Computer-Aided Software Artifacts Generation at Different Stages within the Software Development Process

Fawzi Albalooshi

Assistant Professor

Department of Computer Science

College of Information Technology

University of Bahrain

PO Box 32038, Isa Town, Kingdom of Bahrain

fawzi@itc.uob.bh

fawzii@batelco.com.bh

Abstract: The process of developing software has undergone major improvements over the years and continues to do so. Many attempts have been made to automate or partially automate the development process to reduce the time, effort, and cost associated with it. This paper presents an approach to maintain and partially automate the development process. Automatic transitions from one development stage to the next with the generation of development information for the new stage is achieved as a result and, at the same time, consistency of information is ensured since every software artifact is stored as a single instance. Software artifacts of the system under development are stored in a common repository that can be shared by the development tools for the different development stages as well as by other case tools that are enabled access to the repository. The results of the experiments suggest that partial automation between each of the development stages can be impeded within the unified development process and major advantages can be obtained.

Keywords: Rational Software Development Process, Forward Engineering, Automation, UML, CASE Tools.

Dr Fawzi Albalooshi is an Assistant Professor at the Department of Computer Science at the University of Bahrain since 1996. Completed his graduate studies at the University of Wales were he was awarded his Masters in Computer Science and Ph. D. in the field of Software Engineering. He developed research interest in the area of Object Orientation since his research for his Ph. D. and continued to work in this area up to now. He joined the department of Computer Science as one its faculty members and continued to research and teach in Software Engineering. Other areas of research he is interested in include the use of computers in teaching/learning and educational technologies.

1. Introduction

The set of tools used in a software development effort has a major impact on the development time and the quality of the product. There are many commercially available tools that provide the user with modeling support for the different development stages using a variety of notations, code generation, and so on. Some of the tools that we reviewed include Software through Pictures [1], Rational Rose [21, 20], and Visio Enterprise [24]. The first tool supports automatic code generation and test case derivation from requirements. The other 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 (Computer-Aided Software Engineering) tools carried out by Grundy, Hosking, and Mugridge [11] showed dissatisfaction with the approach followed in maintaining consistency of design information. They found that CASE environments use the notion of a repository with a database view mechanism to keep multiple views consistent. Software through Pictures [1] uses Sybase, and according to Meyers [16], this approach is good to maintain consistency but poor in writing and adding new tools, and presenting simultaneous views. CASE tools that use files such as Rational Rose and Visio Enterprise can be good for incorporating and/or adding new tools, but poor in presenting simultaneous views, maintaining consistency, and avoiding redundancy. The automated services they offer are limited to design and coding, though key researchers in the field such as Booch [3] and Pressman [19] 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.

The unified development process [13] is a use-case driven, architecture-centric, iterative and incremental process. It repeats over a series of cycles making-up the life of a system. Each cycle concludes with a product release and is made-up of four phases: inception, elaboration, construction, and transition. Each phase is usually subdivided into iterations each of which goes through all the five work flows as shown in figure 1. The curves represent an estimate of the required work and to which workflow it belongs to.

Major improvements such as speeding-up the development process, maintaining consistency of system information, avoiding redundancies, and minimizing the development effort can be achieved when supported by automated services.

Automation in software development is given little attention disregarding the fact that the various development stages are dependent on each other and a certain level of automation between them can be achieved. 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 includes 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.

The paper presents an approach to software development that enables automatic transitions from one development stage to the next using a common repository of system information that is shared by the development tools. The repository used is Jasmine, an OO database, which can be accessed from a number of commonly used development languages including C++ and Java. In addition to its open interface, it has its own development environment through which it can be manipulated and examined [14].

The rest of the paper is organized as follows: in section two, a technique for transition between the development stages is proposed. Section three presents the graph structure used for system information representation suitable for our transition technique. Section four addresses the consistency problem associated with software development and explains how it is managed within the environment report here. Section five presents the experiments carried out to automate the transition technique proposed in section two and that is to enable automatic transitions from one development stage to the next, followed by concluding remarks in section six.

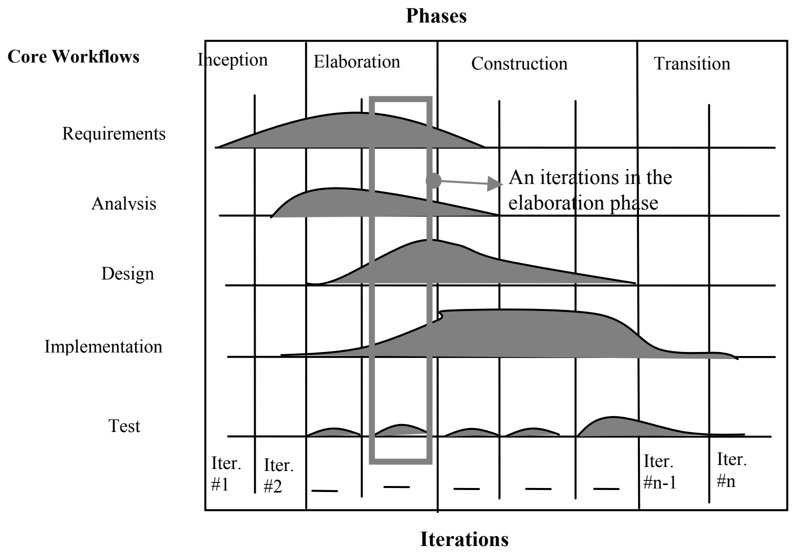


Figure 1: The Unified Development Process [10].

2. Proposed Technique for Transition between the Development Work Flows

Many authors, such as, Booch et al [4], Quatrani [20], Eriksson and Penker [7], and Fowler and Scott [8] have addressed OO software development with UML. Each of these authors follows a different approach in explaining how software could be developed object oriented-ly, using UML as the notational language. The authors clarify the details involved in every stage and, in some cases, give suggestions that aid the developer to move swiftly from one development stage to another. A number of researchers believe that automatic transformations between UML diagrams is possible two of recent such publications are [22 and 26]. Selonen and Koskimies [22] state that "Transformations are closely related to consistency checking: in essence, a transformation defines the information that must be shared by consistent diagrams". In this section, the main development stages are briefly reviewed with a discussion of the type of information that can be extracted from the details specified for that stage.

2.1. Use Cases

Use cases show how various actors interact with the system. A use case presents the functionality expected from the system but does not show how this functionality is implemented. Actors are the users of use cases; they can be human or non-human (another system or hardware). There are three relationships that are possible between use cases. They are ‘generalization’, ‘include’, and ‘extend’. Generalization indicates that a use case inherits the behavior and characteristics of another use case. This type of relationship is also possible between actors. The ‘include’ relationship is used to avoid duplicating the same behavior in more than one use case. Common behavior is represented by a single use case and then linked to any other use case that may need that behavior at some time during its processing. The ‘extend’ relationship between two use cases represents a situation in which a base use case may extend its functionality by using another use case. The base use case can stand alone, but may need the functionality of the other use case in some situations. Figure 2 shows an example use case diagram in which a borrower interacts with a Loan use case that uses the functionality of a Validate User use case.

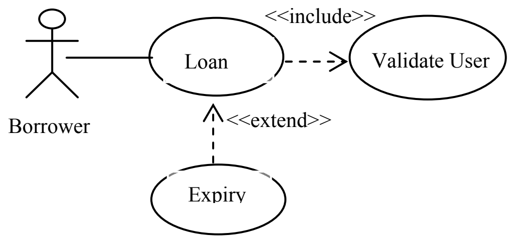


Figure 2: Use Case Diagram for A Loan Operation.

```
Input: use case diagram
Output: sequence diagram
Process: for each use case in the use case
                diagram
        create a sequence diagram representing
            a scenario

        for each actor interacting with the use
            case create an actor interacting
            with the sequence diagram
        end {for}

      end {for}

end {process}
```

Figure 3: Specifying The Sequence Diagrams.

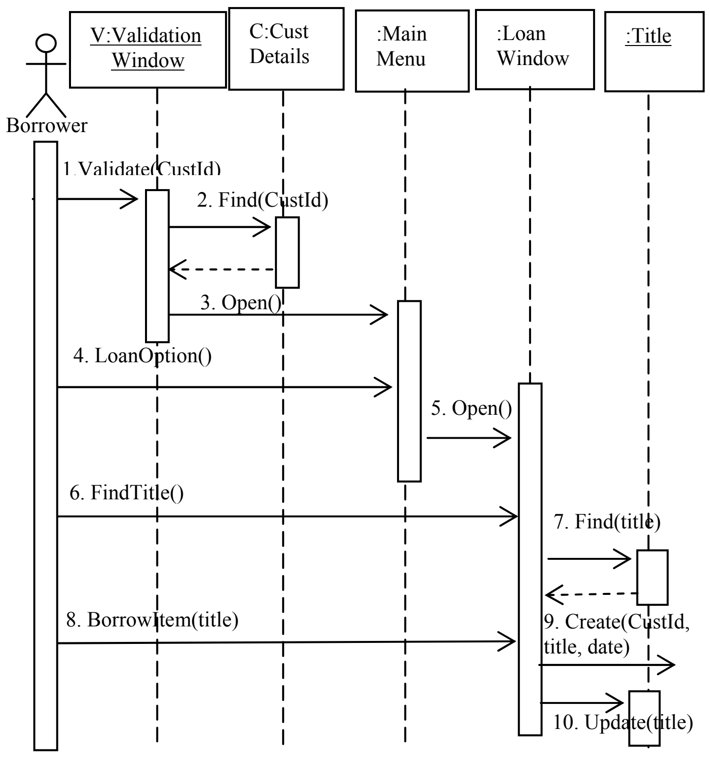


Figure 4: Partial Sequence Diagram for the Loan Scenario in figure 2.

The development process normally starts with the specification of use cases that are then elaborated upon with the specification of scenarios for each, using sequence and communication diagrams. Figure 4 shows a possible scenario for the Loan use case shown in figure 2. The borrower is the actor interacting with the use case, therefore it must be part of the scenario. Figure 3 presents the algorithm of transition from the use case stage to the specification of scenarios stage.

2.2. Interaction Diagrams

At early stages of development, especially in the analysis stage, the system is partitioned using use cases. However, to model the dynamic aspects of the system and show more details, interaction diagrams are used. There are two types of interaction diagrams (sequence and communication) which are used to model possible scenarios the use cases perform in terms of objects that interact and communicate with each other to carry out the necessary functionality. In a sequence diagram, interactions between the objects are ordered in time sequence, and in a communication diagram, the relationships between the objects are shown in terms of messages that are sent and received.

Figure 4 shows the sequence diagram detailing a possible scenario for the Loan use case shown in figure 2. The Borrower enters his identification number to be validated and, if found valid, the main menu is displayed. The loan operation can then be selected from a list of possible options that are provided by the menu. The lending window is then displayed and the title to be borrowed is searched for. If the title is available, a borrow command is issued and a loan transaction is created.

```
Input: sequence diagram
Output: communication diagram
Process: for each sequence diagram

          for each actor in the sequence
               diagram
            create another in the
               communication diagram
          end {for}

          for each object in the sequence
               diagram
            create an object in the
               communication diagram
                 with the same name and type
          end {for}

         for each message between two objects
              in the sequence diagram

              create a link/message between
              the same two objects in the
               communication diagram

         end {for}

       end {for}
end {process}
```

Figure 5: Specifying the Communication Diagrams.

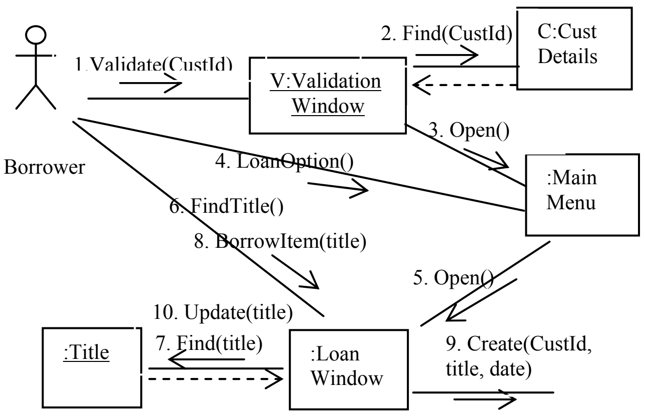


Figure 6: Communication Diagram Extracted from the Sequence in figure 4.

The sequence diagram shown in figure 4 can be presented as a communication diagram since both diagrams contain the same objects and function calls except that they are ordered differently. Therefore, an intelligent tool can generate one from the other. Generating a communication diagram from a sequence diagram can automatically be done. Figure 5 presents the transition algorithm from sequence diagrams to communication diagrams and figure 6 shows the communication diagram extracted from the sequence diagram shown in figure 4. Similarly a sequence diagram can automatically be generated from a communication diagram except that the sequence of operation calls would be missing. The diagrams contain many details that can be used to extract a class diagram for the system, for example, each object is of a class type that can be specified independently. Operation calls between objects suggest communication between them and can, therefore, be seen as relationships. For example, two classes could be specified for the objects of type ValidationWindow and CustDetails shown in figure 4. The first executes the find operation belonging to the other, thus an association relationship between the two classes can be established, and a find() operation prototype for the CustDetails class can be specified.

2.3 Classes

Classes are the building blocks of an OO system. A class is a specification representing a set of objects that share common attributes, operations, and relationships. Deciding on the right set of classes for a problem is important for successful implementation. The main components of a class are its name, attributes, and operations.

Looking at the interaction diagram shown in figure 4, we can locate the classes and the relationships shown in figure 8. The object of type `ValidationWindow` interacts with the objects of type `CustDetails` and `MainMenu` as shown in figure 4. This situation is presented as association links in figure 8. The classes shown are missing many details that can be specified by examining other sequence and communication diagrams the same classes participate in. Figure 7 presents the transition algorithm from sequence diagrams to class diagrams. As we specify more details for the system under development in terms of scenarios more class details can be determined.

```
Input: sequence diagram
Output: class diagram
Process: for each sequence diagram

          create a class diagram

           for each object in the sequence
                      diagram
              create a class
           end {for}

           for each interaction between
              objects in the sequence diagram
             create an association link
               between the two classes
               representing the participating
                      objects
             create a service in the
               destination class to represent
               the message passed/called
           end {for}

       end {for}

end {process}
```

Figure 7: Specifying Classes from Sequence Diagrams.

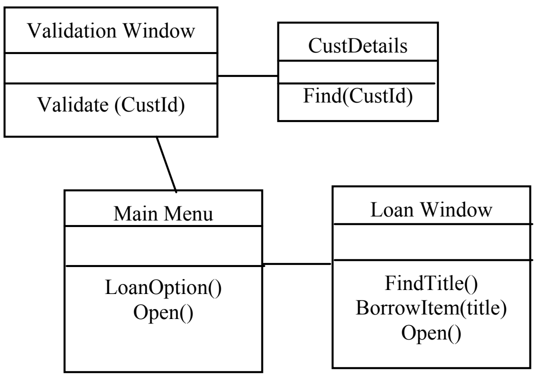


Figure 8: Some Classes Determined from the Sequence Diagram in Figure 4.

After determining navigation for some of the association relationships as shown in figure 8, a class for CustDetails with a public operation `find(CustId)` as shown in the C++ code in figure 10 can be coded. The operation is a signature, and more detailed coding can be specified once it is elaborated upon as an activity diagram. Code for all other classes can be specified in a similar way. Figure 9 presents the algorithm to derive implementation code from class diagrams.

2.4 Relationships

The three most commonly used relationships in OO modeling are ‘dependency’, ‘generalization’, and ‘association’. In a ‘dependency’ relationship, one specification uses another specification. A change in the later may effect the behavior of the first. ‘Generalization’ is a relationship between a general class and a more specific type of that class. The specific class inherits all the properties and relationships of the general class and can modify the inherited behavior or add to it. ‘Association’ is a relationship in which objects of a class communicate with objects of another class. It is also possible that objects of the same class be connected with an association relationship. Roles and multiplicity can be specified at the ends of a relationship. 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 the visibility of an association.

```
Input: class diagram
Output: implementation code
Process: for each class diagram

          for each class in the class diagram
            create an implementation class
               using the class name

           for  each attribute in the class
             create a data member in the
               implementation class
           end {for}

           for each service in the class
             create a function prototype in
               the implementation class
           end {for}

           re-arrange all data and function
           members based on their visibility
           whether private, protected, or
           public
         end {for}

       end {for}

end {process}
```

Figure 9: Coding The Classes

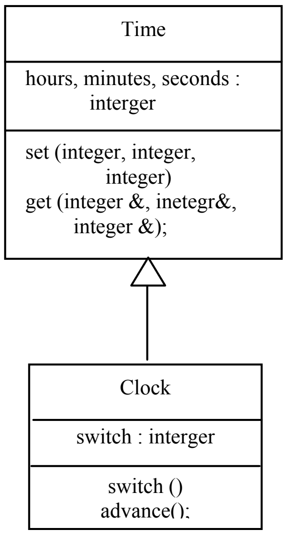
```
Class CustDetails {

        public:

         int find(CustId);
}
```

Figure 10: CustDetails Class.

Analyzing a relationship enables us to specify the necessary implementation code. Figure 11 outlines the procedure to update the implementation code with relationship details. For an ‘inheritance’ relationship, code can be incorporated in the derived class’s header specification to inherit from the base class depending on the access specification specified in the diagram. This situation is shown in figure 12 presenting a diagram demonstrating inheritance relationship between the two classes `Time` and `Clock` and the C++ code to implement the classes and relationship. For a ‘dependency’ relationship a function prototype need to be specified for the dependent class to access/call the class it uses. In an ‘association’ relationship, both classes can be extended with functions to get and set class details at the other end of the relationship. However, if navigation is restricted, the functions need to be specified for the class originating the relationship. Multiplicity, when specified, can be used to determine the input, and output, for the functions.

```
Input: class diagram
Output: implementation code updated with
        relationship specification
Process: for each class diagram

          for each inheritance relationship
              between the classes
            update the subclass's header to
            include the base class name and
            the inheritance specification
          end {for}

          for each association/dependency
              relationship between the
                classes
            update the classes to include
            comments describing the
            relationship with other
              class(es)
           end {for}

         end {for}

end {process}
```

Figure 11: Coding the Relationships.


Time
hours, minutes, seconds : interger
set (integer, integer, integer)
get (integer &, inetegr&, integer &);
Clock
switch : interger
switch ()
advance();

```
class Time {
  protected:
     int hours, minutes, seconds;
  public:
    void set(integer, integer, integer);
    void get(integer &, integer &,
               integer &);
}

class Clock: public Time {
  protected:
     int switch;
  public:
    void switch();
    void  advance( );
}
```

Figure 12: Inheritance Relationship between Classes.

2.5 Activity Diagrams

Activity diagrams are used to specify and document control. They model the dynamic aspects of a system at different levels of abstraction, from highly abstract to more detailed. Sequential and parallel flow of control from one activity to another is documented. An activity is made-up of actions such that when all executed the activity is executed. Figure 13 shows the activity diagram for the “Find a Title” operation. A title is searched for in the object of type `Title` holding all titles in the system. When found, the number of available copies is checked, and the title’s availability status is reported to the calling program. Figure 14 presents a more detailed activity diagram for the action “Search for Title” shown in figure 13. The actions in figure 14 are atomic and can be transformed to implementation code. The titles are searched for one by one until either the list is exhausted or the required title is found and its number of copies is returned.

Reading an activity diagram for an operation, its details can be used to assist the programmer to code the operation. If properly analyzed, other necessary attributes and operations can also be found and specified as part of the code. Figure 15 outlines the procedure to utilize activity diagrams to ease and possibly code operations. The actions, decisions, and flow of control can be inserted as comments in the implementation file.

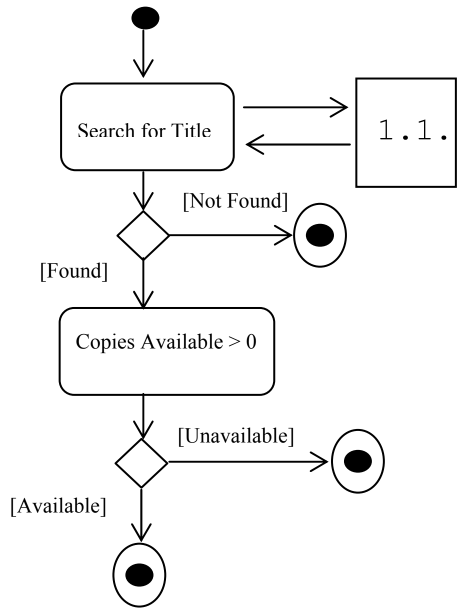


Figure 13: Find a Title Activity Diagram.

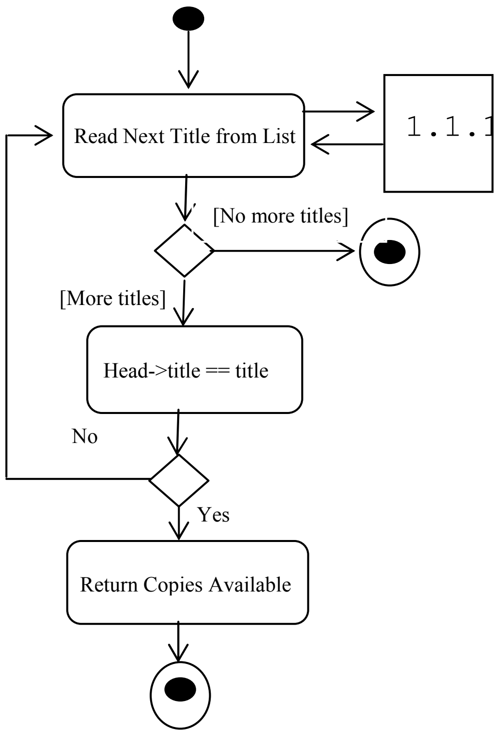


Figure 14: Search for Title Activity Diagram.

2.6 Testing

Two common testing approaches are black box testing and white box testing [19]. A very useful black box testing technique for OO systems is scenario-based testing. In this type of testing, the sequence of events that carry out a specific user need is tested. For example, to carry out a loan operation the sequence of events shown in figure 17 occur.

Similar sequences of events can possibly be generated automatically for all interaction diagrams. Figure 16 outlines the algorithm to generate test cases from sequence diagrams. The scenario-based test case described in figure 17 is specified from the example shown in figure 4 by following the sequence of events from start to end.

```
Input: activity diagram
Output: implementation code
Process: for each activity diagram locate the
            operation it represents

           for each action in the diagram
             - in case of repetition/decision
                follow the links for true and
                false situations
             - insert a line of commented code
                representing the statement in
                the implementation file
           end {for}

       end {for}
End {process}
```

Figure 15: Converting the Activity Diagram to Implementation Code.

```
Input: sequence diagram
Output: scenario-based test cases
Process: for each sequence diagram
          create a scenario-based test case

          for each message in the sequence
            diagram insert it as a testing
            step in the test case
          end {for}

          organize the steps based on the
          messages' sequence numbers

        end {for}

End {process}
```

Figure 16: Specifying Scenario-based Test Cases from Sequence Diagrams.

```
The customer ID is validated.

The main menu is displayed and the loan
operation is selected.

The loan window is displayed.

The title is searched for.

The item is borrowed if found.

The loan transaction is recorded.
```

Figure 17: Scenario-Based Test Case

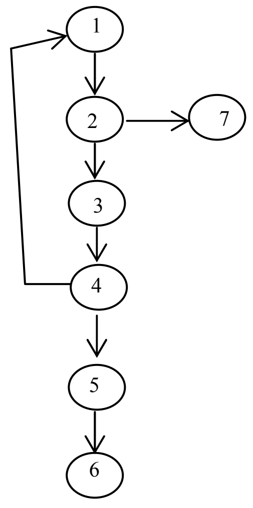


Figure 18: Search for Title Flow Graph.

A common white box testing technique is basis path testing [19]. In it, 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 to derive test cases. For example, a flow graph representing the activity diagram shown in figure 14 can be shown as in figure 18. There are two predicates in the graph. Therefore three independent test paths can be obtained, and they are 1-2-7, 1-2-3-4, and 1-2-3-4-5-6. We can navigate easily from the initial state to the rest of the diagram. An algorithm can be written to navigate the diagram and to visit a certain path once only to generate the test paths.

3. Software Artifacts Representation and Storage.

Earlier research and investigations [16, 10, 25] showed that a graph structure is a suitable representation for Object-Oriented software artifacts. A suitable graph structure is carefully drawn to represent the information for the different development stages. Special care was taken to ensure that the graph structure eases the automated generation of software artifacts whenever possible. A suitable database for such a representation is Jasmine [14] an OO database that can be accessed from a number of commonly used development languages including C++ and Java. In addition to its open interface it has its own development environment through which it can be manipulated and examined. This section explores in detail how each of the main UML modeling elements is mapped to the database.

3.1 Use Case Diagrams

To store details of a use case diagram, four types of classes are defined. The first represents a use case diagram; the second represents an actor; the third represents a use case; and the fourth represents a relationship. Figure 19 shows the database representation for the Loan use case shown in figure 2.

3.2 Interaction Diagrams

Three main classes are required to store details for interactions. The first represents a particular interaction called `'Interaction'`. An interaction is a scenario detailing a specific use case; therefore an interaction object is linked to from a use case object. A particular interaction has a title and contains a list of objects that interact with each other through operation calls. The second class that is required is used to represent an object participating in an interaction called `'Entity'`. Typically an object has a name, a class type, and links to other objects through operation calls. It is important to capture the details of operation calls, therefore, a third class is specified to represent an operation called `'Operation'`. To further decompose details of an operation, a class is specified to represent an operation parameter called `'Attribute'`. To represent a link/operation call between two objects in an interaction a special class is specified called `'Lenk'`. Figure 20 shows partial representation on Jasmine for the Loan Scenario shown in figure 4.

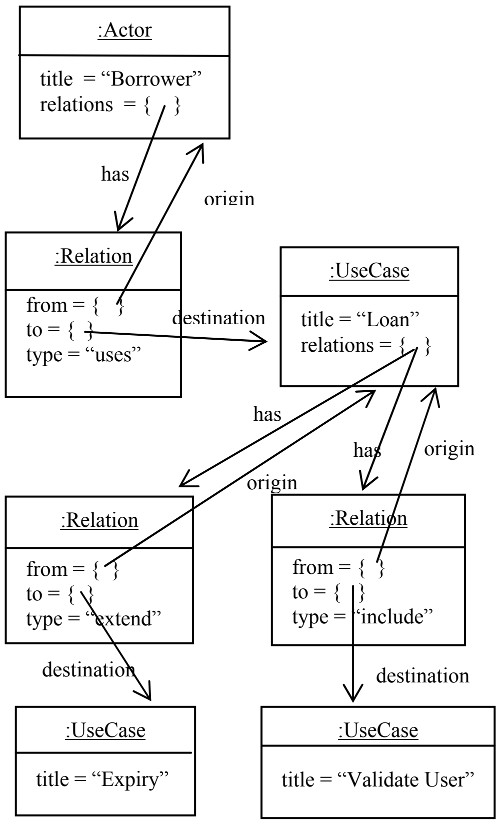


Figure 19: An Example Representation on Jasmine for A Use Case Diagram.

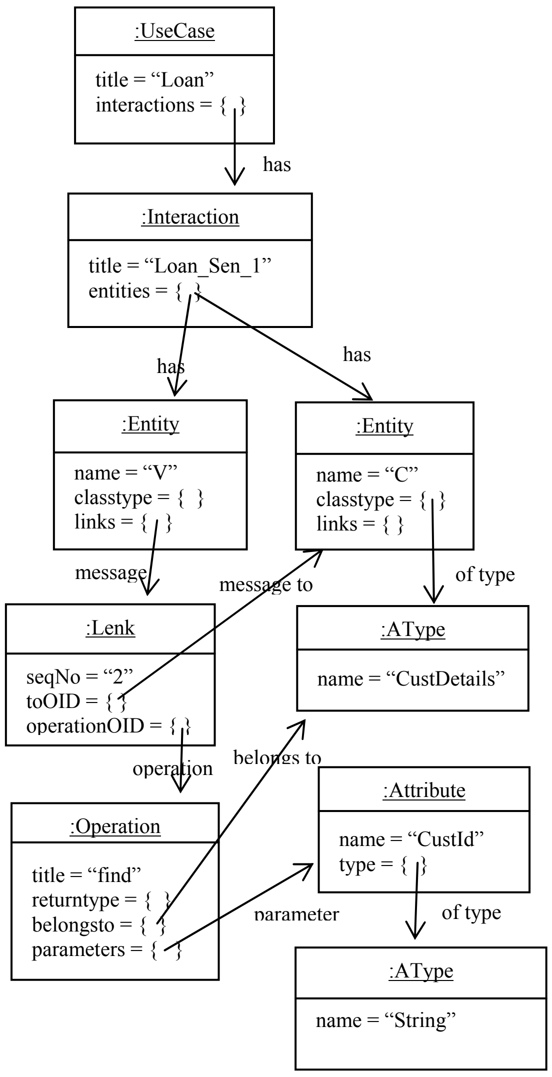


Figure 20: Partial Representation on Jasmine for the Loan Scenario.

3.3 Class Diagrams

As discussed earlier, classes consist of attributes and operations and, in some cases, a special semantic 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. A special jasmine class called `'AType'` is specified to represent a modeling class. Figures 21a and 21b show an example order class and its representation on Jasmine.

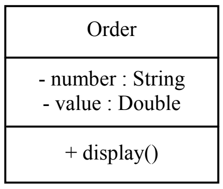


Figure 21a: A Simple Order Class.

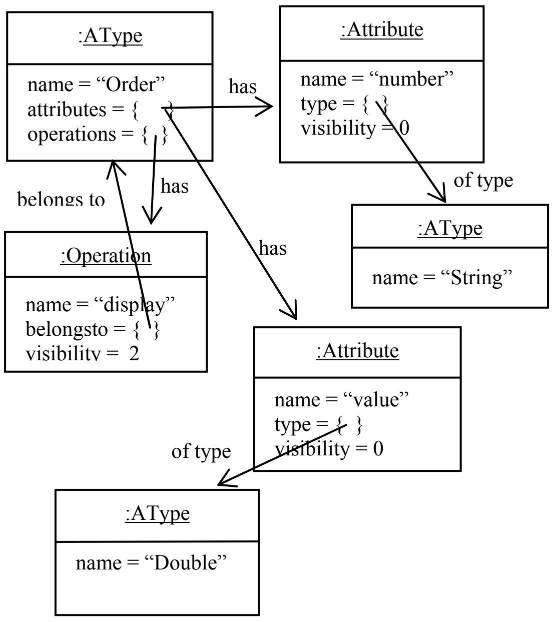


Figure 21b: Representation on Jasmine for the Order Class.

3.4 Relationships

To represent a `'dependency'`, a class definition that holds the object ids of the class objects 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 called `'Relation'` is specified for the purpose. A `'generalization'` relationship can be named, stereotyped and constrained. A new class definition called `'Relation2'` is specified to represent the generalization relationship and inherits from class `'Relation'`. 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'`. Using this class and inheriting from `'Relation2'` a new class definition called `'Relation3'` to hold an association relationship is specified. The new class inherits all details defined for the `'dependency'` and `'generalization'` relationships and adds more that are specific to the association relationship. Figure 22 shows an association relationship between the classes `'Professor'` and `'University'`. It suggests that one or more professors can work for a university.

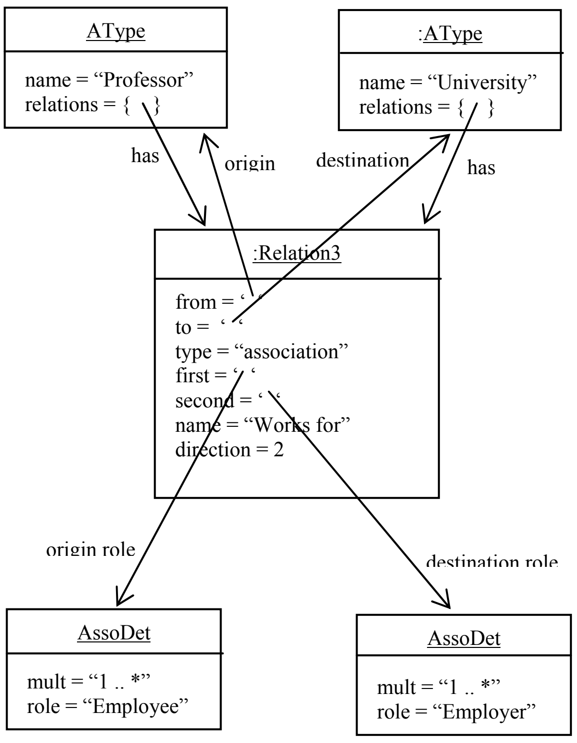


Figure 22: Partial Representation on Jasmine for an Association Relationship.

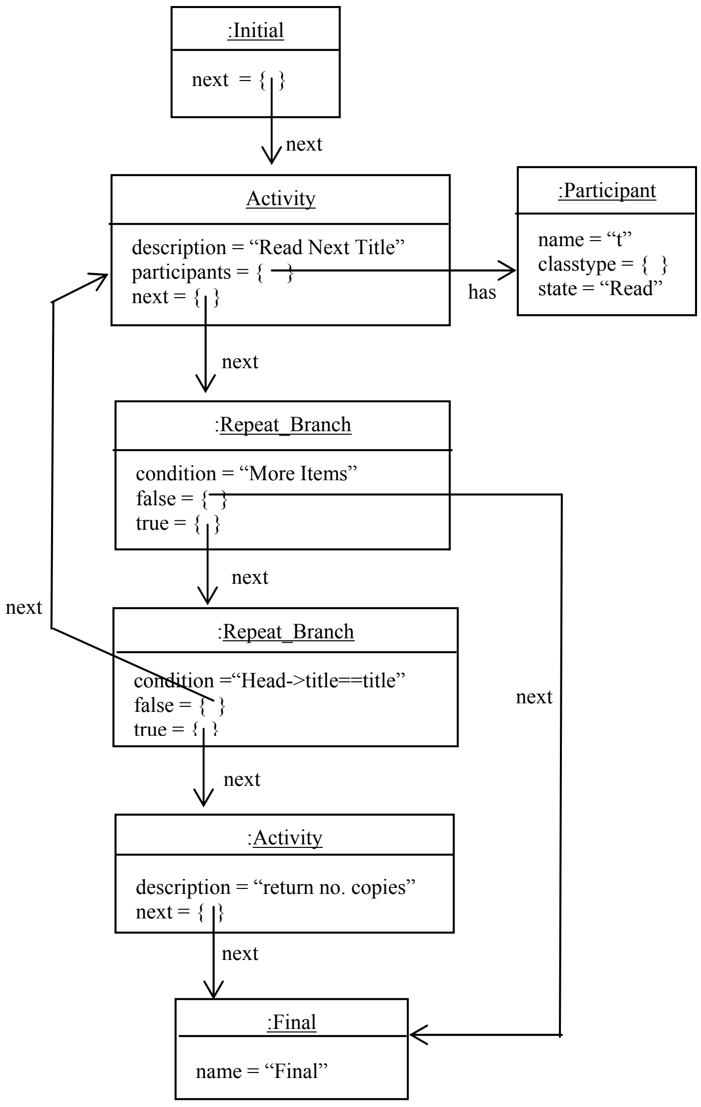


Figure 23: Representation on Jasmine for "Search for Title" Activity Diagram.

3.5 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. A set of classes are defined to represent an activity diagram on Jasmine. The first represents the initial state, the second represents an activity, the third represents an object an activity interacts with, the fourth represents selection and repetition, and the fifth represents the end of an activity. Figure 23 shows the database representation for the activity diagram shown in figure 14.

4. Consistency Management

A major issue in software development is the management of inconsistencies and has been researched by many such as [11, 15, 9, 23, 5, 2, 17, 18]. The discussion in this section is based on the listed publications and many others, but an attempt is made to learn from experiences that specifically address consistency management within and between UML diagrams. UML is a multi view modeling language providing various diagram types for describing a system from different prospective.

Liu, Easterbrook, and Mylopoulos [15] present three classes of inconsistencies within design descriptions and they are redundancy, conformance to constraints and standards, and change. Redundancy occurs when a design artifact is represented more than once in more than one view. Such as when there is an overlap between two views or they have common elements. An typical example of conformance to constraints is the fact that in UML an attribute name must be unique within a classifier. Software development efforts must also conform to best practices standards. Changes are unavoidable in software development and a software design may undergo many changes before it is implemented during of which many inconsistencies may easily be introduced. Grundy, Hosking, and Mugridge[11] identifies four types of inconsistencies structural, semantic, between specifications at different levels of abstraction, and inconsistencies between the work of different developers. Structural inconsistencies relate to parts of a specification in one view not represented by another that should present it, for example, an object defined in a sequence diagram not reflected by the object diagram view. An example of a semantic inconsistency is a service provided by an object in the sequence diagram not supported by the same object class in the class diagram. Inconsistencies between specifications at different levels of abstractions could occur, for example, a sequence diagram that has no use case diagram. The fourth type of inconsistencies may occur if in a multi-development environment in which more than one developer modify the software artifacts concurrently. Many researchers realize that it is impossible and in some cases not desirable to resolve inconsistencies but it is possible to resolve some.

In the suggested development environment presented here the tools are tightly coupled having an advantage to resolve redundancies as specified in [15] and structural inconsistencies as specified in [11]. The graph structure used to represent software information is shared by the different development stages. During a development stage all information related to the software artifacts for that stage are stored. The stage that follows has access to the same software artifacts stored by the stage before; therefore all details already specified are made available for the new stage, and the new stage needs only to add the new details that are not present. Such an approach in information representation that is shared, updated, and used by the various development stages ensures that only one version of any software artifact is present, thus information is consistent and redundancy as specified in [15] is avoided. At the same time a software artifact specified in a diagram is automatically reflected by others, for example, adding a new object in the sequence diagram it is automatically represented by the collaboration diagram, and reflected as a class in the class diagram resolving structural inconsistencies as specified in [11].

A software artifact has little details in an early development stage and as development progresses it is enriched with more details and new artifacts are created, but for the transition to be semantically correct consistency checks must be specified. A number of researchers such as [9, 23, 5, and 2] define consistency checks between UML views. Gomaa and Wijesekera [9] specify three example rules between use case and interaction diagrams, class and state charts, and interaction and state charts. Simmonds and Bastarrica [23] define five consistency checks and they are abstract object, incompatible behavior, multiplicity, class-less instances, and observable behavior conflicts. Derrick, Akehurst, and Boiten [5] suggest that interaction between objects in a sequence diagram must be represented as relationships between the object classes and the objects' called services must be provided by the classes. Bodeveix, Millan, Percebois, and Le Camus [2] specify some inter-diagram coherence rules or consistency checks for class, object, sequence, and state diagrams. Based on these readings and further analysis of UML diagrams a summary of consistency checks between some of the main UML diagrams is as follows:

- **a.** Every use case diagram must have a corresponding interaction (sequence/collaboration) diagram deploying it.
- **b.** Interaction diagram and class diagrams:
  - **i.** Every object in the interaction diagram must have a corresponding class in the class diagram.
  - **ii.** Every message must be represented as a service in the receiving object’s class.
  - **iii.** Communicating objects are represented as a relationship between their classes.
- **c.** Interaction diagram and Scenario-based Test Case
  - **i.** Every interaction diagram represents a testing case of a the use case it deploys.
  - **ii.** Messages between user and objects are listed according to their sequence numbers as testing steps.
  - **iii.** Other messages are also tested.
- **d.** Service and Activity diagram:
  - **i.** Every service in a class is represented by an activity diagram describing the statements/operations executed and their sequence.

e. Activity diagram for a service to Coding steps:

  i. Every action in the diagram represents an execution step in the service.

f. Class diagram to Implementation class:

  i. Every class is represented as an implementation class including attributes and services.

  ii. Classes having a relationship must be visible to each other.

Grundy et al [11] defines semantic inconsistencies with an example as "a type mis-match between method calls or a non-existent method is called" such inconsistencies are avoided in the development environment presented here since the software artifacts are independently represented with high level of granulaty. For example a class is represented as an object on its own in the database that has links to its services and attributes that are also represented as independent objects. Similarly, parameters for services and the services' return types are represented as independent objects in the database that are linked to the service object they belong to, further details of the database representation is discussed in section three. Addition, deletion, and changes to class names, attributes, services, or service parameters are automatically picked-up by the different views. When updating the database object representing the class, service, or attribute other views have no choice but to reflect the update since it is the database's current state. Semantic inconsistencies between the diagrams is further avoided by automating the transition process from a development stage to the next and at the same time ensuring that all consistency checks/rules specified earlier in this section are fully adhered to.

Other inconsistencies such as that between different levels of abstraction and between the work of multiple developers can be avoided by strictly conforming to UML constraints and best practices standards in software development.

5. Automation Experiments

It was necessary to develop a set of CASE tools to test our approach to maintain system information and enable automated transition between the development stages with the generation of software artifacts from the available information whenever possible. Four prototypical tools were developed to maintain the diagrams: use case, sequence, communication, and class as shown in figure 24. They provide the user with an easy to use interface and functionality to maintain the diagrams. For example, the use case editor enables the creation of use cases, actors, and the necessary relationships such as 'include', 'uses', 'extend', and 'inherit'. The sequence diagram editor provides the user with diagrams to draw active objects, actors, and operation calls that can be ordered in time sequence. The communication diagram editor enables the user to create objects, actors, and links in the form of operation calls. Similarly, the class editor enables the creation of classes and all possible relationships. The tools were developed using Java, and the operations to add, retrieve, and delete models from and to the object store use Java's interface with Jasmine referred to as persistent Java (pJ). pJ enables instances of preprocessed Java classes to become persistent and stored as Jasmine objects, and can be accessed using a special set of commands provided by the interface library [14].

To test our automation approach a number of experiments were performed. A set of Java programs is written to populate the database with an example system and process the data at each stage using the algorithms described in section two to generate the system 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 the experiments to test the suitability of our approach in information representation and automatic forward engineering support between the main development stages are presented.

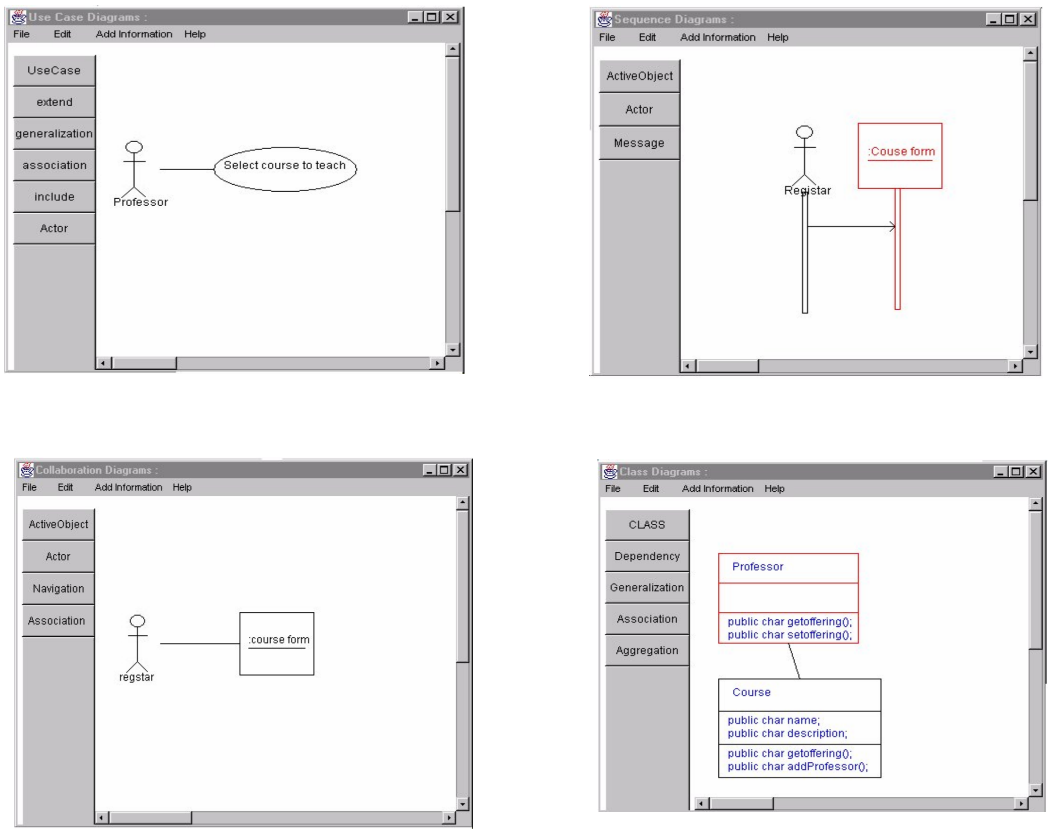


Figure 24: Tools Developed Using Java and Interfacing Jasmine.

5.1 Creating and Processing a Use Case Diagram

At the beginning, the database is populated with a use case diagram consisting of three use cases and an actor as shown in figure 2. 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'`, and an object of type `Actor` to represent 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. Doing so eases navigation between the objects and diagram presentation by the editor. The objects representing the use case diagram are shown in figure 19. The use case diagram is then processed using a special program to generate software artifacts for the stage that follows according to the algorithm shown in figure 3. 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. Transformation at this stage conforms with the consistency check **a** specified in section four.

5.2 Populating and Processing the Interaction Diagram

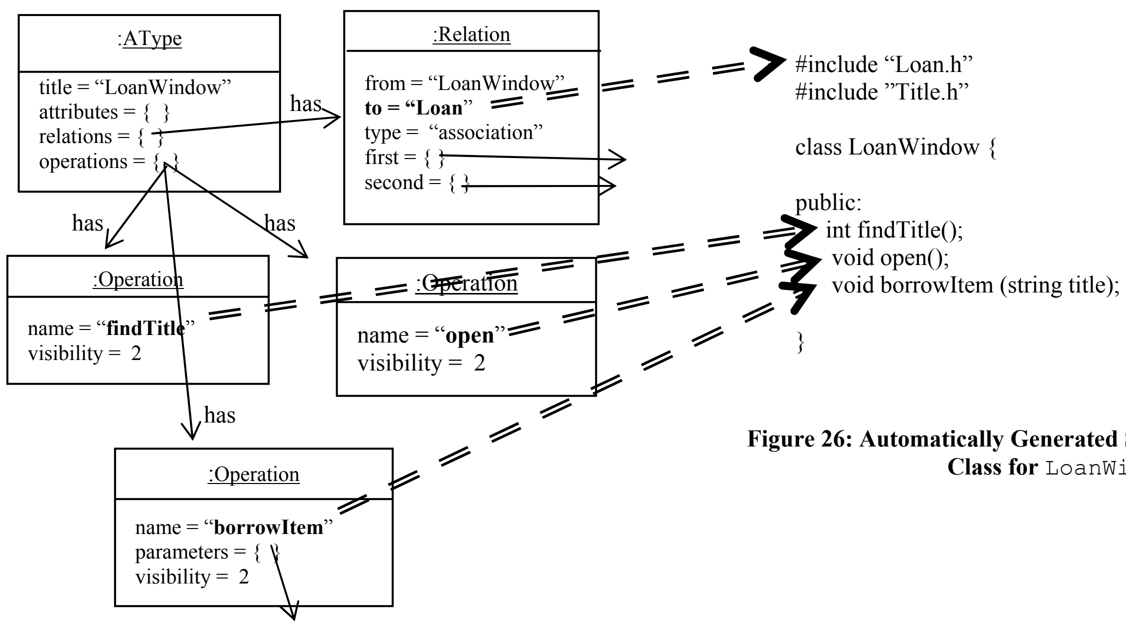
Using a populate program the `Loan` interaction diagram is then elaborated and a set of objects of type `Entity` is created to represent the objects in the diagram, and their details are set. To represent the interactions (operation calls) between the objects in the diagram, objects of type `Operation` are created and their details such as name, return type, and the type of objects they operate 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 communication diagram can automatically be generated by following the algorithm shown in figure 5 and using the same database representation. Figure 4 shows part of the sequence diagram, and figure 20 shows some of the database objects representing it.

The database representation for the interaction diagram conforms with consistency checks **b i** and **ii**. Database objects of `AType` have been created representing the interaction objects' classes and database objects of type `Lenk` and `Operation` have been created to represent the interactions/messages between the objects. Further processing is required to represent the relationships between the classes in the database. The program 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. By doing so conformance with the remaining part of consistency check **b** (i.e. **iii**) is fully achieved and the transformation algorithm in figure 7 is implemented. At this stage the details available in the database are enough to generate the class diagram shown in figure 8. Figure 25 shows a partial representation for the `LoanWindow` class shown in figure 8.

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 by following the algorithms shown in figures 9 and 11. 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. The `AType` object representing the class holds the class name, attributes, and operations. The code generation program reads each attribute and operation and stores them in three different lists depending on their visibility specification whether private, protected, or public. Each attribute has a type and possible initialization details. An operation has a name, a return type and possible parameters. Once they are all read and grouped according to their visibility, suitable code for the class is generated. Figure 26 shows skeleton code for the `LoanWindow` class that is generated from the representation shown in figure 25. More automated details can be generated if necessary, such as set and get functions for the attributes and relations. The transformation steps and generated code conforms with consistency check **f** specified in section four.



```
#include “Loan.h”
#include ”Title.h”

class LoanWindow {

public:
    int findTitle();
     void open();
     void borrowItem (string title);

}
```

Figure 25: Partial Representation for the `LoanWindow` Class.

Figure 26: Automatically Generated Skeleton C++ Class for `LoanWindow`.

5.4 Generating Test Cases

Recent research in automated testing such as that reported by [12, 6] highlight its importance and the need of specialized tools to carry it out. The approach followed in system information representation in the work presented here renders automated test case generation a simpler task.

```
S# Operator         Operation        Operated Upon

1  Borrower:    validate(CustID): ValidationWindow
2  ValidationWindow: find(CustID):    CustDetails
3  ValidationWindow: open():          MainMenu
4  Borrower:        loanOption():     MainMenu
5  MainMenu:        open():           LoanWindow
6  Borrower:        findTitle():      LoanWindow
7  LoanWindow:     find(title):       Title
8  Borrower:      borrowItem(title): LoanWindow
9  LoanWindow:   create(CustID,
                      title,date):   Loan
10 LoanWindow:     update(title):     Title
```

Figure 27: A Scenario-based Test Case Automatically Generated.

To generate a scenario-based test case the algorithm shown in figure 16 is followed, 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. Figure 27 shows the generated scenario-based test case for the Loan interaction diagram shown in figure 4. It shows the sequence of operations necessary to carry out a loan operation, the type of entity responsible for each and the type of object operated upon. The steps followed to generate the test case and the generated test case conforms with consistency check **c** in section four.

6. Conclusions

Incremental software development is the recommended approach for the development of high quality software. Information specified at an early stage is the basis upon which later stages build on leading to the final product. Iterative development further strengthens the product by specifying and implementing more requirements in every iteration. The paper describes an approach that facilitates automatic transitions from a development stage to the next based on a common repository of system information. A transition technique between the development work flows is suggested clearly highlighting the relationship between one work flow and the next. A database representation for each of the UML diagrams is presented. The representation is designed to ensure that system information is uniquely stored and can be shared between the development tools belonging to various development stages. The database used provides all necessary services and tools to maintain the information and at the same time has an open interface with widely used programming languages such as C++ and Java. The experiments carried out to test our approach to software development and automated transitions are presented simulating a software development effort using UML and C++. Four prototypical tools were developed using Java and are interfaced with Jasmine.

Recent research on inconsistencies as presented in section four identifies a number of inconsistency classes. Structural and semantic inconsistencies are overcome as a result of using the graph structure to represent the software artifacts explained in section three that is shared by the development tools. Consistency checks between UML diagrams are also specified in section four based on recent research and are fully adhered to when automatically transforming from a development stage to the next as presented in section five.

It is believed that the approach to software development presented in this paper is unique, and has many advantages. It maximizes automation between the development work flows, thus reducing the development time and effort, and provides a consistent transition mechanism from a development stage to another without the need of translators. Automation includes other software development activities, such as the generation of test cases, and implementation code. By centralizing the information of the system under development in an open repository other project related services can be carried out such as, matrices calculation, generation of reusable components, software reuse, and derivation of project related statistics. New software development tools can also be plugged in to access the repository and share the system information with other tools.

REFERENCES

1. AONIX, **Software through Pictures UML** [online] Available: http://www.aonix.com/. (September 20th, 1999).

2. BODEVEIX J., MILLAN T., PERCEBOIS C., LE CAMUS C., BAZEX P., FERAUD L., and SOBEK R., Extending OCL for verifying UML models consistency. **Workshop on Consistency Problems in UML-based Software Development**. UML 2002. Blekinge Institute of Technology.

3. BOOCH, G., **Object-Oriented Analysis and Design with Applications (2nd ed.).** Menlo Park, CA: Addison-Wesley publishing company, 1994.

4. BOOCH G., RUMBAUGH J., and JACOBSON I., **The Unified Modeling Language User Guide**. Reading, Massachusetts: Addison Wesley Longman, Inc., 1999.

5. DERRICK J., AKEHURST D., and BOITEN E., A Framework for UML Consistency. **Workshop on Consistency Problems in UML-based Software Development**. UML 2002. Blekinge Institute of Technology.

6. DEVANBU Premkumar T., ROSENBLUM David S., and WOLF Alexander L., Generating Testing and Analysis Tools with Aria. **ACM Transactions on Software Engineering and Methodology,** Volume 5, Number 1, January 1996, pp 42-62.

7. ERIKSSON, H., and PENKER, M., **UML Toolkit.** New York, N.Y.: John Wiley and Sons, Inc., 1998.

8. FOWLER, M., and SCOTT, K., **UML Distilled: Applying the Standard Object Modeling Language.** USA: Addison-Wesley Longman, Inc., 1997.

9. GOMAA H., and WIJESEKERA D., Consistency in Multiple-View UML Models: A Case Study. **Workshop on Consistency Problems in UML-based Software Development II**. UML 2003. Blekinge Institute of Technology.

10. GRUNDY, J. C., and HOSKING, J. G., Constructing Multi-View Editing Environments Using MViews. **Proceedings of the 1993 IEEE Symposium on Visual Languages,** IEEE CS Press, August 1993, pp. 220-224.

11. GRUNDY J. C., HOSKING J. G., and MUGRIDGE R., Inconsistency Management for Multiple-View Software Development Environments. **IEEE Transactions in Software Engineering: Special Issue on Managing Inconsistency in Software Development.** Vol. 24, No. 11, IEEE CS Press., 1998.

12. HOWE Adele E., MAYRHAUSER Anneliese Von, and MRAZ Richard T., Test Case Generation as an AI Planning Problem. **Automated Software Engineering**, 4, 1997, pp 77-106.

13. JACOBSON I., BOOCH G., RUMBAUGH J., **The Unified Software Development Process.** Reading, Massachusetts: Addison Wesley Longman, Inc.,1998.

14. KHOSHAFIAN, S., DASANANDA, S., MINASSIAN, N., and KETABCHI, M., **The Jasmine Object Database: Multimedia Applications for the Web.** Academic Press/Morgan Kaufmann, 1998.

15. LIU W., EASTERBROOK S., and MYLOPPULOS J., Rule-Based Detection of Inconsistency in UML Models. **Workshop on Consistency Problems in UML-Based Software Development, 5th Int. Conference on the Unified Modeling Language**, Dresden, Germany, Oct 1. 2002.

16. MEYERS, S., Difficulties in Integrating Multiview Development Systems. **IEEE Software**, Vol. 8, No. 1, 1991, pp. 49-57.

17. NUSEIBEH B., EASTERBROOK S., and RUSSO A., Leveraging Inconsistency in Software Development. **Computer**, 33(4):24-29, 2000.

18. NUSEIBEH B., EASTERBROOK S., and RUSSO A., Making Inconsistency Respectable in Software Development, **Journal of Systems and Software**, 58 (2) 171-180, 2001.

19. PRESSMAN, R. S., **Software Engineering: A Practitioner's Approach (4th ed.)**. Singapore: McGraw-Hill Companies, Inc., 1997.

20. QUATRANI, T., **Visual Modeling with Rational Rose and UML**. Addison Wesley Longman, Inc., 1998.

21. RATIONAL SOFTWARE CORPORATION, **Rational Rose. [online]**. Available: http://www.rational.com/products/rose/index.jtmpl. (September 20th, 1999).

22. SELONEN P., and KOSKIMIES K., Transformations Between UML Diagrams. **Journal of Database Management**. 14(3), 37-55, July-Sept 2003. Idea Group Inc.

23. SIMMONDS J., and BASTARRICA C. M., Descriptions Logics for Consistency Checking of Architectural Features in UML 2.0 Models. **Technical Report**, Departamento de Ciencias de la Computación, Universidad de Chile. January 2005.

24. VISIO CORPORATION, **Visio Enterprise Evaluation Guide.** [online]. Available: http://www.visio.com/products/enterprise/index.html. (September 20th, 1999).

25. WANG, C., LEUNG, C., LONG, F., and RATCLIFFE, M., A PCTE-Based Multiple View Environment. **Proceedings of SEE'93**, 1993.

26. WHITTLE J., Transformations and Software Modeling Languages: Automating Transformations in UML. **Jezequel, J. M., Hussmann H., Cook S. (Eds): UML 2002, LNCS 2460**, pp. 227-242. Springer-Verlag Berlin Heidelberg 2002.