same scenario occurs again causing DEPARS not to create the required delta. DEPARS again scores 63% accuracy in this case.

Test 8 is similar to Test 7 except it is run in the non-incremental mode. The test result is identical to that of Test 7. The three object models DEPARS fails to created the expected templates are object models 9, 11 and 13. The detection accuracy for these object models are 40%, 63% and 63% respectively. The failures are caused by the

same problems discussed previously. In these two tests, the operating mode plays an insignificant role in impacting the results.

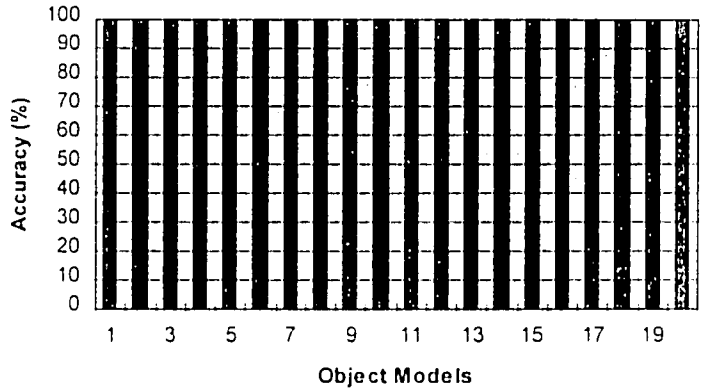



*Table 6: TEST 9. Test Set 1. Incremental Mode. Minimum Knowledge Base. Components Entered in Reverse Order of Ontology Paths.*

Tests 9 and 10 use the first test set as the test data. Again, the minimum knowledge base is used to ensure the patterns to be detected are unknown to DEPARS. The components in the test object models are entered in reverse order of ontology paths. Test 9 is executed in the incremental mode while Test 10 in the non-incremental mode. Table 6 shows the result of Test 9. All design patterns are detected and expected templates are created with 100% accuracy.

Because components in object model 9 are entered in a different order, DEPARS is able to bypass the problem we experience earlier in Tests 7 and 8 and successfully creates the expected pattern template.

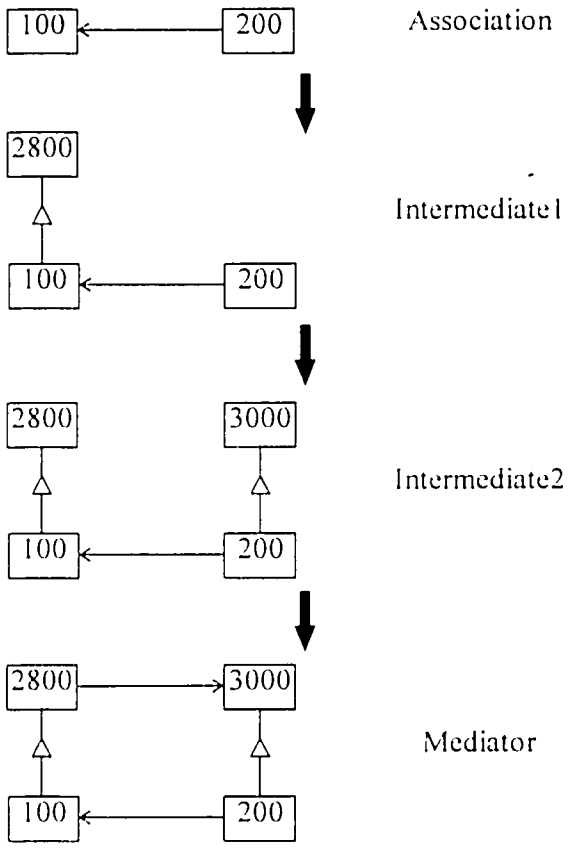



*Figure 57. DEPARS starts by matching to the Association pattern template, creates an intermediate pattern template by extending the Association pattern template. It then extends the first intermediate pattern template and creates a second intermediate pattern template. Finally it expends the second intermediate pattern template to create the Mediator pattern template. The bug does not affect the result because the extensions are added as parent concepts.*

In the case of object model 11, DEPARS matches the child concepts in the Mediator pattern to the Association pattern template, simply because they are entered first. It then creates new deltas to capture the characteristics of the parent concepts in the Mediator pattern (see Figure 57). The bug we experienced earlier with Tests 7 and 8 in the creation rules does not apply, since the deltas are not created to capture the characteristics of a child concept. The same applies to the case of object model 13.

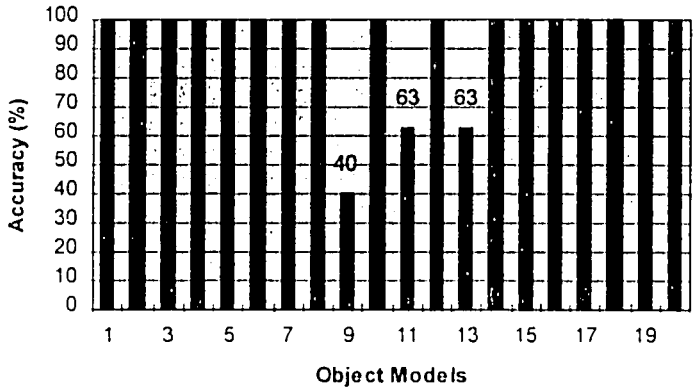



*Table 7: TEST 10. Test Set 1. Non-Incremental Mode. Minimum Knowledge Base. Components Entered in Reverse Ontology Paths.*

Table 7 shows the result of Test 10. The problems we have seen with object models 9, 11 and 13 in Test 8 return because Test 10 is run in the non-incremental mode. That means the component entry order has a minimum impact on the result.

By observing the previous four test results, the entry order of the components seems to have significant impact on the DEPARS's detection process in the incremental mode. It determines the sequence of deltas DEPARS creates. As mentioned in Section 2.2.3.5, the order of deltas affects the effectiveness of DEPARS's analysis process. The order of deltas becomes important if a template has more than one concepts sharing similar characteristics, e.g. they all have an association relation to the same concept or they are all child concepts of the same concept.

Tests 11 and 12 evaluates DEPARS's accuracy in detecting and creating new pattern templates using real life object models, i.e. the second test set. Again, DEPARS is run with the minimum knowledge base. Thus the patterns to be detected are unknown to DEPARS. The components of the object models are entered in no particular order, since we already know that the component entry order influences the outcome when DEPARS is operated in the incremental mode.

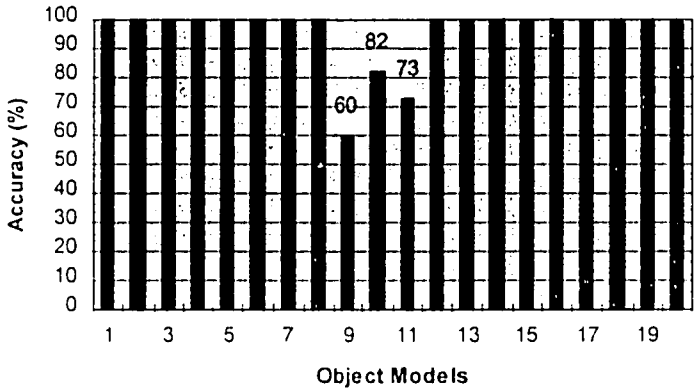



*Table 8 TEST 11 Test Set 2, Incremental Mode, Minimum Knowledge Base, Components Entered in no Particular Order*

Table 8 shows the results of Test 11 which the test set 2 is executed in the incremental mode. DEPARS successfully detects and creates the expected templates with 100% accuracy 13 times out of 15.

In the case of object model 9, DEPARS is expected to detect and create the Interpreter pattern template. The same over-generalizing problem we discovered earlier in Test 7 occurs again here.

In the case of object model 10, DEPARS is expected to detect and create the Iterator pattern template. The main structural characteristic of the Iterator pattern is that the child concepts have an association relation and a creation relation with each other. Because of the component entry order, one of the child concepts is generalized to match its parent concept, while the other child concept is matched to a unique concept. This makes DEPARS thinking that the association relation and the creation relation are actually between a child concept and a parent concept and creating the new template accordingly (see Figure 58).

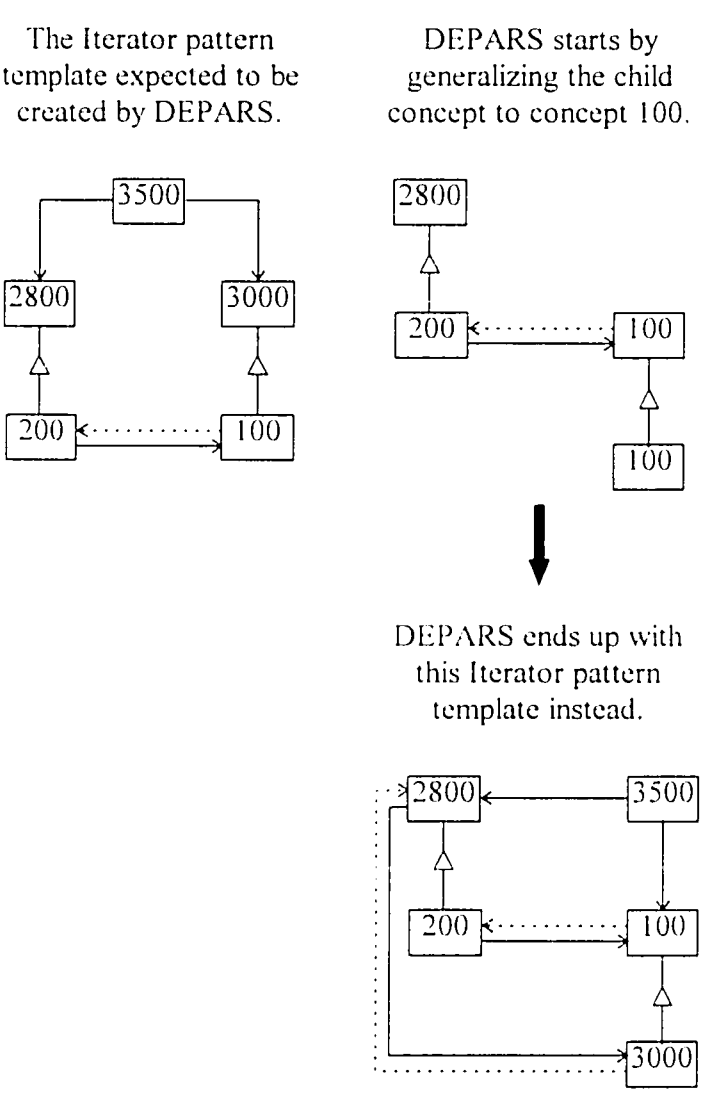



*Figure 58. Because of the component entry order, concept 3000 is initially generalized to match concept 100 and concept 200 is assigned the unique id right away. This makes DEPARS thinking that the association relation and the creation relation are actually between concept 100 and concept 200, instead of concept 200 and concept 3000. Hence it erroneously creates the deltas identifying those relations between the two concepts.*

In the case of object model 11, DEPARS is expected to detect and create the Mediator pattern template. The same scenario preventing DEPARS from creating the expected Mediator pattern template in Tests 7, 8 and 10 has taken effect again here.

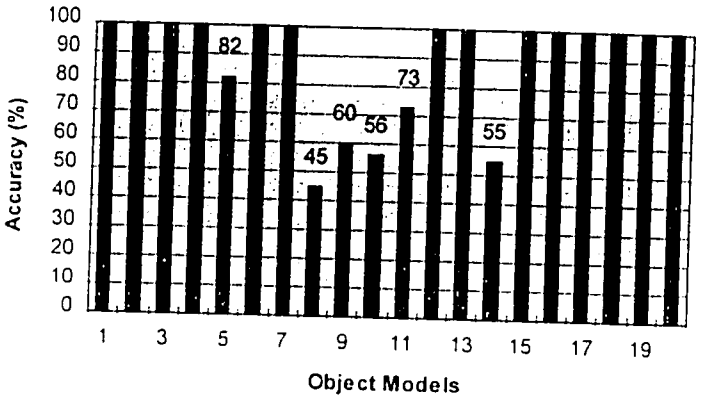



*Table 9: TEST 12 Test Set 2 Non-Incremental Mode, Minimum Knowledge Base Components Entered in no Particular Order*

Table 9 shows the result of Test 12, which is run in the non-incremental mode. DEPARS manages to detect and create the expected pattern templates with 100% accuracy nine times out of 15.

In the case of object models 5, 8, 9, 10 and 14, the errors are again caused by the same over-generalizing bug we discovered in previous tests.

In the case of object model 11, DEPARS is expected to detect and create the Mediator pattern template. The scenario that has occurred previously when creating the Mediator pattern template has reoccurred this time.

Tests 11 and 12 again shows DEPARS has higher accuracy in detecting new patterns when it is operated in the incremental mode. Because the component entry order influences the accuracy of DEPARS in detecting new patterns when it is run in the incremental mode, the component entry order becomes an important factor in determining the accuracy of the template creation process.

## 4.2 Summary of Evaluation Results

DEPARS's overall performance in recognizing known patterns and detecting new patterns is quite satisfactory.

- It can recognize known design patterns with 100% accuracy at least 97% of the time and can detect new design patterns with 100% accuracy at least 85% of the time.

- The frequency of detecting new design patterns with 100% accuracy is influenced by the component entry order if DEPARS is run in the incremental mode. The frequency of detecting new design patterns with 100% accuracy in the incremental mode is 85% when components are entered in the order of ontology paths and 100% when components are entered in the reverse order.

- The frequency of detecting new design patterns with 100% accuracy is higher when DEPARS is run in the incremental mode. DEPARS has a 95% accuracy in the incremental mode comparing to 90% in the non-incremental mode.

- The entry order of components in a diagram has little or no impact when DEPARS is in the non-incremental mode, but plays a more significant role in the incremental mode in detecting new patterns.

As mentioned previously, DEPARS has a higher accuracy detecting and creating new templates in the incremental mode. This also means the component entry order plays a significant role. Depending on the order the components are entered, different templates may be created. The templates created should only be regarded as suggestions from DEPARS for new design patterns. DEPARS users should be allowed to customize the content of the new template and its location in the ontology for better performance.

# 5. Conclusions

Design patterns promote software design reuse. But how to apply design patterns effectively is not something every designer knows. DEPARS is designed to help designers overcoming problems they may encounter in the process of applying design patterns in their designs. It provides a new browsing mechanism that allows designers to search for patterns by their structure. It helps designers to learn patterns by displaying information, such as the pattern's intent, motivation, applicability, structure, collaborations, consequences, other related patterns, etc. It promotes design reuse by detecting new patterns from new and existing object models. It also provides a higher level of abstraction than object models by recognizing patterns in them.

DEPARS uses a knowledge based approach to recognize and detect patterns in object models. It stores known design pattern structures in the form of templates. The templates are arranged based on their structural complexity. The templates are connected by deltas. A delta is the structural difference between two templates. The entire collection of templates and deltas is called ontology. During the analysis process, DEPARS traverses the ontology and tries to match the encountered templates to the components in an object model. The matched templates serve as starting points of the next analysis. When displayed in the recognition order, the matched templates explain how a certain design pattern is recognized. DEPARS stops the analysis process when the following two goals are achieved: 1. Identify all participants of a recognized design pattern. 2. Cover an object model with a minimum number of design patterns. If the combinations of existing design patterns cannot cover the entire object model, DEPARS creates new design patterns to cover it.

Design patterns are design concepts, not rigorous mathematical definitions. Therefore the actual implementations of a design pattern are subject to customizations and are not always identical. However they all share the same key characteristics, hence the pattern. When recognizing a design pattern in an object model, DEPARS does not expect the components in an object model to match exactly the components in a

template. Instead, it is the key characteristics that DEPARS looks for. In order to have a reasonable tolerance in matching, DEPARS applies matching rules during its analysis process. The rules allow relations of certain different types and/or cardinalities to match. They also allow concepts in a hierarchy to be generalized to one single concept.

DEPARS is evaluated to show its effectiveness in recognizing known design patterns and detecting new design patterns. The result of the evaluation is encouraging. DEPARS is able to recognize known design patterns with 100% accuracy at least 97% of the time. It can detect new design patterns with 100% accuracy at least 85% of the time. DEPARS's operating modes are found to influence the outcomes of new design pattern detection. The frequency of detecting new design patterns with 100% accuracy in the incremental mode is 95% while 90% in non-incremental mode.

## 5.1 Limitations of DEPARS

DEPARS has three major limitations. 1. It does not recognize patterns with multiple inheritance. 2. It cannot distinguish patterns with identical structure but for different purposes, e.g. State and Strategy patterns. 3. It cannot recognize combinations of design patterns as separate patterns if one of the overlapped concepts is assigned the id 100 (see 2.2.3 for details). While the usefulness of multiple inheritance is debatable, languages like C++ support it. Not supporting multiple inheritance may prevent DEPARS being used in the C++ environment. Storing related information of both patterns together in the same template can accommodate the second limitation. Treating these design pattern combinations as new design patterns and including them in the ontology compensates the third limitation.

## 5.2 Future Work

There are several areas DEPARS be improved. DEPARS needs a GUI to display the analysis results and pattern information to users. It can also be integrated with an object-modeling tool to facilitate diagram input. If the object-modeling tool utilizes a database system to store object models, it can be expanded to store the analysis results

along with the object models. The templates in DEPARS's knowledge base even can reference these object models as examples of how patterns are applied.

Currently DEPARS is not able to recognize or detect patterns with multiple inheritance. While this may not be an issue if the design is to be implemented in Smalltalk, DEPARS needs to support multiple inheritance to be more versatile.

DEPARS can be extended to be an active apprentice which tries to predict the user's final designs and gives advice based on its observations.

DEPARS's current learning capability is rather primitive. Features like supporting expert user type to train DEPARS and knowledge base editing would improve the quality of the templates in the knowledge base substantially.

### 5.2.1 Extend DEPARS to Analyze Dynamic Behavior of an Object

The dynamic behavior of an object is also an important characteristic of a pattern. Analyzing this information would improve DEPARS's accuracy in both recognizing known patterns and creating new templates. The dynamic behavior of an object is often captured by its methods. Therefore to analyze the dynamic behavior of an object, DEPARS must incorporate method recognition in its analysis process. To recognize a method, the method must be represented in a pattern template. To represent a method of an object in a pattern template, we need to create a new object Method (Figure 59). Each instance of Method represents a method of an object. Each instance of Concept may contain zero or more instances of Method. Method stores the name of the method it represents and a unique id. A Method is recognized in the same way as a Concept. During the analysis process, DEPARS would not only map the objects to the concepts in a template, but it would also map their methods to those of the concepts. For example, the user enters an object model. The object model contains two objects - AbstractConcept and ConcreteConcept, where ConcreteConcept is the subclass of AbstractConcept. Each of these objects have a method named do(). The template pattern DEPARS tries to match contains a single object, Unification. Unification contains one method, genericMethod(). After the user enters

AbstractConcept and its method in the object model, DEPARS maps AbstractConcept to Unification and do() to genericMethod(). After the user enters ConcreteConcept and its method, DEPARS maps ConcreteConcept to Unification by applying generalization rules. DEPARS maps ConcreteConcept's do() method to genericMethod() because ConcreteConcept maps to Unification.

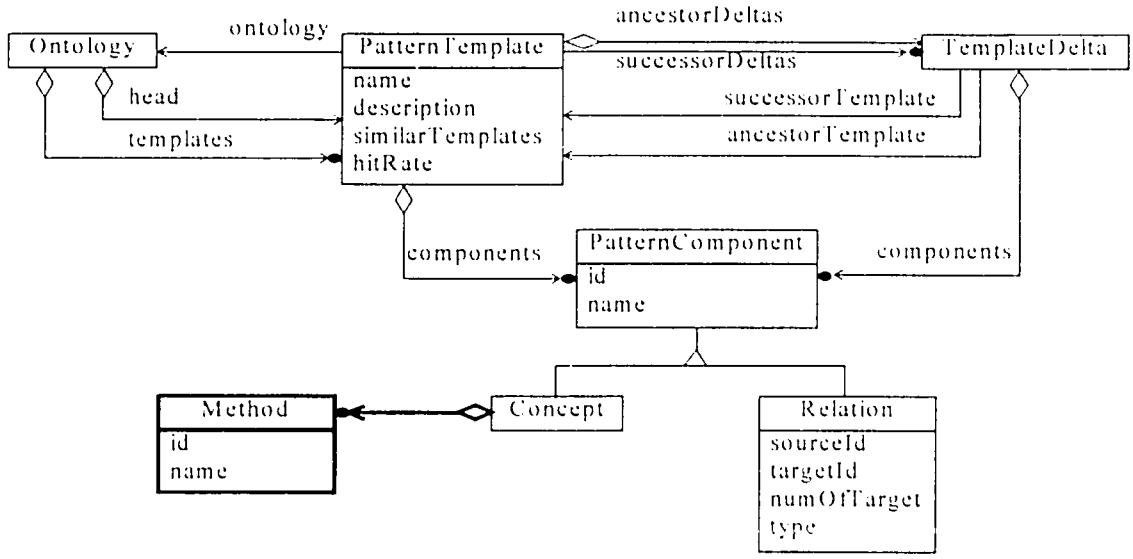



*Figure 59: Method is added as a part of a PatternTemplate. Method represents a method of an object.*

# 6. Bibliography


[Alberts et al., 91] L.K. Alberts, P.M. Wognum and N.J.I. Mars. A Systematic Approach to Knowledge-Based Design, Illustrated for Discrete Circuits Design in Electronics. IJCAI-91, Sydney, Australia, 1991.

[Albrechtsen, 92] H. Albrechtsen. Software Information Systems: Information Retrieval Techniques. Chapman & Hall, London, UK, 1992.

[Alexander et al., 77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A Pattern Language. Oxford University Press, New York, 1977.

[Alexander, 79] Christopher Alexander. The Timeless Way of Building. Oxford University Press, New York, 1979.

[Ambras et al., 88] James P. Ambras, Lucy M. Berlin, Mark L. Chiarelli, Alan L. Foster, Vicki O'Day and Randolph N. Splitter. MicroScope: An Integrated Program Analysis Tool. Hewlett-Packard Journal, August 1988.

[Antonini et al., 87] P. Antonini et al. Maintenance and Reverse Engineering: Low-Level Design Documents Production and Improvement. Conference on Software Maintenance. Austin, Texas, September, 1987.

[Arango et al., 85] G. Arango et al. Maintenance and Porting of Software by Design Recovery. Proceedings Conference on Software Maintenance. CS Press, Los Alamitos, California, 1985.

[Arnold, 86] Robert S. Arnold. Tutorial on Software Restructuring. IEEE Computer Society, 1986.

[Bassett, 97] Paul G. Bassett. Framing Software Reuse: Lessons from the Real World. Prentice Hall, Upper Saddle River, NJ, 1997.

[Biggerstaff and Richter, 87] T. Biggerstaff and C. Richter. Reusability Framework, Assessment and Directions. ACM Press, New York, 1987.

[Biggerstaff, 89] Ted J. Biggerstaff. Design Recovery for Maintenance and Reuse. IEEE Computer, Vol. 22, July 1989.

[Boldyreff and Zhang, 92] C. Boldyreff and J. Zhang. From Recursion Extraction to Automated Commenting. Chapman & Hall, London, UK, 1992.

[Bott and Ratcliffe, 92] F. Bott, and M. Ratcliffe. Reuse and Design. Chapman & Hall, London, UK, 1992.

[Boyle and Muralidharan, 84] J.M. Boyle and M.N. Muralidharan. Program Reusability through Program Transformation. IEEE Transactions on Software Engineering, 1984.

[Brown and Chandrasekaran, 89] D.C. Brown and B. Chandrasekaran. Design Problem Solving: Knowledge Structures and Control Strategies. Morgan Kaufmann Publishers, 1989.

[Brown, 97] Kyle Brown. Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. URL: http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm, 1997.

[Budinsky et al., 96] F.J. Budinsky, M.A. Finnie, J.M. Vlissides, and P.S. Yu. Automatic Code Generation from Design Patterns. IBM Systems Journal, Vol. 35, No. 2, 1996.

[Buhr and Casselman, 96] R.J.A. Buhr and R.S. Casselman. Use Case Maps for Object-Oriented Systems. Prentice Hall, Upper Saddle River, NJ, 1996.

[Casais] Eduardo Casais. The Automatic Reorganization of Object Oriented Hierarchies.

[Chandrasekaran et al., 91] B. Chandrasekaran, J. Josephson, D. Moberg and H. Narayanan. Building Technical Knowledge Bases for Design and Diagnosis: Use of Functional and Visual Representations. IJCAI-91, Sydney, Australia, 1991.

[Chen and Ramamoorthy, 86] Yih-Farn Chen and C.V. Ramamoorthy. The C Information Abstractor. IEEE, 1986.

[Cheng and Gannod, 90] Betty H.C. Cheng and Gerald C. Gannod. Abstraction of Formal Specifications from Program Code. Proceedings of the 3rd Annual International Conference on Tools with AI, November 1990.

[Cheng et al., 94] Betty H.C. Cheng, Enoch Y. Wang and Robert H. Bourdeau. A Graphical Environment for Formally Developing Object-Oriented Software. Proceedings of IEEE on Tools with AI, November, 1994.

[Cheng and Jeng, 95] Betty H.C. Cheng and Jun-Jang Jeng. Reusing Analogous Components, June 1995.

[Cleeland et al., 96] Chris Cleeland, Douglas C. Schmidt and Tim Harrison. External Polymorphism, Proceedings of the Third Pattern Languages of Programming Conference, September 1996.

[Deutsch, 83] L.P. Deutsch. Reusability in the Smalltalk-80 Programming System. ITT Proceedings of the Workshop on Reusability in Programming, 1983.

[Dubinsky et al., 89] Ed Dubinsky, Stefan Freudenberger, Edith Schonberg and J.T. Schwartz. Reusability of Design for Large Software Systems: An Experiment with the SETL Optimizer. ACM Press, New York, 1989.

[Dyer et al., 86] M.G. Dyer, M. Flowers and J. Hodges. Edison: an Engineering Design Invention System Operating Naively. Applications of AI in Engineering Problems, Springer-Verlag, 1986.

[Faltings et al., 91] Boi Faltings, Gerhard Schmitt and Ian Smith. Case-Based Representation of Architectural Design Knowledge. IJCAI-91, Sydney, Australia, 1991.

[Feather, 83] M.S. Feather. Reuse in the Context of a Transformation Based Methodology. ITT Proceedings of the Workshop on Reusability in Programming, 1983.

[Fischer et al., 90] G. Fischer, A.C. Lemke, T. Mastaglio and A.I. Morch, "Using Critics to Empower Users", Empowering People: CHI'90 Conference Proceedings, April 1990.

[Fischer and Girgensohn, 90] G. Fischer and A. Girgensohn, "End-User Modifiability in Design Environments", Empowering People: CHI'90 Conference Proceedings, April 1990.

[Foster and Munro, 87] John R. Foster and Malcolm Munro. A Documentation Method Based on Cross-referencing. Proceedings Conference on Software Maintenance, 1987.

[Frazer, 92] A. Frazer. Reverse Engineering - Hype, Hope or Here? Chapman & Hall, London, UK, 1992.

[Freeman, 83] Peter Freeman. Reusable Software Engineering Concepts and Research Directions. ITTT Proceedings of the Workshop on Reusability in Programming, 1983.

[Freeman, 87] Peter Freeman. A Conceptual Analysis of the Draco Approach to Constructing Software Systems. IEEE Transactions on Software Engineering, 1987.

[Gamma et al., 95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. Addison-Wesley, Reading, Mass, 1995.

[Gerhart, 83] S.L. Gerhart. Reusability Lessons from Verification Technology. ITT Proceedings of the Workshop on Reusability in Programming, 1983.

[Gero, 91] J.S. Gero. Ten Problems for AI in Design. IJCAI-91, Sydney, Australia, 1991.

[Goel and Chandrasekaran, 89] A.K. Goel and B. Chandrasekaran. Functional Representation of Designs and Redesign Problem Solving. IJCAI-89, 1989.

[Goel and Prabhakar, 91] A.K. Goel and S. Prabhakar. A Control Architecture for Model-Based Redesign Problem Solving. IJCAI-91, Sydney, Australia, 1991.

[Goguen, 86] J.A. Goguen. Reusing and Interconnecting Software Components. Computer, February 1986.

[Goldstein, 91] D. Goldstein. Knowledge-Based Systems in a Flexible, Modular, Computer Integrated Manufacturing Environment. IJCAI-91, Sydney, Australia, 1991.

[Hall, 92] P.A.V. Hall. Software Reuse, Reverse Engineering, and Re-engineering. Chapman & Hall, London, UK, 1992.

[Harrison et al., 96] Tim Harrison, Douglas C. Schmidt and Irfan Pyarali. Asynchronous Completion Token. Proceedings of the Third Annual Pattern Languages of Programming Conference, September 1996.

[Harrison and Schmidt, 97] Tim Harrison and Douglas C. Schmidt. Thread-Specific Storage. Proceedings of the Second Annual European Pattern Languages of Programming Conference, July 1997.

[Haselbock and Stumptner, 91] Alois Haselbock and Markus Stumptner. Configuration and Design as a Constraint Satisfaction Task. IJCAI-91, 1991.

[Hodgson, 92] R. Hodgson. The Impact of Software Reuse on Object-Oriented Methods. Chapman & Hall, London, UK, 1992.

[Horowitz and Munson, 84] E. Horowitz and J.B. Munson. An Expansive View of Reusable Software. IEEE Transactions on Software Engineering, 1984.

[Hunt and Lee, 91] John Hunt and Mark Lee. Towards a Knowledge-Based Design Assistant. IJCAI-91, Sydney, Australia, 1991.

[Hutt, 94] Andrew T.F. Hutt. Object Analysis and Design. John Wiley & Sons, New York, 1994.

[Jain and Schmidt, 97] Prashant Jain and Douglas C. Schmidt. Dynamically Configuring Communication Services with the Service Configurator Pattern. C++ Report magazine, June 1997.

[Jandrasics, 81] G. Jandrasics. SOFTDOC - A System for Automated Software Analysis and Documentation. Proceedings of ACM Workshop on Software Quality Assurance, April 1981.

[Jeng, 94] Jun-Jang Jeng and Betty H.C. Cheng. A Formal Approach to Reusing More General Components. Proceedings of IEEE 9th Knowledge-Based Software Engineering Conf., September 1994.

[Jeng, 95] Jun-Jang Jeng and Betty H.C. Cheng. Specification Matching for Software Reuse: A Foundation. Proceedings of ACM Symposium on Software Reuse, Seattle, Washington, April 1995.

[Kernighan, 84] B.W. Kernighan. The Unix System and Software Reusability. IEEE Transactions on Software Engineering, 1984.

[Krueger, 92] Charles W. Krueger. Software Reuse. ACM Computing Surveys, Volume 24, Number 2. ACM Press, New York, June 1992.

[Langergan and Grasso, 84] R.G. Langergan and C.A. Grasso. Software Engineering with Reusable Designs and Code. IEEE Transactions on Software Engineering, 1984.

[McDermott, 82] J. McDermott. A Rule-Based Configurer of Computer Systems, AI 19(1), 1982.

[McGill, 92] R. McGill. Reverse Engineering - not yet? Chapman & Hall, London, UK, 1992.

[Maher and Zhang, 91] M.L. Maher and D.M. Zhang. CADSYN: Using Case and Decomposition Knowledge for Design Synthesis. AI in Design '91, Butterworth-Heinemann, 1991.

[Matwin et al., 91] S. Matwin, R. Holte and F. Oppacher. Application of Machine Learning to Software Re-use. IJCAI-91, Sydney, Australia, 1991.

[Menzies, 95] Tim Menzies. Problems with Patterns. July 1995.

[Munro, 92] M. Munro. Software Maintenance, Reuse and Reverse Engineering. Chapman & Hall, London, UK, 1992.

[Nadel and Lin, 91] Bernard A. Nadel and Jiang Lin. Automobile Transmission Design: a Constraint Satisfaction Formulation and Prolog Implementation. IJCAI-91, Sydney, Australia, 1991.

[Neighbors, 84] James M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. IEEE Transactions on Software Engineering, Volume SE-10, Number 5, 1984.

[Neighbors, 89] James M. Neighbors. Draco: A Method for Engineering Reusable Software Systems. ACM Press, New York, 1989.

[Parnas et al., 83] D.L. Parnas, P.C. Clements and D.M. Weiss. Enhancing Reusability with Information Hiding. ITT Proceedings of the Workshop on Reusability in Programming, 1983.

[Parnas and Clements, 86] David Lodge Parnas and Paul C. Clements. A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986.

[Pree, 95] Wolfgang Pree. Design Patterns for Object-Oriented Software Development. Addison-Wesley, Wokingham, U.K, 1995.

[Prieto-Diaz and Freeman, 87] R. Prieto-Diaz and Peter Freeman. A Software Classification Scheme for Reliability. IEEE Software, 1987.

[Pu and Reschberger, 91] P. Pu and M. Reschberger. Assembly Sequence Planning Using Case-Based Reasoning Techniques. AI in Design '91, Butterworth-Heinemann, 1991.

[Rich and Waters, 88] Charles Rich and Richard C. Waters. Programmer's Apprentice: A Research Overview. IEEE Computer, November 1988.

[Rowles et al., 91] C.D. Rowles, H. Liu and C. Leckie. Representation of Designs and Design Knowledge. IJCAI-91, Sydney, Australia, 1991.

[Rumbaugh et al., 91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Salton and McGill, 83] Gerard Salton and Michael J. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, New York, 1983.

[Schmidt, 95] Douglas Schmidt. Acceptor. C++ Report magazine, November/December 1995.

[Schmidt, 96] Douglas Schmidt. Acceptor. C++ Report magazine, January 1996.

[Schmidt, Dec 96] Douglas Schmidt, A Family of Design Patterns for Application-Level Gateways, Journal of Theory and Practice of Object Systems, Vol. 2, No. 1, Wiley & Sons, December 1996.

[Sevinc and Zeigler, 91] S. Sevinc, J. Rozenblit and B.P. Zeigler. Simulation Knowledge-Based Design. IJCAI-91, Sydney, Australia, 1991.

[Sneed and Jandrasics, 89] Harry M. Sneed and Gabor Jandrasics. Inverse Transformation from Code to Specification. Proceedings Software Tools '89, Blenheim Online, London, 1989.

[Sutcliffe, 92] A.G. Sutcliffe. Analogy in Software Reuse. Chapman & Hall, London, UK, 1992.

[Van Rijsbergen, 79] C.J. van Rijsbergen. Information Retrieval. Butterworths, London, UK, 1979.

[Vlissides, 95] John Vlissides. Reverse Architecture. Position Paper Dagstuhl Seminar, August 1995.

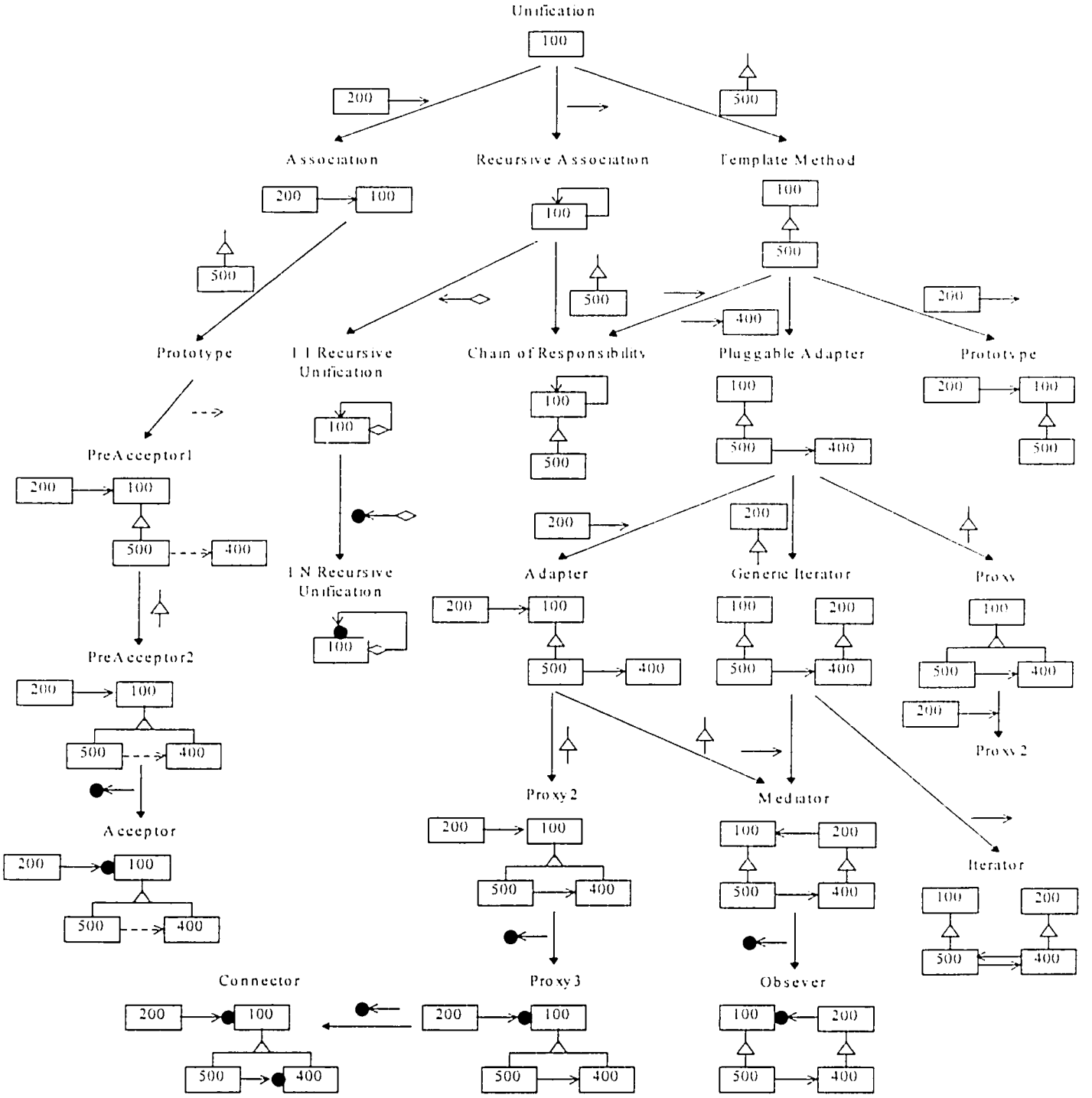
[Volpano and Kieburtz, 89] Dennis M. Volpano and Richard B. Kieburtz. The Templates Approach to Software Reuse. ACM Press, New York, 1989.

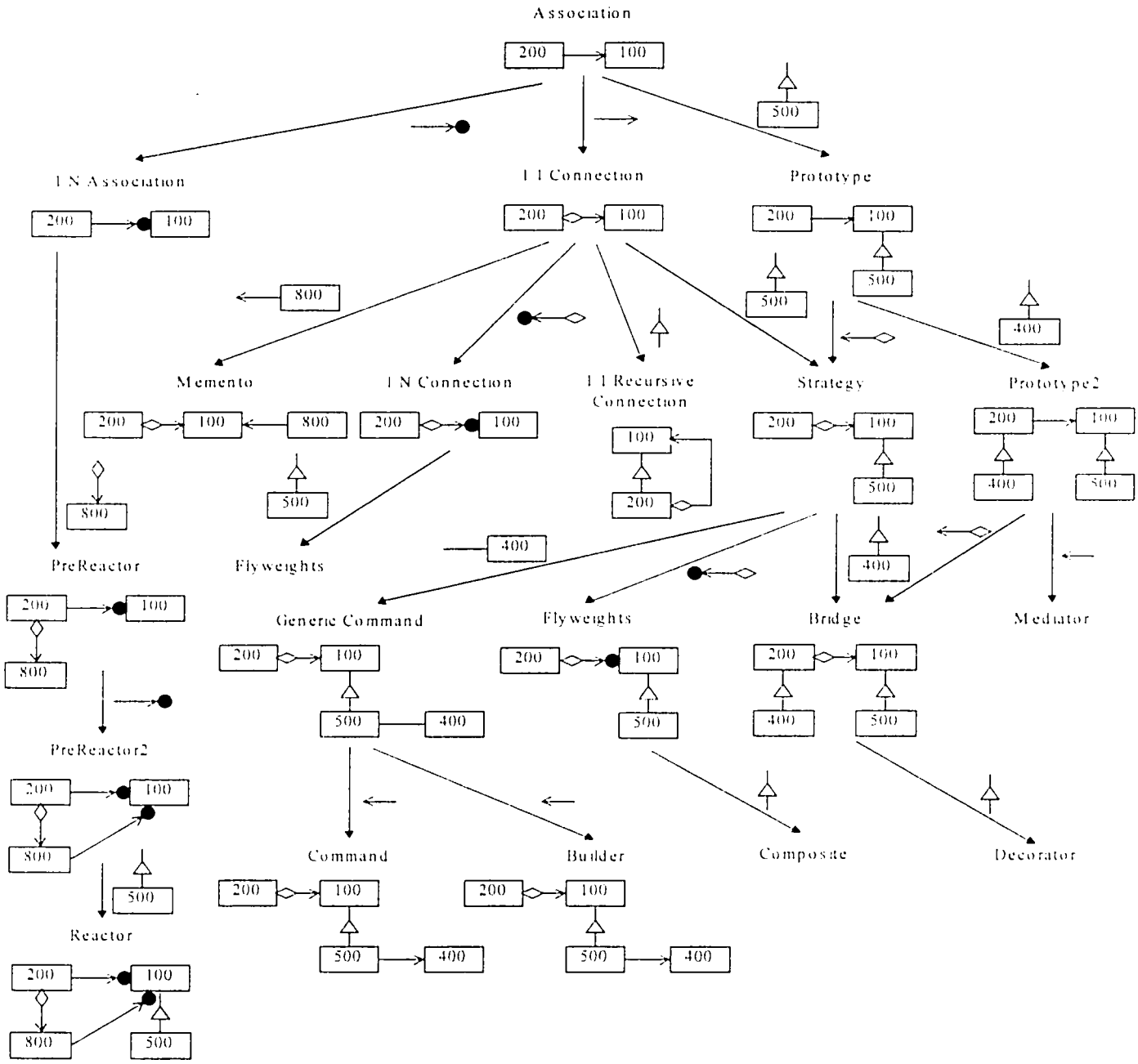
[Ward, 88] Martin Ward. Transforming a Program into a Specification. Computer Science Technical Report 88/1. School of Engineering and Applied Science, University of Durham, January 1988.

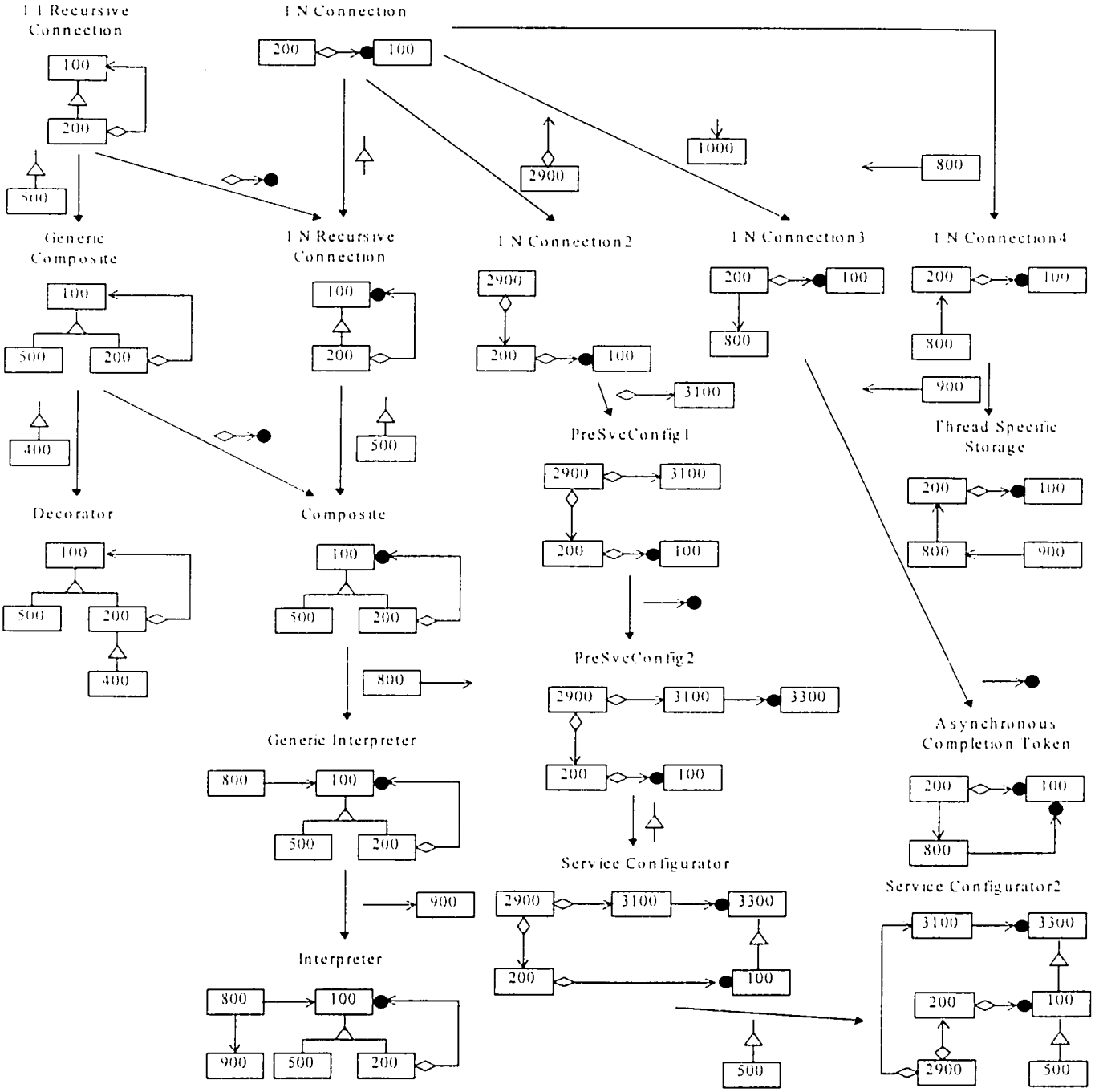
[Wegner, 83] P. Wegner. Varieties of Reusability. ITT Proceedings of the Workshop on Reusability in Programming, 1983.

[Zimmer] Walter Zimmer. Experiences Using Design Patterns to Reorganize an Object Oriented Application.

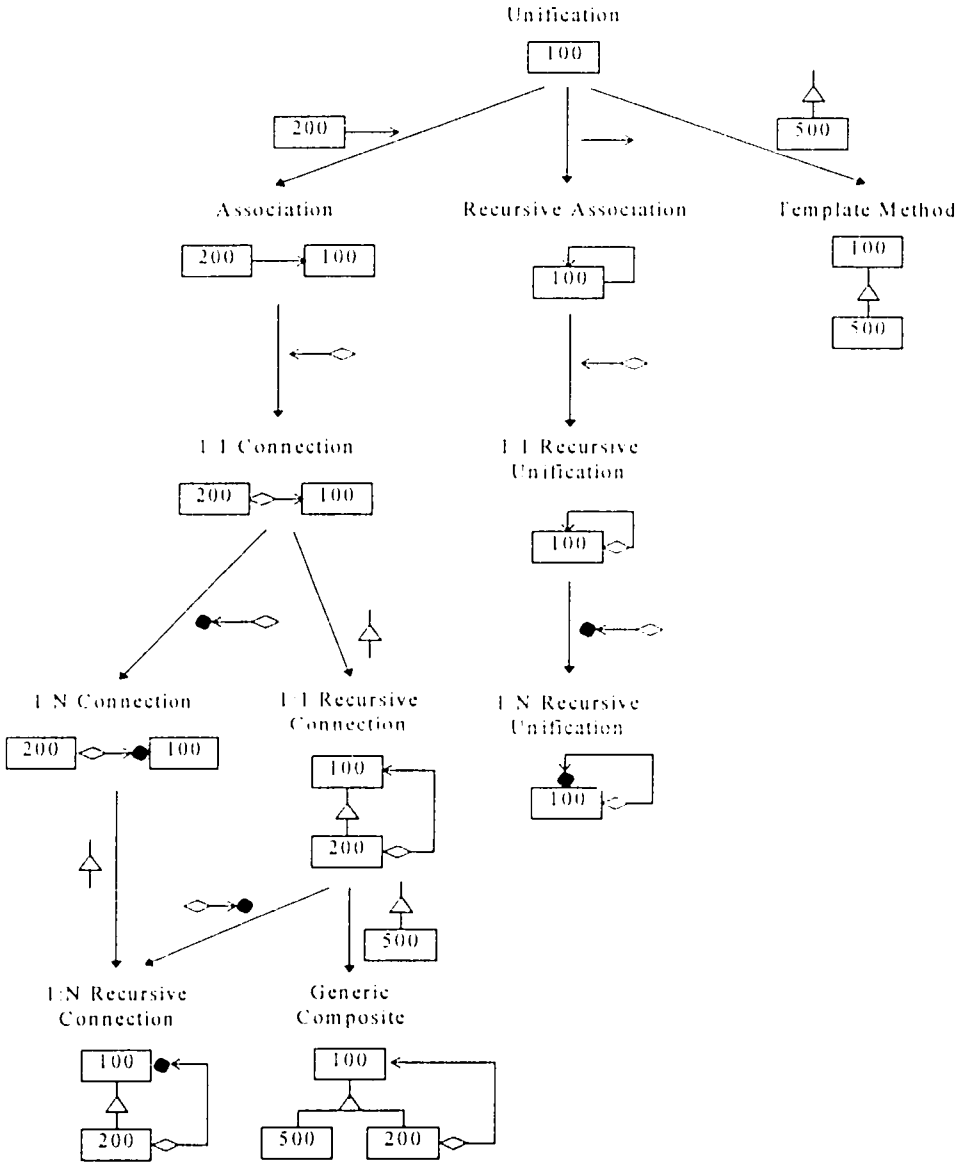
# Appendix A: Complete Knowledge Base of DEPARS



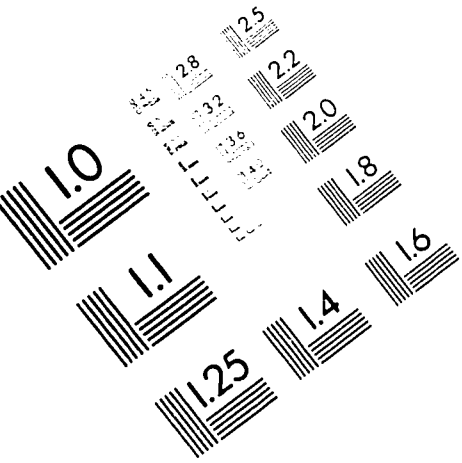

Association
200
100
500
1 N Association
1 1 Connection
Prototype
800
Memento
1 N Connection
1 1 Recursive Connection
Strategy
Prototype2
400
PreReactor
Flyweights
Generic Command
Flyweights
Bridge
Mediator
PreReactor2
Command
Builder
Composite
Decorator
Reactor

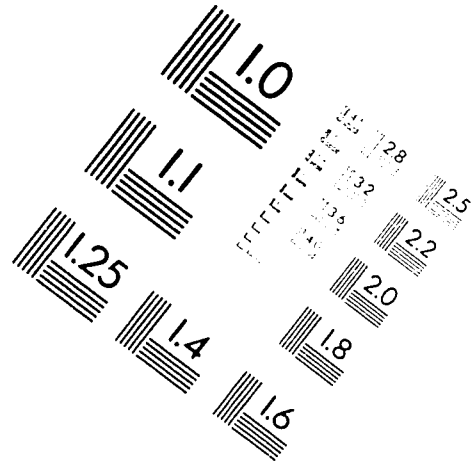

1 1 Recursive Connection
100
200
500
Generic Composite
100
500
200
400
Decorator
100
500
200
400
1 N Recursive Connection
100
200
500
Composite
100
500
200
800
Generic Interpreter
800
100
500
200
900
Interpreter
800
100
900
500
200
1 N Connection
200
100
2900
1000
800
500
1 N Connection2
2900
200
100
3100
PreSvcConfig1
2900
3100
200
100
PreSvcConfig2
2900
3100
3300
200
100
Service Configurator
2900
3100
3300
200
100
500
1 N Connection3
200
100
800
1 N Connection4
200
100
800
900
Thread Specific Storage
200
100
800
900
Asynchronous Completion Token
200
100
800
Service Configurator2
3100
3300
200
100
2900
500

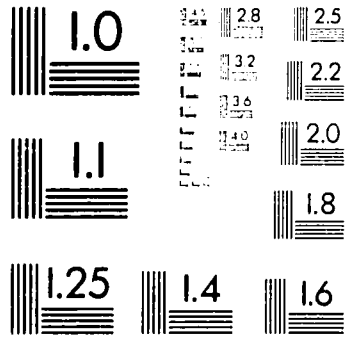
# Appendix B: Minimum Knowledge Base of DEPARS

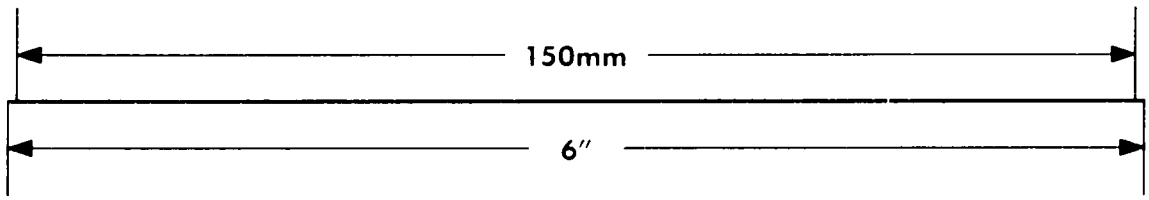


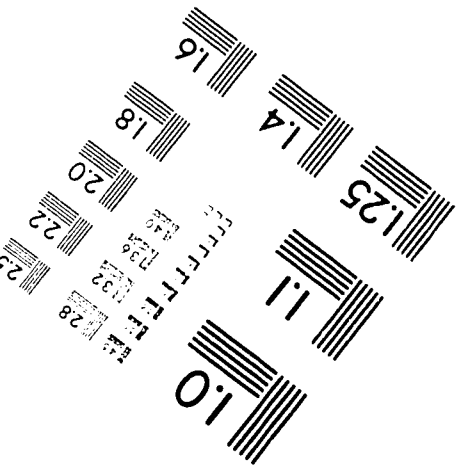
# IMAGE EVALUATION
# TEST TARGET (QA-3)

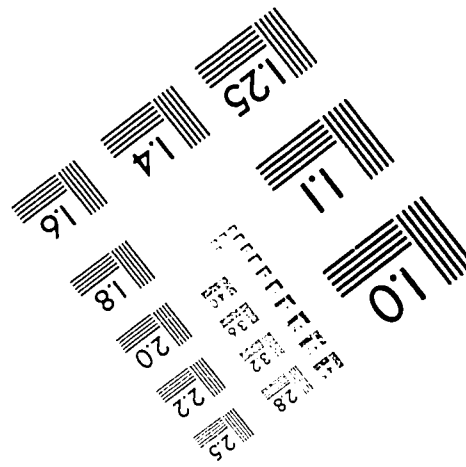



APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989