ISSN:0975-9646

Impact Analysis through Regression Test Selection

S. Sushumna, K. Rakesh, G. HimaBindu

Computer Science Department, GITAM Institute of Technology,
GITAM University, AP, INDIA

ABSTRACT: **Change is unavoidable in software development. During the entire lifecycle of a product, the environment changes; the needs of customers or the market change and grow, and with them the requirements on the system being developed. Impact analysis is then defined as the process of identifying the potential consequences (side-effects) of a change, and estimating what needs to be modified to accomplish a change. The use of Unified Model Language (UML) analysis/design models on large projects leads to a large number of interdependent UML diagrams. As software systems evolve, those diagrams undergo changes to, for instance, correct errors or address changes in the requirements. Those changes can in turn lead to subsequent changes to other elements in the UML diagrams. We propose a UML model-based approach to impact analysis that can be applied before any implementation of the changes, thus allowing an early decision-making and change planning process. We present a methodology and tool to support test selection from regression test suites based on change analysis in object-oriented designs. We first verify that the UML diagrams are consistent (consistency check). Then changes between two different versions of a UML model are identified according to a change taxonomy , and model elements that are directly or indirectly impacted by those changes (i.e., may undergo changes) are determined using formally defined impact analysis rules (written with Object Constraint Language) and we propose a formal mapping between design changes and a classification of regression test cases We also present a prototype tools that provides automated support for our impact analysis strategy and test selection from regression test suites, that we then apply on a case study to validate both the implementation and methodology.**

Keywords: **UML, regression test suite, impact analysis, consistency.**

1. INTRODUCTION

The software systems have traditionally been decomposed into subsystems top down according to their functionality. The object-oriented approach describes the system in terms of objects that make up the problem domain. Applying object-oriented technology can lead to better system architectures, and enforces a disciplined coding style. Rum Baugh states that an object-oriented approach produces a clean, well-understood design that is easier to test, maintain, and extend than non-object-oriented designs because the object classes provide a natural unit of modularity.

As time goes by, there are more demands for evolving existing software. Software evolution refers to the on-going enhancements of existing software systems, involving both development and maintenance. As software ages and evolves, the task of maintaining it becomes more complex and more expensive, which is especially true for systems implemented in object-oriented approach.

An update to existing system may need to know the potential impacts. Potential impacts are identified by using UML models in a very easy manner. As software systems evolve, UML diagrams undergo changes. Such changes to a diagram may lead to subsequent changes to other elements of the model diagrams. The (potential) side effects of a change to the unchanged diagrams should be automatically identified to help (1) keep those diagrams up-to-date and consistent and (2) assess the potential impact of changes on the system models and code. This can in turn help predict the cost and complexity of changes and help decide whether to implement them in a new release.

In large software development teams, the above problems are even more acute as diagrams may undergo changes in a concurrent manner as different people may be involved in those changes. Support is therefore required to help a team assess the complexity of changes, identify their side effects, and communicate that information to each of the affected team members. To address the issues, we are focusing on impact analysis of UML analysis or design models.

Most of the research on impact analysis is based on the program code (implementation). However, in the context of UML-based development, it becomes clear that the complexity of changing Analysis and design models is also very high.

While code-based impact analysis methods have the advantage of identifying impacts in the final product of the code, they require the implementation of these changes (or a very precise implementation plan) before the impact analysis can be performed. However, a UML model-based approach to impact analysis looks at impacts to the system before the implementation of such changes. A proper decision can therefore be made before any detailed implementation of the change is considered on whether to implement a particular (set of) change(s) based on what design elements are likely to get impacted and thus on the likely change cost.

The identification of model inconsistencies is important to ensure that the impact analysis algorithms to get correct results. To find inconsistency, it is beneficial to know what causes the inconsistency and to decide how to fix it.

We do not believe that a tool can automatically resolve inconsistencies because a tool cannot know whether an inconsistency is tolerable or why it was caused. However, a tool can be an assistant that provides the facts the designer must consider. This work demonstrates that it is feasible to *locate all choices for fixing inconsistencies* and to *predict their positive and negative side effects.* However, inconsistencies are not independent events. If a choice for fixing one inconsistency inadvertently affects how to fix another one then the designer should know about this dependency. This work thus also demonstrates how to *identify dependencies among inconsistencies.* No existing work is able to identify all choices for fixing inconsistencies. Also, to the best of our knowledge, no existing work is able to identify dependencies and predict side effects. UML/Analyzer tool relies on the UML Interface Wrapper

component. The infrastructure exposes the modeling data in an UML-compliant fashion. It also employs a sophisticated change detection mechanism. The latter is particularly important because it notifies our tool of changes to the UML model in real time while the designer uses the modeling tool. The consistency rules themselves are hard coded into the logic of the UML/Analyzer tool

In software development, most work on impact analysis focused on source code. Some of these techniques emphasized on static or dynamic program slicing. Other techniques emphasized on traceability. Bohner-Arnold discussed many of these approaches. To narrow down, what part of the system to change and/or what part to reanalyze/retest after the change. These approaches are very powerful but do not readily apply to (UML) design models. An approach for instant consistency checking of UML models was fully automated and correctly decided what consistency rules to re-evaluate when a model changed. We used profiling data to establish a correlation among model elements and consistency rules to decide what consistency rules to reevaluate with changes.

Once we have verified that the diagrams of a UML design model are consistent, and model element changes have been detected, the next step is to automatically perform impact analysis using impact analysis rules, that is, rules that determine what model elements could be directly or indirectly impacted by each model element change. The original test set from which to select can contain both functional and non-functional system test cases. From a UML standpoint, functional test cases test complete use case scenarios. .

The regression testing is to test a new version of a system so as to verify that existing functionalities have not been affected by new system features. Regression test selection is the activity that consists in choosing, from an existing test set, test cases that can and need to be rerun to ensure existing, unmodified functionalities are still working correctly. The main objective of selecting test cases that need to be rerun is to identify regression test cases that exercise modified parts of the system. To achieve this objective, we need to classify test cases in an adequate manner so we classify test cases as follows:

1.1 Obsolete:

A test case that cannot be executed on the new version of the system as it is 'invalid' in that context. Classifying a test case as obsolete may lead to either modifying the test case and corresponding test driver or removing the test case from the regression test suite altogether.

1.2* Retestable:*

A test case is still valid but needs to be rerun for the regression testing to be safe.

1.3 Reusable:

A test case that is still valid but does not need to be rerun to ensure regression testing is safe.

We focused on automating regression test selection based on architecture and design information represented with the Unified Modeling Language (UML) and traceability information linking the design to test cases.

2. REGRESSION TEST SELECTION TOOL (RTSTOOL)

2.1 Functionality:

The RTSTool main functionality (see Figure 1) is to classify regression test cases as obsolete, re testable, and reusable, based on the design information of the old and new system versions and traceability information between the UML design and test cases. Its inputs are the UML diagrams of two system versions (XMI files produced by UML case tools) along with the original regression test suite. It then compares the two versions of each diagram type (class, use case, and sequence), realize some consistency checks, and classifies test cases. Future functionalities that can be easily added to the current architecture include the generation of new regression test cases based on the new versions of UML diagrams. The reader is referred to for more details. Furthermore, the results of the impact analysis (i.e., added, deleted, and changed model elements) can easily be used for other purposes than regression test selection, e.g., to assess the effort of producing the new version or to make a decision on whether to include a change in the next version. For the sake of brevity we do not present the use case model of the tool, though we will refer to some specific use cases in the remainder of the text.

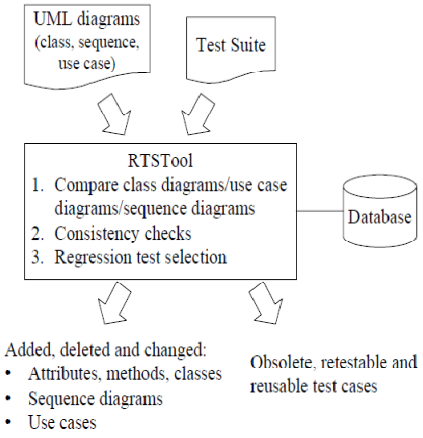

Figure 1: RTSTool Overview

2.2 Test Cases and Traceability

We describe here how the traceability between test cases and sequence diagrams is represented and implemented.

2.2.1 Representation of Test Cases

Any test case is associated with a sequence (ordered set) of triplets: (action name, source classifier name, target classifier name). It specifies the sequence of actions resulting from a test case. In the test driver, a functional test case will consist of operation invocations, signals being sent and object creations as well as destructions, when the language permits. All the messages to boundary classes will directly or indirectly trigger subsequent actions so as to complete a use case scenario. We associate the complete action sequences to test cases as determining changes in non-boundary actions

will be necessary to safely classify test cases as reusable or re testable.

2.2.2 Representation of Sequence Diagrams

In the same way as the test sequences, messages in sequence diagrams are triplets (message label, source classifier name, target classifier name). However, the information about messages is more complete as, in addition to action names, we have possible arguments, guard condition and iteration expressions in message labels. Furthermore, in order to represent every possible message sequences in sequence diagrams, each sequence diagram is represented using a regular expression whose alphabet is composed of the above triplets. This facilitates automation in our algorithms since we can then easily check whether a test case is a legal sequence of a regular expression (i.e., a sequence diagram), and therefore whether a test case can be executed given the design described by a sequence diagram.

2.2.3 Traceability

To automate test selection, we need to have traceability between the UML design and regression test cases, so that we can determine the effect of design changes on those test cases. Traceability is simply handled by the association between test cases and sequence diagrams, each test case testing a use case scenario. We therefore implement traceability as a mapping between sequence diagram scenarios and test cases. A test case exercises, for each use case it executes, only one scenario but one scenario can be exercised by several test cases.

More precisely, in order to identify which sequence diagram is triggered by a particular test case we first extract from the test case sequence of triplets the ones whose target classifier is a boundary class, and thus form the sequence of actions on boundary classes. We then have to find the sequence diagram with the same sequence of actions on the same boundary classes. Since a test case may be composed of a sequence of different sequence diagrams we need to repeat this matching until we have found the complete sequence of sequence diagrams which this test case exercises.

2.3 Architecture

We first provide the design goals that have driven the definition of the architecture and then the architecture is presented in terms of UML packages and their dependencies.

2.3.1 Design Goals

They are clearly identified so as to refer to them in the next section:

- DG1: The tool must be independent of any specific UML case tool.
- DG2: To refine the test case classification, we should consider making use of additional UML diagrams, e.g., state charts.
- DG3: We want to limit the impact of future changes to the test case representation, the XMI standard, and the UML standard.

2.3.2 Packages

The RTSTool architecture is made of seven packages (see Figure 2), four of them being of particular interest as they contain classes that implement the identification of changes in class, use case, and sequence diagrams and their impact on regression test cases: i.e., RTSTool, Class Diagram, Use Case Diagram, and Regression Test Suite. The package RTSTool contains the class that starts up the system (main) and the classes performing the consistency checks across diagrams. The last three packages, as indicated by their name, encapsulate classes and operations performed on class diagrams, use case diagrams (and their corresponding sequence diagrams), and the regression test suite.

Three additional packages (i.e., RTSTool GUI, Text Parser, and XMIParser) describe the Graphical User Interface and parsers that are able to read XMI files (containing the information on UML models) and text files (containing test cases). Note that the architecture described in Figure 2 is not complete to avoid cluttering: It only shows public classes in each package and some associations have been omitted.

The ClassDiagramChanges class is responsible for comparing two class diagram versions (association class with class ClassDiagram) loaded from two XMI files produced by a UML case tool (Figure 2). The Class Diagram class is associated (Figure 3) with several classes (Class instances), themselves associated with several attributes and operations (Operation), and possibly with parameters (Formal Parameter). Relationships between classes in the class diagram are also represented (class Relationship). The access Operations association describes the mapping between attributes and the operations that “use” them. Recall that this mapping can be done using operations’ contracts, which specify which attributes are (potentially) read or updated. Note that these classes and relationships are an adaptation of the UML class diagram Meta model. Following the definitions in, classes in Figures 1, 2 and 3 are classified as either Control or Entity. Control classes are responsible for realizing a use case by invoking the right sequence of operations (e.g., ClassDiagramChanges realizes the CompareClassDiagramUsecase) and entity classes are repository classes modeling application domain entities. In terms of attributes, all entity classes have a name, and can be further defined by a version (i.e., class ClassDiagram) and a type (classes Operation, Attribute, and Formal Parameter). From two versions of a class diagram, the Class Diagram Changes class produces the sets of added, deleted and changed attributes, operations and classes. This information is then used in the RTSTool and Regression Test Suite packages.

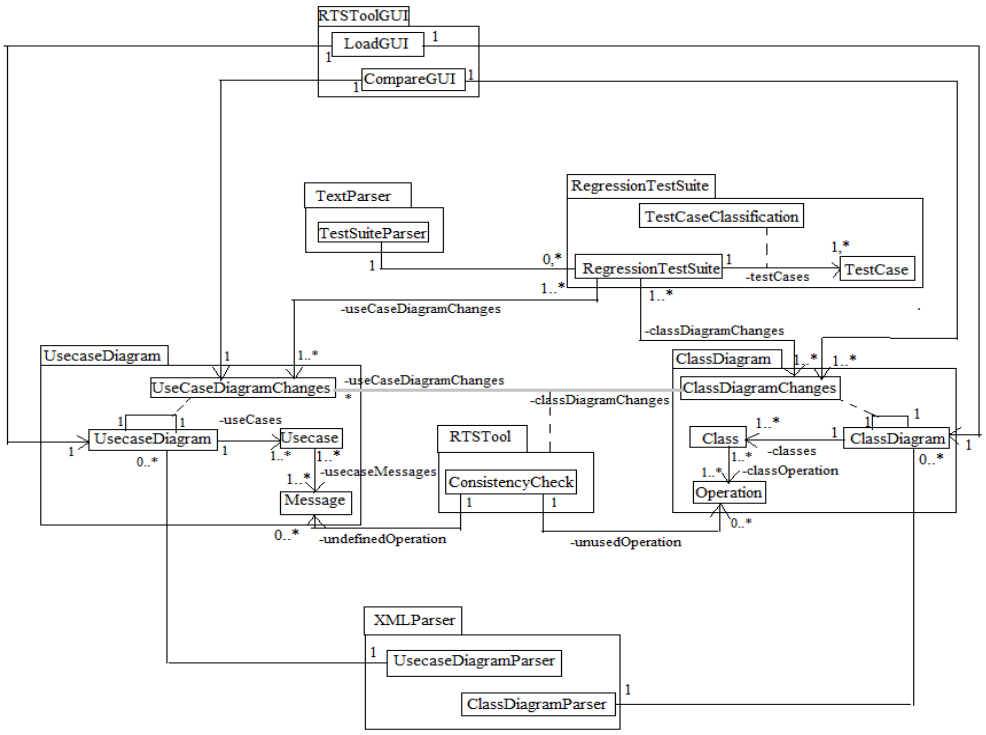

Figure 2 : Architecture of the RTSTool (simplified8)

Similarly, the UseCaseDiagramChanges class (UseCaseDiagram package) is responsible for comparing two versions of a use case diagram, each use case diagram (class UseCaseDiagram) being associated with several use cases (class UseCase). Relationships between use cases are also accounted for (association class UseCaseRelation). Since each use case corresponds to one sequence diagram (attribute sequences in UseCase is a regular expression representing the possible message sequences in the sequence diagram), the Use Case class is associated with messages (class Message), i.e., the messages triggered in the sequence diagram (see Figure 4). These classes and relationships are also an adaptation of the UML use case and sequence diagram Meta models. From two versions of a use case diagram, the UseCaseDiagramChanges class produces the sets of added, deleted and changed messages and use cases. This information is then used in the RTSTool and RegressionTestSelection packages.

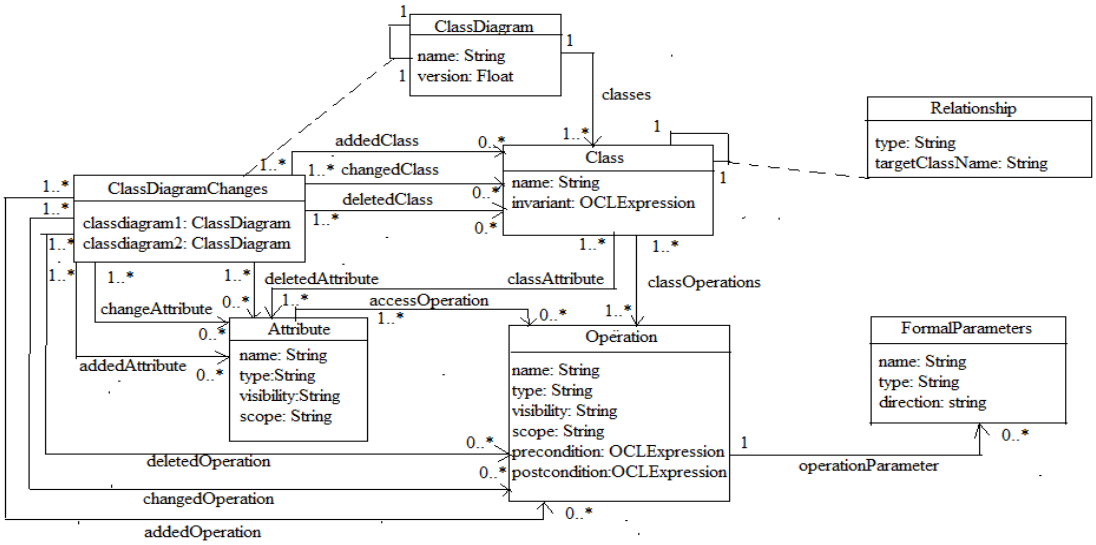

Figure 3: Class Diagram Package

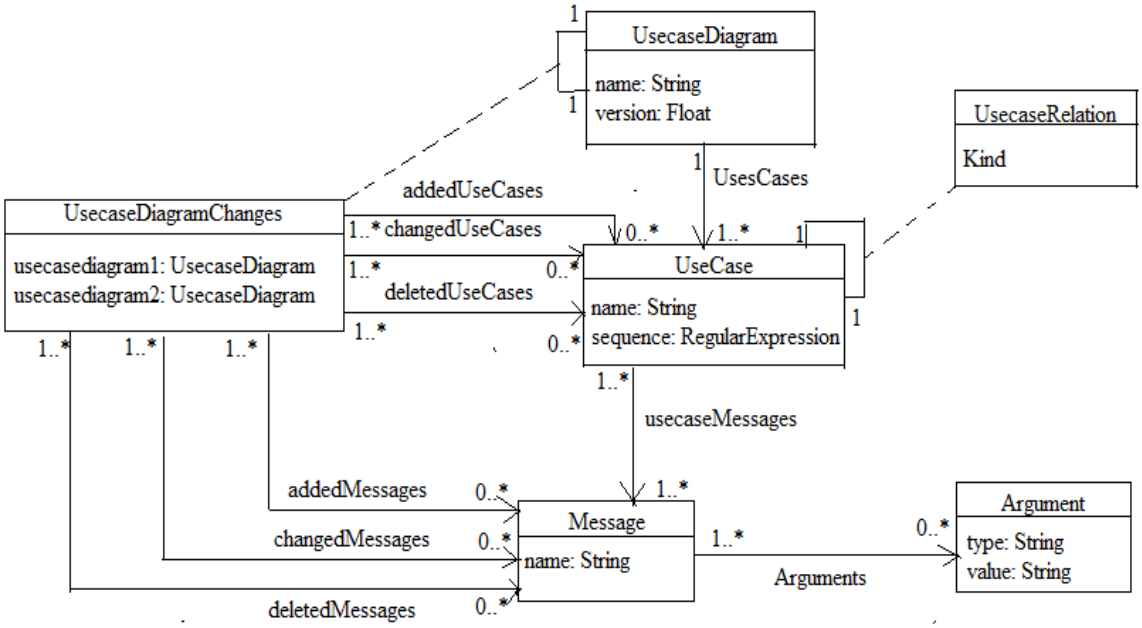

Figure 4 – Use Case Diagram Package

The RegressionTestSuite package is responsible for loading (class RegressionTestSuite uses class Test Suite Parser in package Text Parser), storing (class TestCase) and classifying the different test cases (association class TestCaseClassification), using the information provided by the Class Diagram and the UseCaseDiagram packages (see Figure 2). Class RegressionTestSuite generates the three different sets of test cases defined in previous sections: obsolete, re testable, reusable (see Figure 5). Note that, to compare test cases and sequence diagrams, test cases and sequence diagrams are represented as sequence of messages and regular expressions. Classes Message and Argument are therefore reused from package UseCaseDiagram.

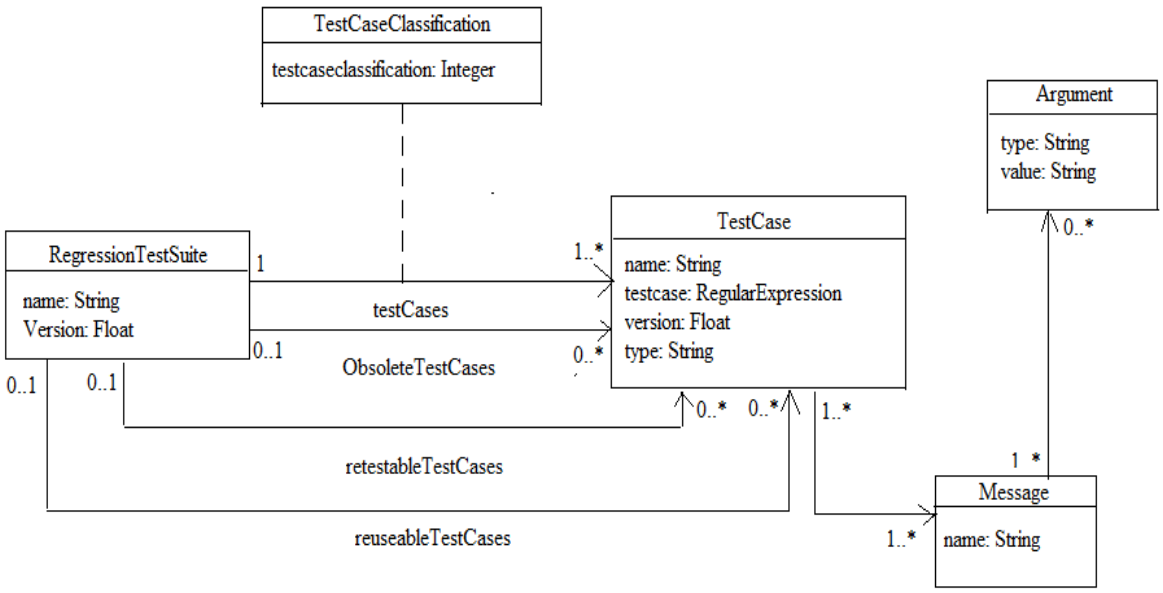

Figure 5 – Regression Test Suite Package

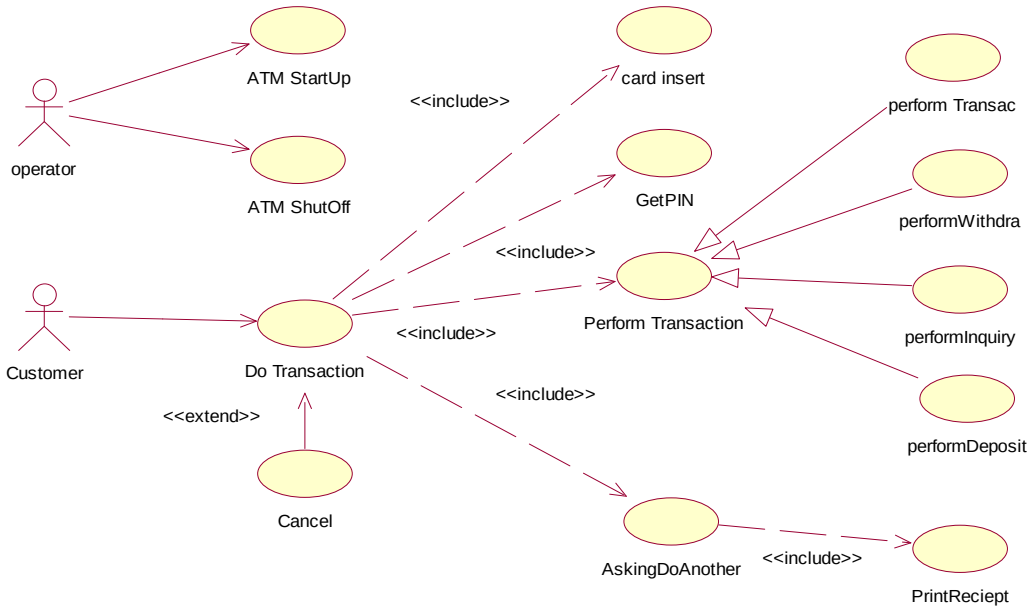

Figure 6: Use case diagram for the ATM case study

If we now make the mapping between our architecture and our original design goals explicit, we see that:

- DG1 is simply achieved by using XMI as a data interchange format.
- DG2 is achieved as new UML diagrams will correspond to new packages, which will be responsible of identifying changes on the new diagrams and letting other packages know about implications on changes in class diagrams and sequence diagrams.
- DG3 is achieved by ensuring that representations of UML class diagrams and test suites are encapsulated within their respective packages and unknown to other packages.

2.3.3 Technical Features

The RTSTool uses an object-oriented database management system to store the different versions of UML models and test cases, thus allowing the reuse of previously loaded diagrams or test cases. As the information stored is complex and can be voluminous, it is important to use a database management system (DBMS). An object-oriented DBMS is a natural choice here as the database schema can match our class diagram and as performance, which is the factor that usually favours the use of relational DBMS, is not an issue with the volume of data we are commonly handling.

The XMI parser embedded into the RTSTool uses the SUN's Java API for XML Processing (eXtensible Markup Language). This package provides classes and operations that enable applications to parse and transform XML documents, and thus XMI documents, using the Document Object Model (DOM). DOM specifies a tree-based representation for XML documents, which is easy to navigate.

The RTSTool is implemented with Java (Java 2 Platform, Standard Edition version 1.49)

3. CASE STUDY

In this section we apply our methodology, using the RTSTool, we could define a variety of changes so as to make the study more diverse and interesting. We first describe the system and then discuss the changes that were performed and present the results of the regression test selection.

3.1 An Automated Teller Machine System

The case study is an Automated Teller Machine (ATM) system. The ATM design model contains 20 classes, 74 operations, 31 attributes and 15 use cases. The ATM's main function is to perform transactions based on the user's inputs. Four types of transactions can be carried out – Deposit, Withdraw, Transfer and Inquiry. What is specific to the use case diagram for the ATM is that all of the use cases, except for two of them, depend on one main use case, doTransaction. All the other use cases are either inclusions or extensions of doTransaction. This main use case describes the details of how the system performs transactions. The two other use cases describe the start up and shut down procedures of the ATM. The test set for the system contains 30 functional test cases which were developed using the methodology. Most test cases test a different transaction, combination of transactions or error conditions which may arise when performing a transaction. However almost all of these test cases execute the same main high-level use case, doTransaction, therefore we can already foresee that if there

is a change to doTransaction all these test cases will be classified as re-testable. Two test cases exercise the start up and shut down procedures of the ATM.

Four different logical changes were performed from this original design, and we present them, as well as the result in terms of regression test selection, in the following sub-sections.

3.1.1 First logical change (version 2 of the ATM)

Description:

The first logical change has to do with how many times a user could enter an incorrect PIN number. In the original system there was no limit to the number of times a customer could enter an incorrect PIN. In the new version a user has only three attempts to enter a valid PIN before their card is retained by the system. This logical change translates into 1 new attribute named numOfTries in the ATM class which keeps track of how many times a customer has entered the PIN. Operation getPIN() in class ATM has a new pre-condition pertaining to the value of numOfTries. Three operations are added to the ATM class: resetNumTries(), incrementNumTries() and getNumTries(). Operation displayRetainCard() is added to the Display class. Each of these new operations appears exactly once on the sequence diagrams. resetNumTries() is added to the CardInsert use case and causes the operation which called it, getCardNum() (in class ATM), to be classified as changed increment NumTries() is added to the GetPIN use case and getNumTries() and displayRetainCard() are added to the doTransaction use case. getNumTries() and displayRetainCard() are both called by the same operation, doTrans() (in class ATM), which is therefore classified as changed. The RTSTool generated the results in Table 1.

	Total (V.1)	Added	Changed	Deleted	Total (V 2)
Attributes	31	1	0	0	32
Operations	74	4	3	0	78
Classes	20	0	2	0	20
Use cases	15	0	3	0	15

Table 1 – Impact Analysis Results using RTSTool for the ATM (first logical change)

Regression Test Selection:

As shown in Table 2, of the 30 test cases in this study, 28 were classified as retestable and only 2 were reusable. The reason for this lies in the type of change that was made. The 28 test cases that are retestable all contain a call to the operation getCardNum() since all of these test cases explore situations where the user wants to perform a transaction, and in order to perform a transaction the user must first enter a card into the machine. Although the change description has to do with how many times the PIN is entered the GetPIN use case is not the only one affected: CardInsert and doTransaction are also affected by these changes and it is actually the change to CardInsert that causes all of the 28 test case to be retestable. The 2 test cases that are reusable are the ones that test the start up and shut down procedures and do not involve the user putting a card in the machine and entering a PIN number.

Test Cases			
Amount	Obsolete	Re testable	Re usable
30	0	28	2

Table 2 – Regression Test Selection Results using RTSTool (first logical change)

3.1.2 Second logical change (version 3 of the ATM)

Description:

The second logical change imposes some extra restrictions on the savings type of account. In the original system savings and cheque accounts were identical in terms of which transactions could be performed on them. In the new version a user cannot withdraw money directly from an account of type savings. Therefore in order to remove their money from a savings account they must first transfer the money to a chequing account. This translates into the following changes: The Constants class is changed because an attribute, INVALID_TRANS has been added. INVALID_TRANS is an error code representing the situation when a customer attempts to perform a withdrawal from a savings account. Class Withdrawal is changed because its operation doTransaction() which performs a withdrawal transaction has a changed post condition. The doTransaction use case is changed because the operation displayErrorMsg() has been added. Operation doTrans() in the doTransaction use case, which calls displayErrorMsg() is classified as changed. The RTSTool gives the results in Table 3.

	Total (V.1)	Added	Changed	Deleted	Total (V 2)
Attributes	31	1	0	0	32
Operations	74	0	2	0	74
Classes	20	0	3	0	20
Use cases	15	0	1	0	15

Table 3 – Impact Analysis Results using RTSTool for the ATM (second logical change)

Regression Test Selection:

When first reading the change description one would come to the conclusion that only test cases involving a withdrawal transaction would need to be retestable. However since this change resulted in a new error condition which is checked after each transaction, all 25 test cases which contain a call to doTrans() are considered re testable (see Table 4). Operation doTrans() represents the execution of a transaction. The 5 test cases which are reusable explore the following situations: start-up, shutdown, card not readable, user presses the cancel button when the PIN is requested and the user presses the cancel button when the transaction type is requested. In these 5 test cases operation doTrans() is never called which is why they are classified as reusable.

Test Cases			
Amount	Obsolete	Re testable	Re usable
30	0	25	5

Table 4 – Regression Test Selection Results using RTSTool (second logical change)

3.1.3Third logical change (version 4 of the ATM)
Description:
The third logical change has to do with the cash dispenser. The current dispenser only holds twenty dollar bills – therefore all withdrawals need to be in multiples of twenty. The new dispenser will be able to handle twenty and five dollar bills. This results in 4 new operations to the CashDispenser class, each of which appears once in the updated sequence diagrams: numberOfFives(), numberOfTwenties(), dispenseFives() and dispenseTwenties(). They all appear in the doTransaction use case (i.e., in the corresponding sequence diagram) and are all called by the same operation, i.e., dispenseCash() (in class CashDispenser), which is therefore considered changed. The results from the RTSTool are presented in Table 5.

	Total (V.1)	Added	Changed	Deleted	Total (V 2)
Attributes	31	0	0	0	31
Operations	74	4	1	0	78
Classes	20	0	1	0	20
Use cases	15	0	1	0	15

Table 5 – Impact Analysis Results using RTSTool for the CCS (third logical change)

Regression Test Selection:
There are only 9 test cases which call the operation dispenseCash(). It is these 9 test cases that are classified as retestable (see Table 6). The other 19 test cases either perform startup, shutdown, a transaction other than withdrawal or a withdrawal in which an error occurs before the cash is dispensed. In contrast with the second change this version of the system behaves as one would intuitively think: A change has been made to the way a withdrawal transaction is executed and only the test cases which exercise the corresponding behaviour need to be retested.

Test Cases			
Amount	Obsolete	Re testable	Re usable
30	0	9	21

Table 6 – Regression Test Selection Results using RTSTool (third logical change)

4. CONCLUSION

In some cases, the number of reusable test cases represented a large proportion (up to 100%): It seems to indicate that substantial savings can be obtained, especially that the whole process can be automated. However, the case studies have shown that changes can have a widely variable impact on the resulting system. Large numbers of test cases may be obsolete, retestable, or reusable. In some cases the results are intuitive; in others the RTSTool was useful to uncover unexpected retestable test cases. But in general, we expect such a technology to be even more useful for large systems, involving many designers in diagram changes, when no one person has a comprehensive understanding of all the use cases and their design. In such a system, a manual impact analysis would likely lead to errors, especially in a context with typical project pressures.

REFERENCES

[1] S. Bennett, S. McRobb and R. Farmer, Object-Oriented Systems Analysis And Design Using UML, McGraw-Hill, 2nd Ed., 2002.
[2] G. Booch, J. Rumbaugh and I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley, 1999.
[3] L. C. Briand and Y. Labiche, "A UML-Based Approach to System Testing," Software and Systems Modeling, vol. 1 (1), pp. 10-42, Springer, 2002.
[4] L. C. Briand, Y. Labiche and L. O'Sullivan, "Impact Analysis and Change Management of UML Models," Proc. IEEE International Conference on Software Maintenance, Amsterdam, The Netherlands, 22-26 September, 2003.
[5] B. Bruegge and A. H. Dutoit, Object-Oriented Software Engineering - Conquering Complex and Chalenging Systems, Prentice Hall, 2000.
[6] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. GilChrist, F. Hayes and P. Jeremaes, Object-Oriented Development - The Fusion Method, Object-Oriented Series, Prentice Hall Ed., 1994.
[7] M. J. Harrold, "Testing Evolving Software," Journal of Systems and Software, vol. 47 (2-3), pp. 173-181, 1999.
[8] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha and S. A. Spoon, "Regression Test Selection for Java Software," Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), Tampa Bay, Florida, USA, October 14-18, 2001.
[9] H. K. N. Leung and L. White, "Insights into Regression Testing," Proc. IEEE International Conference on Software Maintenance (ICSM), Los Almitos, CA, pp. 60-69, October 16-19, 1989.
[10] H. K. N. Leung and L. J. White, "A Cost Model to Compare Regression Test Strategies," Proc. Conference on Software Maintenance, Sorrento, Italy, pp. 201-208, October 15-17, 1991.
[11] P. Linz, An Introduction to Formal Language and Automata, Jones and Bartlett, 2nd Ed., 1997.
[12] B. Meyer, "Design by Contracts," IEEE Computer, vol. 25 (10), pp. 40-52, 1992.
[13] R. Mitchell and J. McKim, Design by Contract, by Example, Addison-Wesley, 2001.
[14] OMG, "Unified Modeling Language (UML)," Object Management Group V1.4, www.omg.org/technology/uml/, 2001.
[15] OMG, XML Metadata Interchange (XMI), www.omg.org/technology/documents/ formal/xmi.htm, 2001
[16] Poet, POET Object Server Suite, the Essential Database for Java and C++ Objects, 2000.
[17] G. Rothermel and M. J. Harrold, "Analysing Regression Test Selection Techniques," IEEE Transactions on Software Engineering, vol. 22 (8), pp. 529-551, 1996.

AUTHERS BIOGRAPHY

S. Sushumna is pursuing M.Tech in Software Engineering from GITAM University, Visakhapatnam, INDIA. My research areas include change management, software process models, software coupling and cohesion.

K. Rakesh is pursuing M.Tech in Software Engineering from GITAM University, Visakhapatnam, INDIA. My research areas include change management, Software cost estimation, software coupling and cohesion.

G. HimaBindu Asst. Professor in the Department of CSE at GITAM University, Visakhapatnam, INDIA.