A Database Representation That Improves Automation and Maintains Consistency in A Multiple View Environment

Fawzi Albaloshi
Assistant Professor
Department of Computer Science
University of Bahrain
PO Box 32038, Isa Town, Kingdom Of Bahrain
Mobile: +973-9432994, Fax: +973-682582
fawzii@batelco.com.bh

ABSTRACT

The paper presents an approach to information representation that takes advantage of the incremental nature of software development to facilitate automation. Software is developed in stages where input to a stage is highly dependent on the stage that precedes it. A canonical database representation that is shared by the development tools is presented. It aims at the provision of maximum automated services in the development process. At the same time ensuring the consistency of design information and avoiding its redundancy, and yet providing an open environment in which new tools and services can be incorporated with no effect on existing ones. Experiments are carried out to examine the usefulness of the representation to store the artifacts of a system under development and its suitability in achieving automation in the development process. Similar work carried out by researchers in the field and a number of commercially available tools are analyzed.

Categories and Subject Descriptors

General Terms
Design, Languages, Documentation.

Keywords
Information Representation, Object-Orientation, Multiple Views, Automation, Software Development.

1. INTRODUCTION

A major requirement in a software development effort is the right set of tools to maintain the information related to the system under development, including analysis and design models, code, data dictionary, and test cases. There are many commercially available tools that provide the user with support, such as, modeling for the different development stages using a variety of notations, and code generation. Most have attractive interfaces, support a wide range of modeling and maintenance functionality for a number of notations, and provide the capability of deriving code from designs (forward engineering), and designs from code (reverse engineering) as is presented in the literature survey in section 2. The automated services offered are limited to that between design and coding, though key researchers in the field such as Booch [4] and Pressman [24] emphasis the importance of the analysis and design stages and their impact on a successful implementation, and the role of testing in the reduction of errors.

Environments that support multiple views of software development provide the user with different views of the system under development, thus linking the various stages in the development process. Such environments have been developed by Leung and Long [20]; Grundy and Hosking [12]; Gorth, Hermann, Jahmichen, and Koch [15]; and Rosch [27]. Meyers and Reiss [23] found that when increasing the number of views of a software system the performance of the programmers working on the system was increased. The bases of such an environment are three levels of integration, and they are, data, presentation, and control integration [3]. Data integration is the basis upon which the other two levels of integration can be built. Meyers [22] presents and compares the various approaches to integration and they are shared file systems, selective broadcasting, simple databases, view-oriented databases, canonical representation, and hybrid approaches.

Automation in software development is given little attention in such environments regardless of the fact that various development stages are dependent on each other and a certain level of automation can be achieved rather than presenting the same information in different form. For example, during the analysis stage, details related to a problem are specified in brief, and are extended during the design stage. It is natural that the design specification would also include details specified during the
analysis, thus part of the development process includes re-
specification. Similarly, the process of coding a design is mainly
mapping or translating the design to code in a suitable
programming language. If details of the system are properly
captured at the analysis stage, part of the design can be
automatically produced, and, if details of the design are stored in
a suitable form, part of the code can be automatically generated.
Therefore, we strongly believe that by capturing the system
artifacts and allowing the tools to share it more automated
services can be provided than those currently offered by existing
CASE tools.

The approach we follow in information representation as
presented in this paper provides the base for an environment in
which the user can swiftly move from an early development stage
to the next with maximum automation possible based on design
information so far provided. The work presented here is based on
and improves upon earlier work carried out by researchers in the
field as reported in section 2. Improvements include increased
automation, maintaining consistency, avoiding redundancy; and
achieving openness. Consistency of design information is an
important issue that we address in various sections in the paper.
Transition between the stages in object-oriented (OO) software
development using the unified modeling language (UML) is
explored upon which automation is based. Information
representation for the various UML models is presented.
Experiments are carried out that demonstrate the usefulness of the
suggested representation as a base of a multiview environment
that maintains consistency and provides automated transition from
a stage to another.

2. RELATED WORK

Meyers [22] discusses the difficulties associated with integrating
multiple view environments, and highlighted five important
considerations that could be used to determine the extent with
which an environment is integrated. Interesting work on a
multiple view environment supporting graphical and textual tools
is being carried out at the university of Wales as reported in [21,
30, 20]. Grundy and Hosking [10] present the Snart Programming
Environment (SPE) that supports multiple textual and graphical
views of a program. It provides support for object-oriented
analysis and design using diagrams, textual programming, and
hypertext-based browsing. An approach to manage inconsistencies in multiple views of software development is
presented in [14] and applied on SPE. The authors recognize
the importance of sharing the development information between the
views through a common repository in improving consistency.
They present an approach to inconsistency management in a
multiple view environment. Project integrating reference object
library (PIROL) is a software engineering environment developed
at Technische Universitat Berlin and GMD-First Berlin [15]. It
uses the structural decomposition approach in which data is
derived from data elements in the repository. Documents are
stored as conceptual objects that enable tools to rely on the same
data despite being generated at different stages in the
development cycle. Rosch [27] proposes the graph model that
provides the basis for a flexible internal representation of design
artifacts (abstract objects) in multiple view environments. In it the
system under development has a single representation as a graph
structure.

A combined approach to information representation is used by
Long, Wang, and Leung [21] and according to Meyers [22] it can
be classified as the database with views and the canonical
representation. This approach has the advantage to be good in
presenting simultaneous views, maintains consistency, and avoids
redundancy. The ability to incorporate new tools to the
environment is fair. Other works such as that proposed by
Albalooshi and Long [1] follow a similar approach. Grundy and
Hosking [9] use the database with views approach that has similar
strengths of the canonical representation. It is good in presenting
simultaneous views, consistency management, and redundancy
avoidance, but suffers from difficulties when adding/writing new
tools to the environment. Rosch’s [27] proposal uses a similar
approach since information is stored in a common database, but
different views can access certain parts. The approach followed in
PIROL [15] can be classified as the simple database to
information representation. It is considered to be poor in
writing/adding new tools and in presenting simultaneous views,
but good in consistency management and fair in redundancy
avoidance.

A number of commercially available and widely used
environments have been reviewed to obtain a clear view of the
state of art in CASE tools practice. Such as Software through
Pictures [2], Rational ROSE [26, 25], and Vision Enterprise [29].
The first supports automatic code generation and test case
derivation from requirements. The later two support forward and
reverse engineering between design and coding in addition to
semantic and syntax checking the models during an editing
session. A study on CASE tools carried out by Grundy, Hosking,
and Mugridge [14] showed dissatisfaction with the approach
followed in maintaining consistency. They found that CASE
environment use the notion of a repository with a database view
mechanism to keep multiple views consistent. Software thru
Pictures uses Sybase, and according to Meyers [22] this approach
is good to maintain consistency but poor in other criteria such as
writing new tools, adding new tools, and presenting simultaneous
views. Case tools that use files such as Rational Rose and Visio
Enterprise can be good for incorporating new tools and adding
tools but poor in presenting simultaneous views, maintaining
consistency, and avoiding redundancy.

3. TRANSITION BETWEEN THE
DEVELOPMENT STAGES

Several researchers, including, Booch, Rumbaugh, and Jacobson
[5], Quatrani [25], Eriksson and Penker [7], and Fowler and Scott
[8] have addressed OO software development with UML. The
unified software development process [17] is a use case driven
approach to software development that starts with the
specification of the use case diagrams for the system to be
modeled. The developer then moves on for more detailed system
investigation and the production of interaction diagrams. An
interaction diagram describes the behavior of a use case diagram,
thus the actor(s) involved must be included and can be generated
automatically based on the information specified in the use case
diagram. The situation is visible in figures 1 and 3 in which the
Borrower is part of both diagrams. There are two types of
interaction diagrams and they are sequence and collaboration.
Both diagrams contain the same objects and function calls except
that they are ordered differently. Therefore one can automatically
be driven from the other.
At this stage more details are specified and the development process gradually shifts to design and the production of classes and their relationships. The main components of a class diagram are the classes, their attributes, operations, and the relationships. They can automatically be driven from the interaction diagrams. The objects must be represented by classes, the function calls must be part of the used classes, and the call parameters can be generated as class attributes. This situation is shown in figures 3 and 5 in which classes are driven from the sequence diagram. Further refinement would obviously need to be done by the user manually. Interactions between objects are the relationships between the classes that can be generated automatically and refined. Automation can include the generation of implementation code for the classes and their details. Figure 6 and program 1 presents a situation in which code is generated based on a representation for classes. Activity diagrams that are used to model the methods/operations can automatically be analyzed to generate compile-able code.

Recent research in automated testing such as that reported by Howe, Mayrhauser, and Mraz [16] and Devanbu, Rosenblum, and Wolf [6] highlight its importance and the need of specialized tools to carry it out. The approach followed to information representation in the work presented here renders automated test case generation a simpler task. Two common testing approaches are black box testing and white box testing [24]. A very useful black box testing technique for OO systems is scenario-based testing. In it the sequence of events that carry out a specific user need is tested. Such events can automatically be generated based on the interaction diagrams. A common white box testing technique is basis path testing. In this, a flow graph is drawn for the operation or procedure to be tested and a basis set of linearly independent paths is determined. Flow graphs are similar to activity diagrams, and the later can be used instead to derive test cases.

4. DATABASE REPRESENTATION

An important aspect in any development environment that has a major impact on the type of services that can be offered is information representation. The study presented in section 2 emphasizes this fact and shows that the graph structure is used in a number of environments to represent the system details on an object store; a similar approach is followed for the environment presented here. A graph structure is carefully drawn to represent the information from different development stages that would aid automation in the development process. Jasmine, an OO database that can be accessed from a number of commonly used development languages including C++ and Java, is used as the store for the development information. In addition to its open interface it has its own development environment through which it can be manipulated and examined [18]. This section explores in detail how each of the main UML modeling elements is mapped to the database.

4.1 Use Case Diagrams

To store details of a use case diagram three types of classes are used. The first to represent use case diagrams called ‘UCD’, an instance for each diagram; the second to represent actors called ‘Actor’, an instance for an actor; and the third to represent use cases called ‘UseCase’, an instance for each. The class definitions are shown in appendix A.1.

4.2 Interaction Diagrams

To store details of interactions three main classes are needed. The first is a class titled Interaction as shown in appendix A.2 that represents a particular interaction. An interaction is a scenario detailing a specific use case; therefore an interaction object is linked from a use case object as defined in section 4.1. A particular interaction has a title and contains a list of objects that interact with each other through operation calls. An instance of the class Entity as shown in appendix A.2 is used to represent an object participating in an interaction. Typically an object has a name, a class type, and links to other objects through operation calls. Objects interact with each other through operation calls, and it is important to capture details of an operation. Therefore a class definition called ‘Operation’ is specified as shown in appendix A.2. To further decompose details of an operation a class called ‘Attribute’ is also defined as shown in the same appendix to represent an operation parameter. To represent a link/operation call between two objects of type Entity a special class called Lenk is defined as shown in appendix A.2.

4.3 Class Diagrams

As discussed earlier, classes consist of attributes and operations, and in some cases a special semantical description may be given to a class definition through a stereotype specification. In a class diagram it is necessary to define relationships between individual classes. To represent a class a Jasmine class definition called ‘Atype’ is suggested to hold the necessary details as shown in appendix A.3.

4.4 Relationships

To represent a ‘dependency’, a class definition that holds the object ids of the class objects (objects of type AType) sharing the relationship is needed; other details to be stored include the name and/or stereo type if any, and the relationship type. A class definition called ‘Relation’ is specified as shown in Appendix A.3. A ‘generalization’ relationship can be named, stereotyped and constrained. A new class definition to represent the generalization relationship called ‘Relation2’ is specified as shown in Appendix A.3. In most cases navigation between the ends of an ‘association’ is bi-directional, but in some cases it may be useful to restrict navigation to one direction. It may also be necessary to restrict visibility of an ‘association’. Another useful mechanism available in UML is to apply a constraint to the relationship. All the details that need to be maintained for an association require the definition of a special class called ‘AssoDet’ as shown in appendix A.3. Using this class a new definition to hold an association relationship is specified called ‘Relation3’ as shown in the same appendix. The new class inherits all details defined for the ‘dependency’ and ‘generalization’ relationships and adds more that are specific to the association relationship.

4.5 Activity Diagrams

An activity has an initial state that signifies its start, and an end state to indicate its termination. Other components of an activity
are selection, repetition, sequence, and concurrency. To represent an activity diagram on Jasmine a set of classes are defined, and they are ‘Initial’, ‘Activity’, ‘Participant’, ‘Repeat_Branch’, and ‘Final’ as shown in appendix A.4.

5. TOOLS AND EXPERIMENTS
Four prototypical tools have been developed to maintain the diagrams: use case, sequence, collaboration, and class. They provide the user with an easy to use interface and functionality to maintain the diagrams. To come up with the representation presented in section 4, that cases a nd aids automated transition from one development stage to another, a number of experiments are carried out. A set of Java programs is written to populate the database with an example system and process the data at each stage to generate the design information that is useful for the stage that follows. These are a series of ‘populate’ and ‘process’ programs that access the object base using the interface library of functions. The ‘populate’ programs enrich the database with test data (simulating an edit session), and the ‘process’ programs automatically generate new data for the development stage that follow (simulating a forward engineering operation). In the sub sections that follow we present the experiments to test the suitability of our approach in information representation and automatic forward engineering support between the main development stages.

5.1 Creating and Processing a Use Case Diagram
At first the database is populated with an example use case diagram consisting of three use cases and an actor, part of which is shown in figure 1. Three objects of type UseCase are generated one for each use case. Three objects of type Relation are also generated representing the relationships: ‘uses’, ‘include’ and ‘extend’. Finally, an object of type Actor represents the actor. The objects representing the relationships are linked to the use case each originates from. The ‘uses’ type is added to the actor’s list of relations, the ‘include’ and ‘extend’ to the use cases’ list of relations. Figure 2 shows some of the objects making up the representation. Doing so eases navigation between the objects and diagram presentation by the tool. The use case diagram is then processed using a special program to generate design information for the stage that follow. An object representing an interaction diagram for each use case is created, and an entity object representing the Actor is created and added to the list of entities belonging to the interaction diagram it is part of. In other words, three objects of type interaction are created for the use case diagram, one for each use case, and one entity object representing the actor is made to be part of the interaction object representing the use case it uses.

![Figure 1: Use Case Diagram for A Loan Operation.](image)

![Figure 2: The Database Representation for the Use Case Diagram in Figure 1.](image)

5.2 Populating and Processing the Interaction Diagram
Using a populate program the Loan interaction diagram is then elaborated and a set of objects of type Entity is created to represent the objects in the diagram, and their details are set. To represent the interactions (operation calls) between the objects in the diagram, objects of type Operation are created and their details such as name, return type, and the type of objects it operates on are set. Parameters to operations are represented as database objects of type Attribute. They are then linked to the operations they belong to as parameters. The types of the objects (classes) in the diagram are represented as database objects of type AType, and the objects representing the operations are linked to the classes they belong to, so that they become accessible from the classes. A set of database objects of type Lenk are created for each operation call and their details such as serial number, operation name, and the entity operated upon are set. Each is then linked to the object representing the entity that used it. By doing so the interaction diagram is completely mapped to the database. A collaboration diagram can be automatically

![Figure 3: Partial Sequence Diagram for A Loan Scenario.](image)
generated using the same representation. Figure 3 shows part of the sequence diagram, and figure 4 shows some of the database objects representing it.

Moving on to class diagrams, some details can automatically be driven, including the classes along with their available operations and relationships. The program to generate class details reads the objects of type Entity first and creates an object of type Relation3 for each object of type Lenk originating from an entity object. These new objects represent association relationships between the classes, and are linked to the classes (objects of type AType) they originate from. At this stage the details available in the database are enough to generate the class diagram shown in figure 5. Figure 6 shows a partial representation for the LoanWindow class shown in figure 5.

5.3 Populating and Processing the Class Diagram

The class diagram is enriched with more details such as operations, attributes, and relationships. To add operations to classes, objects of type Operation are created and their details set. To add attributes, objects of type Attribute are created, they are then linked to the class (objects of type AType) they belong to. Details of an association relationship can be extended with cardinality, role, visibility, and direction. Other types of relationships can also be established between classes, such as, ‘inheritance’, ‘aggregation’, and ‘dependency’.

Skeleton code can be generated at this stage. The code generation program reads the AType object for a class. It can decide on the relationships the class has and generate appropriate include statements for the class header. It can then generate the class using its name, and partition its attributes according to their access specifications. All data and function members are grouped and stored temporarily according to their visibility specification, and generated along with their details at the end. A data member has a type and possible initialization details, and an operation has a name, a return type and possible parameters. Program 1 shows skeleton code for the LoanWindow class that is generated from the representation shown in figure 6. More automated details can be generated if necessary, such as set and get functions for the attributes and relations.

5.4 Generating Test Cases

To generate a scenario-based test case the program reads all entities belonging to an interaction (object of type Interaction). For each Entity object the links (objects of type Lenk) with other entities are listed. Each such object holds the sequence number for the operation call and the object ID of the object of type operation representing the operation. For each Lenk object the operation signature and class are retrieved using its ID, and are stored along with the sequence number and the type of the entity object the link originated from (Lenk object belongs to). This operation is repeated for all entities belonging to an interaction. The details are then sorted by sequence number and printed as a test case.
6. CONCLUSION

The time and effort spent in learning and using existing CASE tools may render their use impractical, since they do not reward the user with a major reduction in development time and effort due to the lack of automated services they offer. Software is mostly developed incrementally. By reusing part of the development information from an early development stage to complete a later stage, consistency is ensured, redundancy is avoided, and the development time and effort are minimized. In this paper a database representation for the main UML diagrams is presented. It serves as a basis for an integrated multiple view environment supporting OO software development. It enables the provision of simultaneous views of design information that share the same design information thus ensuring consistency and avoiding redundancy. The database is open and can be integrated with many well known programming languages, databases, and systems. Therefore new tools can easily be added to the environment to share the design information, and at the same time no restrictions are imposed on the way the tools should be developed other than the need to adhere to the common data representation. An approach to maximize automation in the development process is presented. Thus reducing the development time, and effort and providing a consistent transition mechanism from a development stage to another without the need of translators. Automation includes other system development activities, such as the generation of test cases, and implementation code. Experiments are carried to test the proposed representation and automation mechanism. A set of prototypical tools is developed using Java and integrated with the database. By centralizing the details of the system under development other project related services could be achieved, such as matrices calculation, generating reusable components, software reuse, and deriving project related statistics. The paper presents improvements in computer aided software development to that suggested by researchers and commercially available CASE tools reported in section 2, and overcomes some the limitations experienced. Improvements include increased automation, maintaining consistency, avoiding redundancy, and achieving openness.

7. ACKNOWLEDGMENTS

I thank all those who contributed to bring this paper to its presented form.

8. REFERENCES


**APPENDIX A. THE JASMINE CLASS DEFINITIONS FOR THE REPRESENTATION**

**A.1 Use Case Diagram**

```java
public class UCD { // Use Case Diagram
    protected String title;
    protected CollectionOfString usecases;
    protected CollectionOfString actors;
}
```

**A.2 Interaction Diagram**

```java
public class Interaction { // Interaction Diagram
    protected String title;
    protected CollectionOfObject entities;
}
```

**Appendix A.1 Use Case Diagram**

```java
public class UCD { // Use Case Diagram
    protected String title;
    protected CollectionOfString usecases;
    protected CollectionOfString actors;
}
```

**Appendix A.2 Interaction Diagram**

```java
public class Interaction { // Interaction Diagram
    protected String title;
    protected CollectionOfObject entities;
}
```
protected String title;
protected String belongsto;
protected int visibility;
protected String retturntype;
protected CollectionOfObject parameters;

public class Attribute { // Attribute
    protected String title;
    protected String type;
    protected int visibility;
    protected String initialization;
}

public class Lenk { // Lenk
    Double seqNo;
    String operationOID;
    String toOID;
}

A.3 Class Diagram

public class AType { // Class
    protected String title;
    protected String stereoType;
    protected CollectionOfObject attributes;
    protected CollectionOfString operations;
    protected CollectionOfObject relations;
}

public class Relation { // Dependency Relationship
    protected String title;
    protected String from;
    protected String to;
    protected String type;
    protected String stereotype;
}

public class Relation2 extends Relation { // Generalisation Relationship
    protected AssoDet first;
    protected AssoDet second;
    protected int direction;
}

A.4 Activity Diagram

public class Initial { // Initial state
    protected CollectionOfObject next;
}

public class Activity { // Activity
    protected String swimlane;
    protected String description;
    protected CollectionOfObject participants;
    protected CollectionOfObject next;
    protected String detail;
}

public class Participant { // Participant
    Protected String name;
    Protected String classtype;
    Protected String state;
}

public class Repeat_Branch { // Branching and Repetition
    Protected string false;
    Protected string true;
    Protected string condition;
}

public class Final { // End State
    Protected String name;
    Public Final() { name = “Final”;};
}

public class AsoDet { // Association Relationship Details
    protected String mult;
    protected String role;
    protected int visibility;
    protected String constraint;
}

public class Relation3 extends Relation2 { // Association Relationship
    protected AssoDet first;
    protected AssoDet second;
    protected int direction;
}