IMPLEMENTING A DOCUMENT-BASED REQUIREMENTS TRACEABILITY: A CASE STUDY

Suhaimi Ibrahim,
Norbik Bashah Idris
Centre For Advanced Software Engineering,
Universiti Teknologi Malaysia,
Kuala Lumpur,
Malaysia
(suhaimi,norbik)@case.utm.my

Malcolm Munro
Department of Computer Science,
University of Durham,
United Kingdom
malcolm.munro@durham.ac.uk

Aziz Deraman
Fac. of Technology & Infor. System,
Universiti Kebangsaan Malaysia,
Selangor, Malaysia
a.d@pkrisc.ukm.my

Abstract

Requirements traceability as being mandated by many standards governing the development of systems (e.g. IEEE/EIA 12207) is undoubtedly useful to software maintenance. To many organizations, it is viewed as a measure of system quality and is treated as an important component of their efforts towards achieving process improvement (CMM). However, not much elaboration on what types of information needed and how a strategy to achieve this is described in the development standards and guidelines. In this paper, we present our approach in implementing a requirements traceability derived from system documentation. It provides visibility into a system composing of different artifacts that include requirements, test cases, design and code. Our approach supports the top down and bottom up traceability in response to tracing for the ripple-effects. We developed a traceability tool, called Catia and applied it to a case study of system documentation and discussed the results.

Key Words: documentation standards, requirements traceability, call graphs, impact analysis

1. Introduction

Documentation is one of the important items in software maintenance. Tryggeseth [1] had demonstrated in his experiment that documentation significantly improves maintainer's ability to understand a system and make efficient change to it. Despite this, many developers are still reluctant to use anything but the source code for maintenance tasks, as documentations are generally considered abstracts and not up-to-date. However, many agree that the use of documentation is more significant if the information it contains are traceable between software models [2]. We mean software models are the work products such as code, design and requirements that differ from one another in terms of its granularity. Code is a software model that emphasizes on the detailed granularity while requirements are another set of software model that describe a system from the user's point of view.

There is a need to relate from one model to another e.g. from a requirement to its implementation code or design to test cases. This is called traceability. Ramesh relates traceability as the ability to trace the dependent items within a model and the ability to trace the corresponding items in other models [3]. Pursuant to this, Turner and Munro [4] assume that a software traceability implies that all models of the system are consistently updated. Tracing such kind of traceability is called *requirements traceability* [3].

Many software engineering standards governing the development of systems (e.g. IEEE/EIA 12207) emphasize the need of requirements traceability to be included into documentation and treat it as a quality factor towards achieving process improvement (CMM)[5]. The issue is that not much elaboration on what types of information needed and how a strategy to achieve this are described in the development standards and guidelines. The developers normally prepare some traceability relationships between software components at the high level abstracts and take least effort to integrate both the high level and the low level software models e.g. a requirement to its implementation code or vice versa. The ability to implement such a requirements traceability would allow a maintainer or management to visualize the impacts prior to actual change. This will greatly help one take appropriate actions with respect to decision making, schedule plans, cost and resource estimates.

In this paper, we present an implementation of requirements traceability to support visibility into the system environment that involves four main sets of artifacts, namely requirements, test cases, design and

code. These artifacts are made available from the documentation. Our approach of requirements traceability provides the ability to not only relate a requirement to its implementation code but also to any other artifact levels. It does provide ability for top-down and bottom-up tracing between artifacts.

This paper is organized as follows: Section 2 presents an overview of our traceability model. Section 3 discusses our mechanism to manage a requirements traceability followed by the total traceability approach in section 4. Section 5 presents a case study with some results and lessons learnt. Section 6 presents some related works. Lastly, section 7 gives a conclusion and future work.

2. A Traceability Model

Figure 1 reflects the notion of our model to establish the relationships between artifacts. The thick arrows represent the direct relationships while the thin arrows represent the indirect relationships. Both direct and indirect relationships can be derived from the static or dynamic analysis of component relationships. Direct relationships apply actual values of two components, while indirect relationships apply intermediate values of relationship e.g. using a transitive closure.

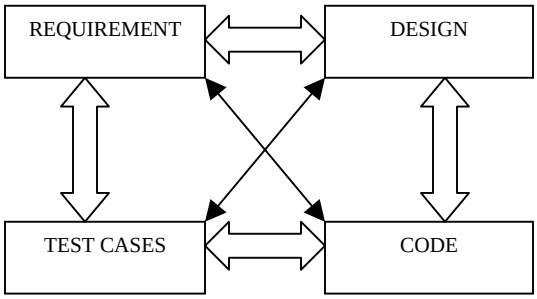

Figure 1 : Meta-model of traceability system

Static relationships are software traces between components resulting from a study of static analysis on the source code and other related models. Dynamic analysis on the other hand, results from the execution of software to find traces such as executing test cases to find the impacted codes. We classify our model into two categories; vertical and horizontal traceability. Vertical traceability refers to the association of dependent items within a model and horizontal traceability refers to the association of corresponding items between different models [6].

2.1 Horizontal Traceability

We regard horizontal traceability as a traceability model of inter-artifacts such that each component (we call it as an artifact) in one level provides links to other components of different levels. Figure 2 shows a traceability from the point of view of requirements. For example, R1 is a requirement that has direct impacts on test cases T1 and T2. R1 also has direct impacts on the design D1, D2, D3 and on the code component C1, C3, C4. Meanwhile T1 has its own direct impact on D1 and D1 on C4, C6, etc which reflect the indirect impacts to R1. The same principle also applies to R2. R1 and R2 might have an impact on the same artifacts e.g. on T2, D3, C4, etc. Thus, the system impact can be interpreted as follows.

$S = (G, E)$

$G = GR \cup GD \cup GC \cup GT$

$E = ER \cup ED \cup EC \cup ET$

Where,

S - represents a total impact in the system

G - represents an artifact of type requirements (GR), design (GD), code (GC) or test cases (GT).

E - represents the relationships between artifacts from the point of view of an artifact of interest. This is identified by ER, ED, EC and ET.

Each level of horizontal relationship can be derived in the following perspectives.

i) Requirement Traceability

$ER \subseteq GR \times SGR$

$SGR = GD \cup GC \cup GT$

A requirement component relationship (ER) is defined as a relationship between requirement (GR) with other artifacts (SGR) of different levels.

ii) Design Traceability

$ED \subseteq GD \times SGD$

$SGD = GR \cup GC \cup GT$

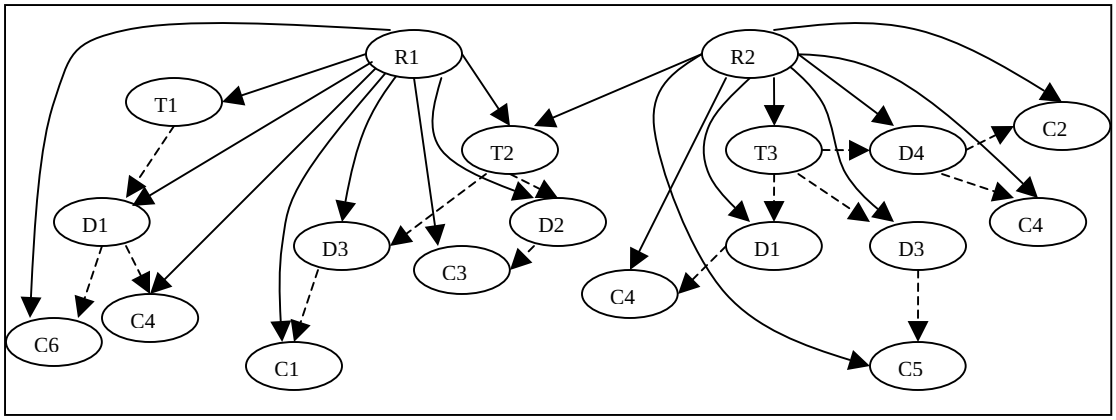

Figure 2: Traceability from the requirement perspective

A design component relationship (ED) is defined as a relationship between a design component (GD) with other artifacts (SGD) of different levels. GD can be decomposed into more detailed design components, if necessary.

iii) Test case Traceability

$ET \subseteq GT \times SGT$

$SGT = GR \cup GD \cup GC$

A test case component relationship (ET) is defined as a relationship between a test case (GT) with other artifacts (SGT) of different levels.

iv) Code Traceability

$EC \subseteq GC \times SGC$

$SGC = GR \cup GD \cup GT$

A code component relationship (EC) is defined as a relationship between a code component (GC) with other artifacts (SGC) of different levels. Code can be decomposed into more detailed components.

2.2 Vertical Traceability

We regard a vertical traceability model for intra-artifacts of which an artifact provides links to other components within the same level of artifacts. In principle, we consider the following as our vertical platforms.

a) Requirement level
b) Test case level
c) Design level
d) Code level

Requirement level here refers to the functional requirements. While the test case level refers to the test descriptions that describes all possible situations that need to be tested to fulfill a requirement. In some systems, there might exist some requirements or test cases being further decomposed into their sub components. However, to comply with our model, each is uniquely identified. To illustrate this phenomenon, let us consider the following example.

Req#: 5
Code : SRS_REQ-02-05
Description: The driver presses an "Activation" button to activate the AutoCruise function.
The test cases involved :

1) Test case #: 1
   Code: TCASE-12-01
   Description : Launch the Auto Cruise with speed is > 80 km/hr.
      i) Test case#: 1.1
      Code : TCASE-12-01-01
      Description: Launch the Auto Cruise while not on fifth gear.
      ii) Test case#: 1.2
      Code : TCASE-12-01-02
      Description: Launch the Auto Cruise while on fifth gear.
2) Test case#: 2
   Code : TCASE-12-02
   Description: Display the LED with a warning message "In Danger" while on auto cruise if the speed is >= 150 km/h.

We can say that *Req#5* requires three test cases instead of two as we need to split the group of test *case#1* into its individual test *case#1.1* and test *case#1.2*. In the design and code, again there might exist some ambiguities of what artifacts should be represented as both may consist of some overlapping components e.g. should the classes be classified in the design or code ? To us, this is just a matter of development choice.

Design level can be classified into high level design abstracts (e.g. collaboration design models) and low level design abstracts (e.g. class diagrams) or a combination of both. In our implementation, we pay less attention on high level design abstracts to derive a traceability as this needs more research and would complicate our works. We apply the low level design abstracts that contain the software packages and classes with their interactions. While, the code is to include all the methods and their interactions.

3. Approach

3.1 Hypothesize Traces

We believe that there exists some relationships among the software artifacts in a system. We need to trace and capture their relationships somehow not only within the same level but also across different levels of artifacts before a impact analysis can be implemented. The process of tracing and capturing these artifacts is called hypothesizing traces. Hypothesized traces can often be elicited from system documentation or corresponding models. It is not important in our approach whether the hypotheses should be performed by manually through the available documentations and software models or by automatically with the help of a tool. Figure 3 reflects one way of hypothesizing traces. It can be explained in the following steps.

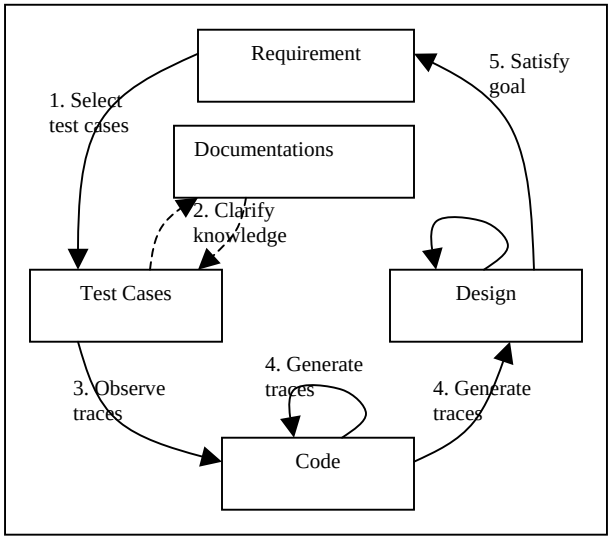

Figure 3 : Hypothesized and observed traces

1. For each requirement, identify some selected test cases (RxT).
2. Clarify this knowledge with the available documentation, if necessary.
3. Run a test scenario (dynamic analysis) for each test case based on the available test descriptions and procedures, and capture the potential effects in terms of the methods involved (TxM). We developed a tool support, called *CodeMentor* to identify the impacted code by instrumenting the source code prior to its execution [7].
4. Perform a static analysis on the code to capture the call relationships of method-to-method (MxM) and class-to-class (CxC) dependencies.

We experimented using tool supports such as *McCabe* [8] and *Code Surfer* [9] to help capture the above program dependencies. However, other manual works as well as the need for other types of information saw us developing our own code parser called *TokenAnalyzer* [10].

3.2 Traceability Approach

Intrinsically, traceability provides a platform for impact analysis. We can classify three techniques of traceability.

1. Traceability via *explicit links*
   Explicit links provide a technical means of explicit traceability e.g. traceability associated with the basic inter-class relationships in a class diagram modelled using UML [11].
2. Traceability via *name tracing*
   Name tracing assumes a consistent naming strategy and is used when building models. It is performed by searching items with names similar to the ones in the starting model [12].
3. Traceability via *domain knowledge* and *concept location.*
   Domain knowledge and *concept location* are normally used by experienced software developer tracing concepts using his knowledge about how different items are interrelated [13].

We apply 1) and 3) in our traceability approach. We obtain the *explicit links* of component relationships from the hypothesized traces and establish a set of matrices to implement the traceability between components in the system. We use *concept location* to establish links between requirements and test cases with the implementation code. This process requires a maintainer to understand the *domain knowledge* of the system he wants to modify. With this prior knowledge of a requirement, a maintainer should be able to decompose it into more explicit items in terms of classes, methods or variables. These explicit items represent a requirement or a concept that are more traceable in the code [13]. With the help of test cases in hand, our approach via *codeMentor* should be able to support a maintainer tracing and locating the ripple-effects of the defined items in terms of the impacted methods and classes.

Name tracing is another technique for implementing traceability. It can be used to locate the corresponding items of a model with another model e.g. to locate the occurrences of an item of similar name in a requirement with the ones that exist in the implementation code in an effort to establish some links between requirements and code. However, this strategy is not practical in our context of study. The reason is that *name tracing* cannot be used to search for structural relationships of program dependencies.

4. Total traceability

Figure 4 describes the implementation of our total traceability. The horizontal relationships can occur at the cross boundaries as shown by the thin solid arrows. The crossed boundary relationship for the requirements-test cases is shown by RxT, test case-code by TxM, and so forth. The vertical relationships can occur at the code level (MxM- method interactions) and design level (CxC- class interactions, PxP- package interactions) respectively.

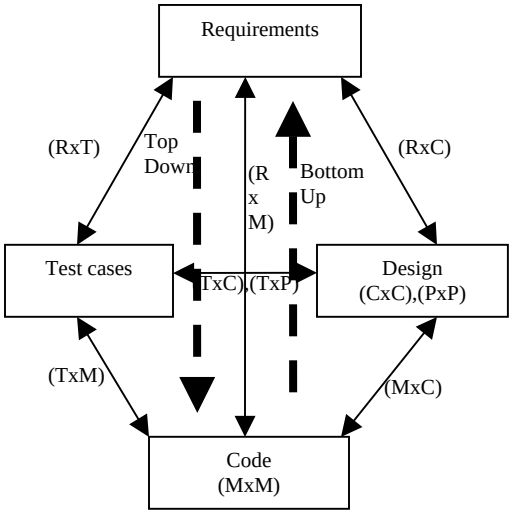

Figure 4 : System artifacts and their links

The method interactions can simply be transformed into class interactions and package interactions by the use of mapping mechanism based on the fact that a package is made up of one or more classes and a class is made up of one or more methods. The thick doted lines represent the total traceability we need to implement in either top down or bottom up tracing. By top-down tracing, we mean we can identify the traceability from the higher level artifacts down to its lower levels e.g. from a test case we can identify its associated implementation code. For bottom-up tracing, it allows us to identify the impacted artifacts from a lower to a higher level of artifacts e.g. from a method we can identify its impacted test cases and requirements.

The table 1 represents an example of mapping table of our system to support traceability. Each method (i.e. member function in C++) is assigned an ID and a method name. CSU column represents the class numbers and CSC column represents the package numbers. It can be interpreted as e.g. *driving StationHandler()* is a method and the only method of class#1 that resides in a package#1. *getState()* is a method of class#2 and still belongs to package#1, etc.

ID	Method_Name	LOC	VG	CSU	CSC
1	drivingStationHandler()	37	12	1	1
2	getState()	16	4	2	1
3	isOn()	10	2	3	1
4	isHighest()	6	2	4	1
5	keypadHandler()	48	23	5	2
6	CInitManager::CInitManage	2	1	6	3
7	CInitManager::init()	10	2	6	3
8	CInitManager::getSignal()	18	1	6	3
9	main()	10	1	6	3
10	CFcdCruise::CFcdCruise()	4	1	7	4
11	doCruise()	3	1	7	4
12	doCruiseStatus()	3	1	7	4
13	CCruiseManager::CCruisel	7	1	8	4
14	getOperationSignal()	49	15	8	4
15	regulateSpeed()	62	18	8	4
16	checkCruiseMode()	15	16	8	4
17	CCruiseInfo::CCruiseInfo()	12	1	9	4
18	isActive()	3	1	9	4
19	setCruiseSpeed()	3	1	9	4
20	setCruiseStatus()	38	12	9	4
21	getPrevSpeed()	3	1	9	4

Table 1: A mapping table

We assign each method with some LOC (lines of code) and VG (value of program complexity) of which these metrics are made available before hand with the help of a code parser, e.g. *McCabe* tool. We need to capture these metrics for each class and use classes to compute the LOC and VG for packages. We need to store the similar mapping scheme for classes and packages but in separate tables. This information is required as the system goes along the way tracing for potential effects, it can collect and sum up the metrics of the impacted artifacts.

4.1 High Level Traceability

We relate the high level traceability as a tracing involving the requirements and test cases. This tracing is based on dynamic analysis of program execution via test scenarios. In our case study, each requirement is characterized by some test cases as explained in the previous section 2.2. As we had the RxT and TxM from the hypothesized traces earlier, we can produce the RxM using a transitive closure.

(RxT) and (TxM) → (RxM)

We use the underlying RxM to create a RxC and RxP by upgrading the methods into classes and classes into packages with the help of the same mapping mechanism. The same principle applies to bottom up tracing but now we use the underlying MxR (i.e. the inverted RxM table) and transforms it into CxR and PxR. The same principle applies to the test cases with their low level artifacts.

4.2 Low Level Traceability

We classify the low level traceability as a tracing that can be established around the code and design levels. This includes the methods, classes and packages. The impacts can be established between components based on dependence graphs of MxM such that an impact to a method implies an impact to its class and an impact to a class implies an impact to its package it belongs to. This dependency makes it possible to transform the MxM into MxC, CxM, CxC, CxP, PxC and PxP. At the low traceability level, the vertical links can occur between a) method-method, b) class-class, and c) package-package. While, the horizontal links can occur across boundaries between a) method-class, b) method-package, c) class-package.

5. Case Study: OBA

To implement our model, we applied it to a case study of software project, called the Automobile Board Auto Cruise (OBA). OBA is an embedded software system of 3k LOC with 480 pages of documentation developed by the M.Sc group-based students of computer science at the Centre For Advanced Software Engineering, university of Technology Malaysia. OBA was built as an interface to allow a driver to interact with his car while on auto cruise mode such as accelerating speed, suspending speed, resuming speed, braking a car, mileage maintenance, and changing modes between the auto cruise and non-auto cruise. The project was built with complete project management and documentations adhering to DoD standards, MIL-STD-498[+][14]. The software project was built based on the UML specification and design standards [11] with a software written in C++. The documentations provided us with some useful information on artifact relationships, e.g. we obtained the requirement-test case relationships (RxT) from the SRS (Software Requirement Specification) document, the class-package relationships (CxP) from the SDD (Software Design Description) document and test procedures from the STD (Software Test Description) document.

5.1 Impact Generation

We identified from the OBA project, 46 requirements, 34 test cases, 12 packages, 23 classes and 80 methods. Our system, *Catia* assumes that a user request has already been translated and expressed in terms of the acceptable artifacts i.e. requirements, classes, methods or test cases. *Catia* was designed to manage the potential effect of one type of artifacts at a time.

MIL-STD-498[+] - This standard was formally closed by the US DoD in 1998 and adopted a new standard, the IEEE/EIA 12207 in replacement. Nevertheless, most of the detailed data items of MIL-STD-498 were absorbed into the new standard and remained intact.

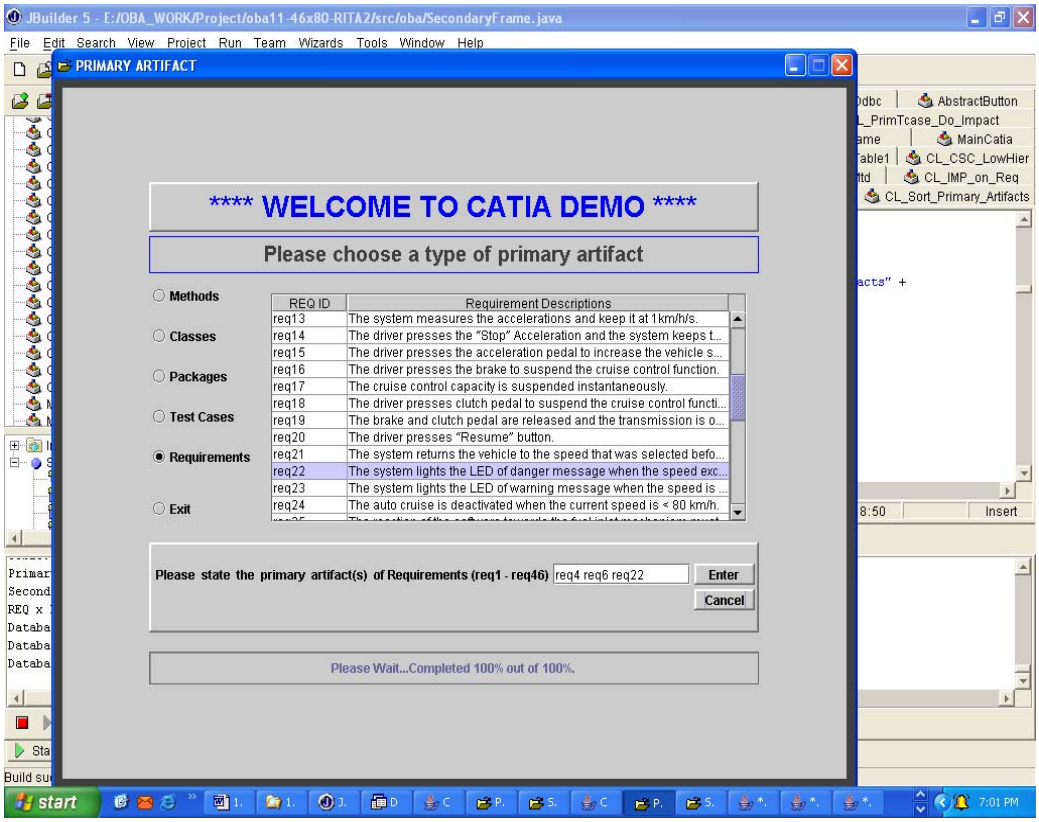

Figure 5 : First user interface of CATIA

The system works such that given an artifact as a primary impact, *Catia* can determine its effects on other artifacts (secondary artifacts) in either top-down or bottom-up tracing. Figure 5 shows an initial user entry into the *Catia* system by selecting a type of primary artifacts followed by a set of the detailed artifacts. The user selected requirements as the primary artifact and chose the req4, req6 and req22 as the detailed requirements of interest.

Figure 6 represents an output of the propagated artifacts and its summary after the user selects one or more types of secondary artifacts. In Figure 5, the user selected all the artifact levels as the secondary artifacts to visualize the impacts. After 'generate button', *Catia* then produced a list of impacted methods, classes, packages, test cases and requirements for each primary artifact chosen earlier. In the summary table (Figure 5), all the impacted artifacts associated to req4, req6 and req22 were shown in terms of counts, LOC and VG. Taking an example of req4, this requirement had caused potential effect to 40 methods out of 80 total methods in the system. In terms of the impact metrics, these methods took up LOC(349) and VG (101) out of total LOC (925) and VG (250) respectively. Please note that no impacts being described on req4, req7 and req12 over other requirements as this situation is not allowed in our model.

5.2 Some Discussions and Lessons Learnt

There are some knowledge and experience we would like to highlight with respect to the implementation of our prototype.

1. DLL files (4 packages)
   DLL files only contain all the executable files as the reusable software packages and no source code available. As this is the case, there is no way for us to neither using the *McCabe* nor *CodeMentor* to capture the methods and classes in the DLL packages. Thus, we treated the DLL files as special packages with no metric values.
2. Self impact
   There were cases in the (MxM) and (CxC) relationships, a component only made an impact on itself not to others. This is due to the fact that a method or class was designed just to provide a service rather than call invocation to others.
3. Non functional requirements (1 requirement)
   There was a timing requirement, STD_REQ-02-19 stated that "fuel inlet mechanism should respond in less than 0.5 seconds on actions by a driver". This requirement had no impact on other classes or methods. This is due to the fact that the timer is produced by the kernel operating system not by any other classes or methods. The result of timing may be needed by some classes or methods for some tasks e.g. in speed calculation, but no action being carried out to check the violation of timing. The developers verified this requirement manually by running a test

driver to spy the timing at the background mode. As no program verification can be made on this particular issue, we dropped this type of requirement from our work.

service functions and data objects. The tool, called GEOS is constructed on the basis of a relational database and populated with meta-model on concept. Hypertext techniques and ripple effects are used to identify the

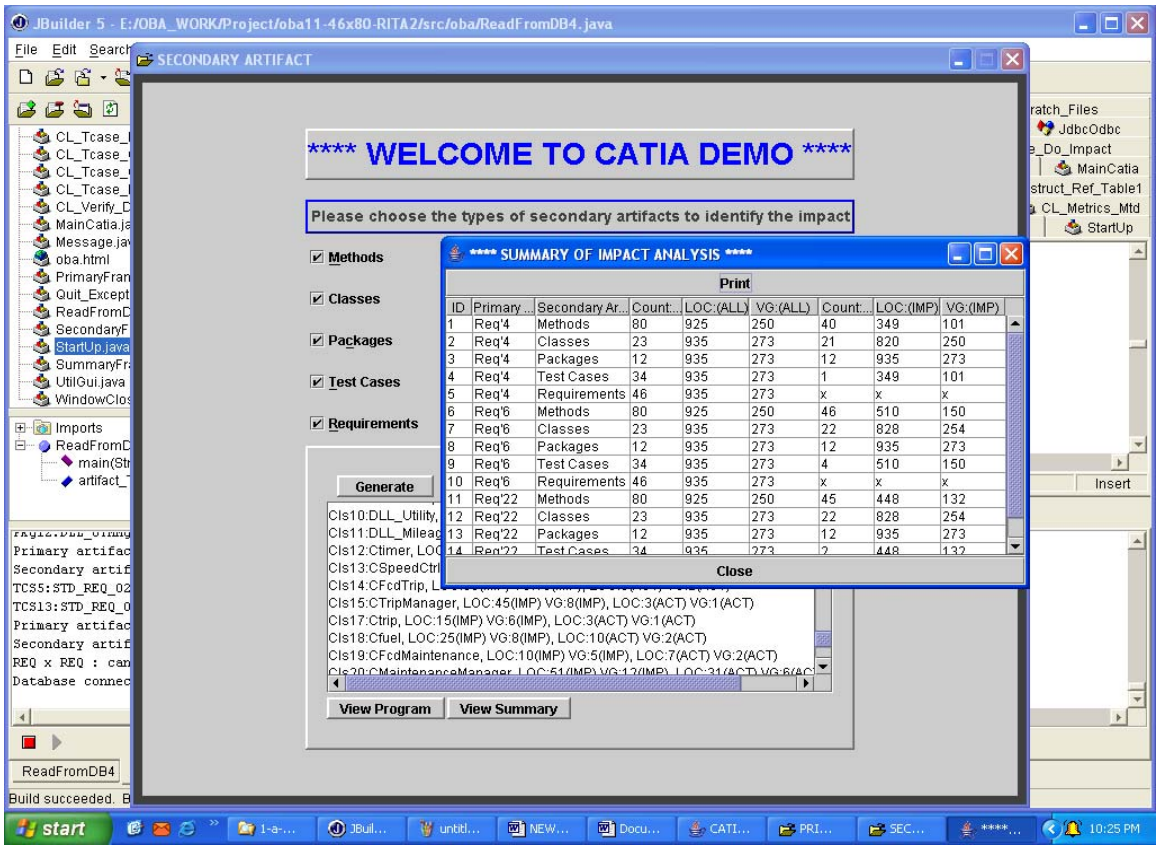

Figure 6 : Output of requirements traceability

6. Related Work

A number of requirement traceability approaches and tools being cited in both the literature and as by-products in the industry. Some other advanced tools such as Teamwork/RQT [15], RTM [16], and TOORS [17] provides the capabilities to include mechanisms to create parent and child relationships, functional hierarchies, definition of keywords and attributes to requirements and other system artifacts, ad-hoc and predefined querying, requirements extraction from documents, customized report generation, and maintenance of information about allocation of requirements to system components or functions.

Lindvall and Sandahl [12] present a traceability approach based on domain knowledge to collect and analyse software change metrics related to impact analysis for resource estimates. However, their work do not consider automated *concept location*. They relate some change requests to the impacted code in terms of classes but no requirements and test cases involved. Sneed [18] constructs a repository to handle maintenance tasks that link the code to testing and concept models. His concept model seems to be too generalized that includes the requirements, business rules, reports, use cases, software interdependencies. Bianchi *et al.* [19] introduce and experiment with several examples of traceability link using ANALYST tool, with the aim of assessing how effectively these links support impact analysis in object-oriented environments and what effects they produce on the accuracy of the maintenance process. Both [18,19] works provide some good framework at coarse levels but they do not associate with test cases and user requirements.

Our work differs from the above in that we attempt to integrate the software components that include the requirements, test cases, design and code. Our model and approach allow a component at one level to directly link to other components of any levels. Another significant achievement can be seen in its ability to support top down and bottom up tracing from a component perspective. This allows a maintainer to identify all the potential effects before a decision can be made.

7. Conclusion and Future Work

Our current approach offers considerable leverage in implementing a software visibility based on call invocations. The traceability part of the documentation is

typically prepared manually by the software engineers and very few attempt to establish links to the implementation code. The crucial part of the traceability is to establish and integrate the high level software models with their implementation code that many software maintenance tools tend to ignore.

This effort may need a close cooperation between the project manager, designers, testers and developers as it involves different responsibilities and software models. However, having such an automated requirements traceability would not only benefit both high-level and low-level users of the software development and maintenance, but also to support regression testing.

Our next attempt is to gear towards addressing the change impact analysis, an important issue in software maintenance. Change impact analysis require special attention on its implicit, explicit links and some design decisions as being explored by some researchers [12,18,19]. For example, in the call invocation relationships,

M1 → M2
→ M4

M1 calls two other methods; M2 and M4. This means any change made to M2 or M4 would have a potential impact on M1. So, in the context of change impact analysis, we have to work on the other way around by picking up a callee and find its corresponding callers. In another example, if class A is inherited from class B, then any change made in class B may affect class A and all its lower subclasses, but not to its upper classes. We need to consider all other structural relationships such as friendship, composition and aggregation and object creation on account of the potential impacts.

Acknowledgements

This research is funded by the IRPA of Malaysian Plan (RM-8) under vot no. 74075. The authors would like to thank the Universiti Teknologi Malaysia, the University of Durham, the Universiti Kebangsaan Malaysia and individuals for their involvement, invaluable comments and suggestions throughout the development and review process.

References

[1] E. Tryggeseth, Report from an Experiment: Impact of Documentation on Maintenance, *Journal of Empirical Software Engineering,* Kluwer Publishing, 1997, vol. 2(2), pp. 201-207.

[2] A. Bianchi, A.R. Fasolino, G. Visaggio, An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models, *IWPC*, 2000, pp. 149-158.

[3] B. Ramesh, Requirements traceability: Theory and Practice, *Annuals of Software Engineering*, vol. 3, 1997, pp. 397-415.

[4] R.J. Turver, M. Munro, An Early impact analysis technique for software maintenance, *Journal of Software Maintenance: Research and Practice*, Vol. 6 (1), 1994, pp. 35-52.

[5] B. Ramesh, M. Jarke, Toward Reference Models for Requirements Traceability, *IEEE Transactions on Software Engineering*, 27(1), January 2001, pp. 58-93.

[6] O. Gotel, A. Finkelstein, An Analysis of the Requirements Traceability Problem, *in Proceedings of the First International Conference on Requirements Engineering*, Colorado, 1994, pp. 94-101.

[7] S. Ibrahim, N.B. Idris, A. Deraman, Case study: Reconnaissance techniques to support feature location using RECON2, *Asia-Pacific Software Engineering Conference, IEEE*, Dec 2003, pp. 371-378.

[8] http://www.mccabe.com

[9] http://www.gramatech.com/products/codesurfer/index.html

[10] S. Ibrahim, R.N. Mohamad, "Code Parser for C++", *Technical report of Software Engineering*, CASE/August 2004/LT2, August, 2004.

[11] G. Booch, I. Jacobson, J. Rumbaugh, *UML Distilled Applying the Standard Object Modeling Language* (Addison-Wesley, 1997).

[12] M. Lindvall and K. Sandahl, Traceability Aspects of Impacts Analysis in Object-Oriented System, *Journal of Software Maintenance Research And Practice*, vol. 10, 1998, pp. 37-57.

[13] V. Rajlich, N. Wilde, The Role of Concepts in Program Comprehension, *Proceedings of 10th International Workshop on Program Comprehension, IEEE*, 2002, pp. 271-278.

[14] Joint Logistics Commanders on Computer Resource Management, *Overview and Tailoring Guidebook on MIL-STD-498* (Arlington, 1996).

[15] McCausland C.D., A Case Study in Traceability, *Proceedings of the Colloquium by the Institution of Electronic Engineers Professional Group Cl (Software Engineering)*, London, 1991.

[16] Marconi Systems Technology, RTM Requirements and Traceability Management, Arlington, VA. 1991.

[17] F.A.C. Pinheiro and J. Goguen, An Object-Oriented Tool for Tracing Requirements, *IEEE Software*, 1996, pp. 52-64.

[18] H.M. Sneed, Impact Analysis of maintenance tasks for a distributed object-oriented system, *Proceedings of Software Maintenance, IEEE*, 2001, pp. 180-189.

[19] A. Bianchi, A.R. Fasolino, G. Visaggio, An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models, *IWPC*, 2000, pp. 149-158.