Management of Test Cases Using Database Concepts

Mats Grindal

International Conference On
Software Testing, Analysis & Review
November 19 - 23 Stockholm, Sweden

Presentation

T19

Thursday 22nd November, 2001

Thursday 22 November 2001
T19

Management of Test Cases Using Database Concepts

Mats Grindal

After receiving his Master's degree in Computer Science from the Royal Institute of Engineering (KTH) in Stockholm, Mr Grindal has over ten years of experience working with software testing. Most of this time he has spent in the industry, working as a consultant for Enea Realtime AB. Recent assignments include mentoring, teaching, and test process improvement work. Since 2000, Mr. Grindal holds a part time research position at the University of Skövde. His research focus on optimizing test case selection for distributed real-time systems.
Mr Grindal's professional interests include all test-related questions, with a bias towards problem solving in the early phases in the test process.

Managing Test Cases Using Database Concepts

Mats Grindal

Enea Realtime AB
www.enea.se
Box 232, S- 183 23 Täby, Sweden
Tel: +46-8-50 714 000
Fax: +46-8-50 714 040
E-mail: magr@enea.se

Ingvar Nordström

Ericsson Utvecklings AB
www.ericsson.com
Box 1505, S-125 25 Älvsjö, Sweden
Tel: +46-8-727 35 38
Tel: +46-8-727 40 78
E-mail: Ingvar.Nordstrom@uab.ericsson.se

Abstract

Using traditional ways of handling test cases often hampers maintenance and re-use. One main reason is that test cases are produced with a project scope, while the contents in the test cases should evolve with the requirements on the product instead of a single project.

In this work we describe a database model for test case storage, which supports effective re-use and management of test cases and test scripts. Integrated in the model is also support for status tracking of product quality and test-project progress. One of the benefits with using this database organization is the modularization of test cases and test scripts, which allows a mix of storage and execution strategies to co-exist. We also describe an inexpensive and simple method to evaluate most of the benefits of the database model. To use this method, an operating system supporting links and version handling are the only requirements. Finally we describe some experiences from a custom-designed tool, with a full implementation of the database model, and compare it with some commercially available tools.

Our main conclusion is that applying a database strategy to the management of test cases is beneficial to both current and future projects. Recognizing that some test artifacts are product related while other test artifacts are project related, improves the structure of the underlying database model, as well as simplifying maintenance and re-use of the test artifacts.

1 Introduction

Lots of resources in a system development project are devoted to testing [Bei90]. The five phases of a testing project, defined in the British Standards Institution (BSI) standard BS7925-2 [BS98b], are planning, specification, execution, recording, and check for completion. Undoubtedly, specification and execution are the two most time consuming activities. Thus, test process improvement attempts targeted towards these activities should have the largest potential. Traditionally, focus has been on improving the execution phase. Mostly this is due to the interest in automation of [parts of] the test execution through the use of various test execution tools. The abundance of commercial test execution tools is one piece of evidence of this. Methods of improving the specification phase have not been explored to the same extent. General attempts to improve the methods of this phase have focused more on test quality than on efficiency. One example of this is the vast amount of test case selection methods, which increase the objectivity of the testing, thus increasing the test quality. Efforts on increasing the efficiency in the specification phase has mostly been focused on automatic derivation of test cases but these efforts have only reached limited success. There are at least two major difficulties. First, for a tool to be able to automatically generate test cases, requirements need to be expressed in some formal or semi-formal form. Second, even if test cases can be automatically generated, there is still the oracle problem, i.e., the problem of foreseeing the correct expected behavior.

In this paper we take another approach for making test case specification more efficient. We organize test cases in a database, which will enable easy re-use and maintenance of test cases while providing traceability and easy progress reporting of project status. Thus, the focus of this work is neither on automatic generation of test cases, nor the automatic execution of test cases, although nothing in this work contradicts these things from also being focused on.

A well-known feature of automation in general, and automation of test case execution, in particular is that the investment will not start to pay-off until in the second or third iteration. This is also true for the ideas presented in this work. Substantial time-savings, by applying these ideas will not materialize until the time a test case can be re-used. However, except for the initial cost of setting up the database in the first place, populating the database with new test cases will not cost more than the cost of writing the test cases the ordinary way. Consequently a test case database approach can be easily motivated. A positive spin-off effect of organizing test cases in databases is that maintenance testing may also benefit from this by easy access to the correct versions of the test cases for a particular version of a product.

The introduction of a test case database is no different from the introduction of any other test tool. In his report of test automation in Swedish industry, Bereza [BJ00] stresses that automating chaos will only faster lead to chaos. Thus, we draw the conclusion, that in order for the introduction of a test case database to be successful, a process for test case development should already be in place.

This paper is organized as follows. Section 2 contains some basic definitions and assumptions essential for this work. In section 3, we define and motivate a number of fundamental requirements of a test case database. In section **Error! Reference source not found.**, we gradually build up a logical database model, while showing how the requirements from the previous section can be incorporated. Section 5 describes how the model, with some restrictions can be implemented only using an ordinary file system, which supports links and some type of revision handling. Section 6 briefly describes a custom-designed full implementation of the database model, and in section 7 some related work is mentioned. Finally, section 8 contains some conclusions and final remarks.

2 Definitions and Assumptions

Within the testing community numerous attempts have been made to define terms and concepts. To simplify the reasoning in this paper, we have decided to follow the British Standards Institution (BSI) BS7925-1standard glossary of terms used in software testing [BS98a] wherever nothing else is specified. However, since this work is based on an actual test case database implementation, we decided to maintain some of the terms and concepts from this development. To be as clear as possible, we will supply definitions and explanations of acronyms more often than usual.

In the remainder of this work, a *TCS* (*Test Case Specification)* is a description of what to test. Although a TCS is usually expressed in natural language, the applicability of our ideas is not limited to this, but will also work for test cases expressed more formally. The minimum requirements of a TCS are to contain *preconditions*, i.e., what must hold before the test case can be executed, *actions* taken during test case execution, and the *expected result* of the test case [Bei90]. Other common requirements of a TCS are to have a unique identifier, a heading that describes the important properties of the test case, and a purpose for the test case, for instance a reference to a requirement, etc. In this work we have not assumed fixed contents of a test case. Instead we will demonstrate how to tailor the database to fit the existing test case contents.

As we have assumed that a TCS only describes what to test, we also need to describe how to perform the test. We call this description a *TCI (Test Case Instruction)*. A TCI either contains a sequence of steps to perform in the case of manual test execution, or a program/script in the case of automatic test case execution. We do not make any assumptions of the format of the test instruction, although the presented ideas aim towards automatic testing. If only manual test execution is used, the database model can be somewhat simplified, which will be indicated towards the end of this paper.

Sometimes there is an argument for combining the information in the TCS and the TCI. The main motivation for that is that it is easy to forget to update multiple sources, should a change be necessary. We

acknowledge this problem, and suggest a reminder functionality be built into the database interface to aid the maintenance of the TCSs and TCIs in the database. The main reason for keeping TCSs and TCIs as separate entities is that a one-to-one correspondence between these two entities does not necessarily exist. Consider a TCS in which the maximum number of concurrent calls through a telephone switch should be tested, and suppose that there exists several configurations of the switch with different capabilities. Thus, there might be a need for multiple TCIs for the same TCS. A document containing a collection of TCSs is called *TS (Test Specification)* and the corresponding document for TCIs is called *TI* (*Test Instruction).*

The terms *testware* and *test artifacts* will be used interchangeably in this work to denote all the different products, e.g., test cases, test results, and test plans, that are produced by a test project.

3 Requirements on a test case database

The following sections will summarize and motivate some important requirements on a test case database. Each stated requirement has an associated tag that will be used in later sections as cross-references to illustrate how this particular requirement is satisfied in the final database implementation. Certainly there will be many more requirements on a real implementation than can be expressed below. The reason for selecting this particular subset of requirements is that they help illustrate the main design issues of the test case database.

3.1 Product and Project Artifacts

One of the most fundamental parts of this work is the realizing that parts of the testware are product related and other parts are project related. This is important since the product related testware should live and develop during several projects while project related testware is important in a particular project, but once the project is closed, this part of the testware is of less importance. To illustrate this problem, consider a test case designed to test a particular requirement. This requirement is implemented in a whole family of products. The TCS of the test case is obviously related to the requirement, and should remain constant as long as the requirement is stable. In other words, no matter which product is currently being tested, the TCS remains the same, and if a new product should be developed with this requirement, it should be perfectly feasible to re-use the TCS. Thus, we draw the conclusion that a TCS is product rather than project related. Now, consider a project in which a test case is executed. The result from executing the test case is either pass or fail. In either case, this is valuable information when a release decision should be taken for the product. However, once the product has been released the result from executing a particular test case is of less value for future products. In particular, there can be no guarantees that the same test case will yield the same results in different projects. Thus, we draw the conclusion that the result of a test case execution is project rather than product related. Hence we have shown that a test case contains both product and project related information.

From the above discussion we can deduce that TCSs should evolve with their associated requirements. However, the set of test cases applicable in a project is normally maintained in a document called TS (Test Specification). This means that the TS is a project related test artifact. Apart from being a means of storage, the primary benefits of a TS, are that it appoints a set of test cases to be executed, it simplifies reviews of test cases, and it enables test execution without access to the test specification in electronic form. All these benefits can be preserved even if a database is used for storing the test cases as long as there is a logical concept of a TS in the database, which enables the printing of a test specification. Thus we require the *possibility to generate a test specification from the database repository (REQ_1A).*
With a similar discussion applied to the test case instruction, it may be deduced that *a TI (Test Instruction) is project related, and should have the possibility to be automatically generated (REQ_1B).* An obvious requirement of the test database is also that *all test artifacts stored in the database should have unique identities (REQ_2).*

3.2 Traceability

Requirement traceability is often used in development projects. Among the benefits are the possibility to make completeness controls of a lower design level, impact analysis of a suggested requirements change and the tracking of faulty products. Requirements traceability is normally handled in some kind of tool, where relations between requirements can be expressed. Many testing techniques derive test cases from

requirements, making test case traceability with respect to requirements an obvious extension to requirement traceability. Thus, *the test case database should include support for traceability between requirements and test cases (REQ_3)*. Requirements are not the only type of information that is desirable to maintain in relation to a test case. Whenever a test case is executed in a project, *There should be possibilities to store a test case result (pass/fail) in context of a project (REQ_4)*. Furthermore, in the case of a failure, *there should be possibilities to store a reference to an error report for a test case in a project context (REQ_5).* During execution it is likely that a log is produced by the test object, thus *a test case should include support for referencing the log(s) where this test case has been part of the execution (REQ_6).*
Finally, as it is impossible to foresee future needs in the area of test case related information, *there should be some general purpose attributes of a test case (REQ_7).*

3.3 Workflow and Search Mechanisms

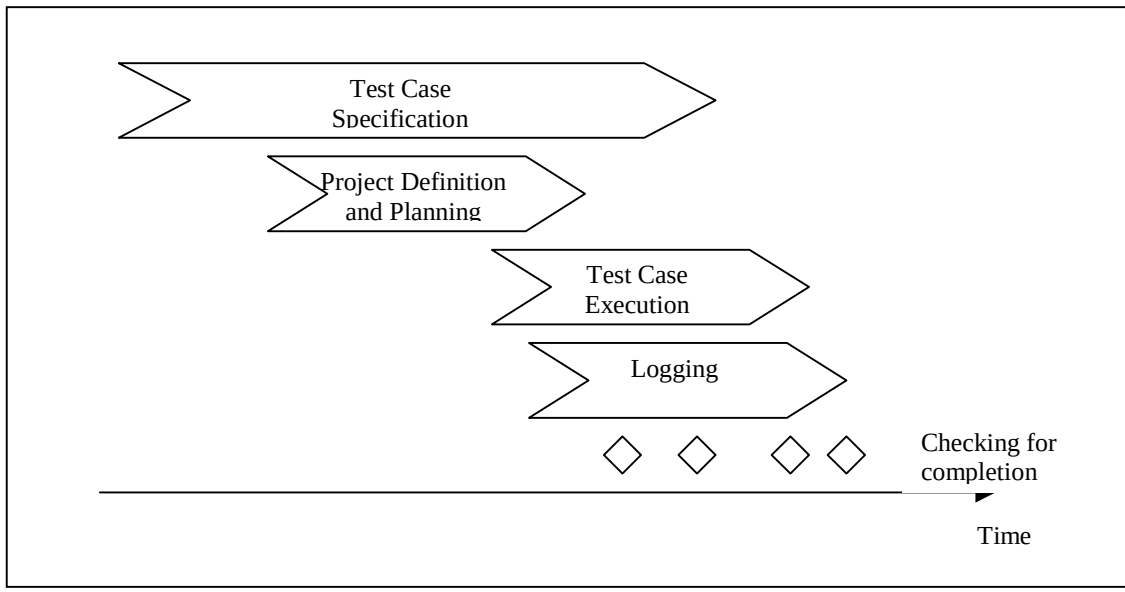


Figure 1 Possible timing relation among the activities of a test process.

Figure 1 shows different activities in the test process from BS 7925-2[BS98b], and a possible relation among them with respect to calendar time. An important issue with this picture is the fact that some test cases are specified in advance of the definition of the project in which they are executed. The obvious example of such a scenario, is the re-use of test cases from earlier projects. Later on in the process, a project is defined. Old existing test cases are associated to this project, and more test cases are specified within the scope of this project. For a test case database to support these ways of working we have a number of requirements that needs to be fulfilled. First of all, *product related artifacts, e.g., TCS and TCI should be able to exist stand-alone in the test case database (REQ_8).* When associating existing test cases to a new project *there needs to be a general search mechanism to locate possible test case candidates (REQ_9). Each project should have its own scope (REQ_10),* and *any product related artifact can be associated to any number of projects (REQ_11).*

3.4 Status Reporting

As can be seen in Figure 1, the checking for test completion can be made more than once. Often the completion criteria of a test project consist of several different requirements, e.g., a certain amount of structural coverage, a certain amount of functional coverage, and some sort of test object quality estimation. A common way of estimating the test object quality is to require a certain number of test cases

to be executed with a passing result. Thus for a test case database to be useful during test completion checking, *it should be possible to find the number of test cases planned and so far executed in a specific project (REQ_12).* Furthermore, *the test case database should provide the means for reporting different aspects of the product status, e.g., pass/fail information (REQ_13).*

3.5 Version handling

As an existing product evolves over time, it is not unlikely that some requirements need to be changed. This will automatically trigger a change of the product related test artifacts, as testing needs to be performed on the updated product with valid test cases. This scenario suggests that the test case database need to support updates of test cases and other test artifacts. However, the situation may be more complex than this. Imagine that while testing the new version of the product with the updated test cases, an urgent problem occurs in an old version of the product at the customer site. This problem requires an immediate correction with complementary regression testing, using the old versions of the test cases. Thus the conclusion is that *the test case database needs to support real version handling of the test artifacts (REQ_14).*

3.6 Properties of test scripts

When testing, it is not uncommon with several test cases only differing in the input values and expected output. For instance, the use of boundary value analysis as test case generation strategy may yield test cases, which only differ in input and expected output. If automatic test case execution is used, via some scripting language, a desired property of a test script language is parameterization. This property will help testers to make test scripts more maintainable since the code of a test script is not copied for each instance of the test case. From this follows that *the test case database should support parameterized test scripts (REQ_15).* Furthermore, as different test cases might contain common parts, procedure calls might be beneficial to the tester, in order to increase the maintainability of the testware. Thus, *the test case database should support procedure calls (REQ_16).*

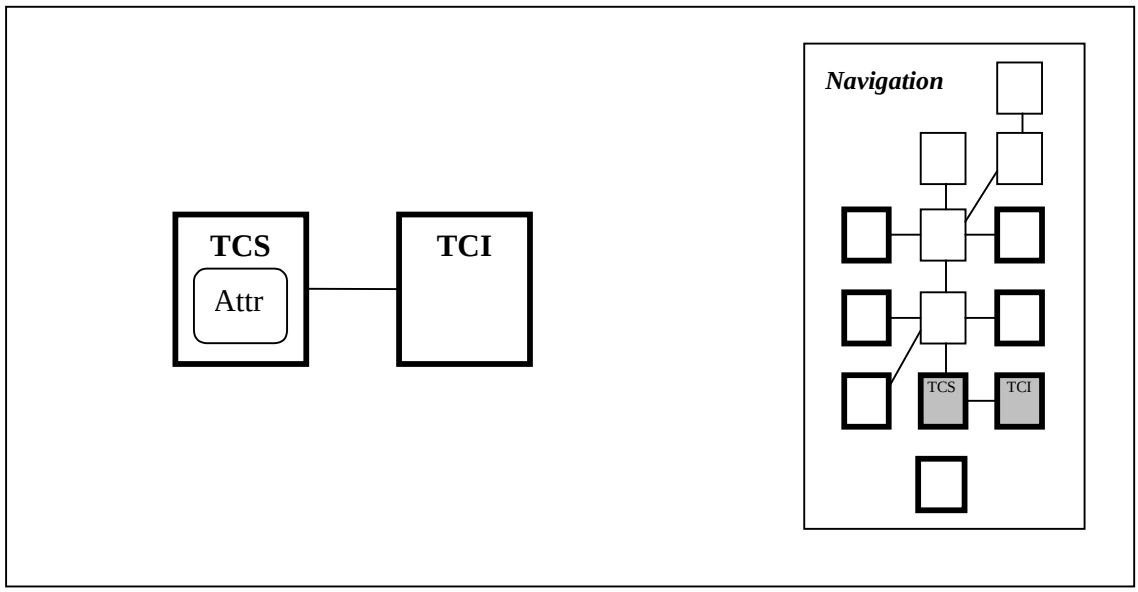


Figure 2 Relation between test case specification and test case instruction entities

Regardless of the possible support for procedure calls in the scripting language, both construction and maintenance of test scripts can benefit from organizing test scripts according to a predefined structure. Such a structure could for instance support the separation of setup actions, actual test actions, and restore actions. One motive for this is that there are likely to be several test cases that require the same setup and

restore actions. Another motive is that the actual test case module often contains comparisons between actual and expected results. Such modules may therefore require a different structure than the setup and restore modules. Thus, we arrive at the conclusion that *the test case database should support a modular test script organization (REQ_17).*

4 Logical Model

This chapter will step by step introduce a logical database model, in a form that resembles an entity-relationship model. As the model is developed, the requirements from the previous section will be incorporated. Whenever a particular requirement is motivated in the database model, this is indicated, to simplify the understanding. The concepts are introduced via figures. In the navigation bar, the concerned parts of the logical database model are displayed by highlighting the corresponding entities. A complete version of the database model, which is the source of the navigation aid in the figures, can be found in appendix A. The bold borders of an entity indicates that this entity contain product related information, and should be version handled.

4.1 Fundamental Concept

The central item of a test case database is obviously the test case. As been described earlier, we favor the separation of the what- and how-parts of a test case. Thus it seems natural to start the modeling of the database by introducing TCS (Test Case Specification) and TCI (Test Case Instruction) entities. Of course there is a natural relation between these two entities, since the TCS contains what to test and the TCI contains how to perform that test. Figure 2 illustrates this relation. In the figure, the attributes of the TCS are highlighted. Among these attributes, there is a possibility to store references to requirements tested by that TCS (REQ_3). Another attribute that might be interesting to store in the TCS is some kind of indication of which feature or functional area the test case belongs too. With the right type of attributes and a general search mechanism over the attributes and the contents of the database items, re-use of existing test cases is greatly simplified (REQ_9). As these entities are explicitly modeled in the database, there is no need to define a project before populating the database with TCS and TCI items (REQ_8).

4.2 Defining a Project

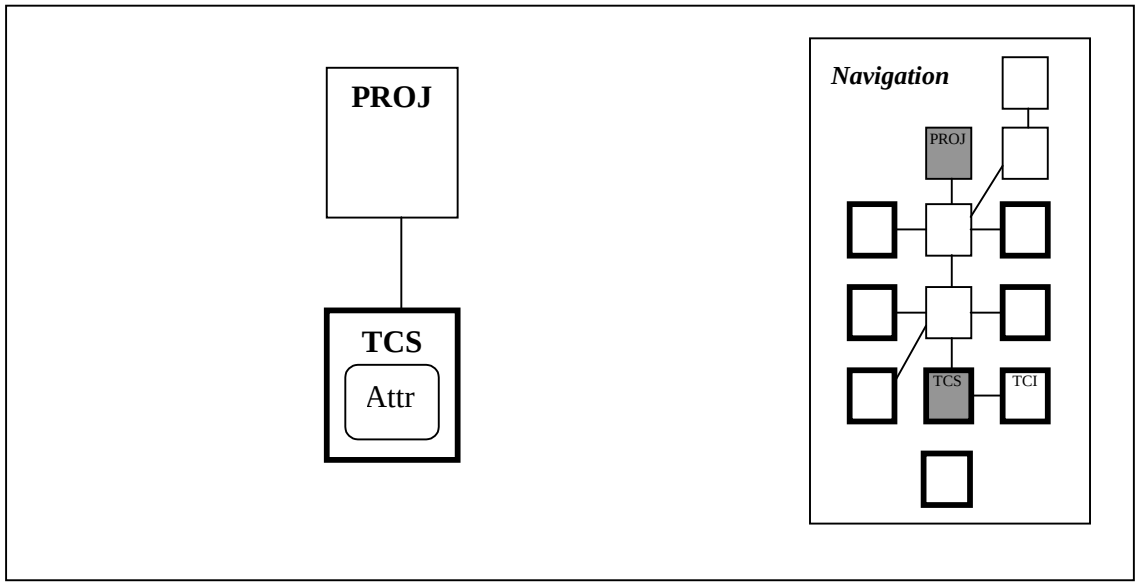


Figure 3 Naïve relation between project and test case specification entities

The next step is to introduce the possibility of association of a test case to a project. The naïve idea is to create a *PROJ (project)* entity and create a relation between the project and the TCS entities, as indicated in Figure 3. However there are a few problems with this solution. First, we might want to organize test cases in different sets, for instance with respect to which feature a particular test case is aimed at testing. This is traditionally accomplished by collecting the test cases in test specifications. In such solution, each test specification contains all test cases of a certain kind, e.g., some functionality, or a regression suite. Second, the same test case might be included in several different projects at the same time, and we want to be able to report test results independently in each project.

By using a database solution, each item of the database will be assigned a unique key (REQ_2) to distinguish it from the other items of the database. The choice of implicit or explicit keys is left to the individual implementation. Some companies may already have unique identities on each test artifact. Then these keys can be re-used. Other companies will need to implement a function to assign these keys.

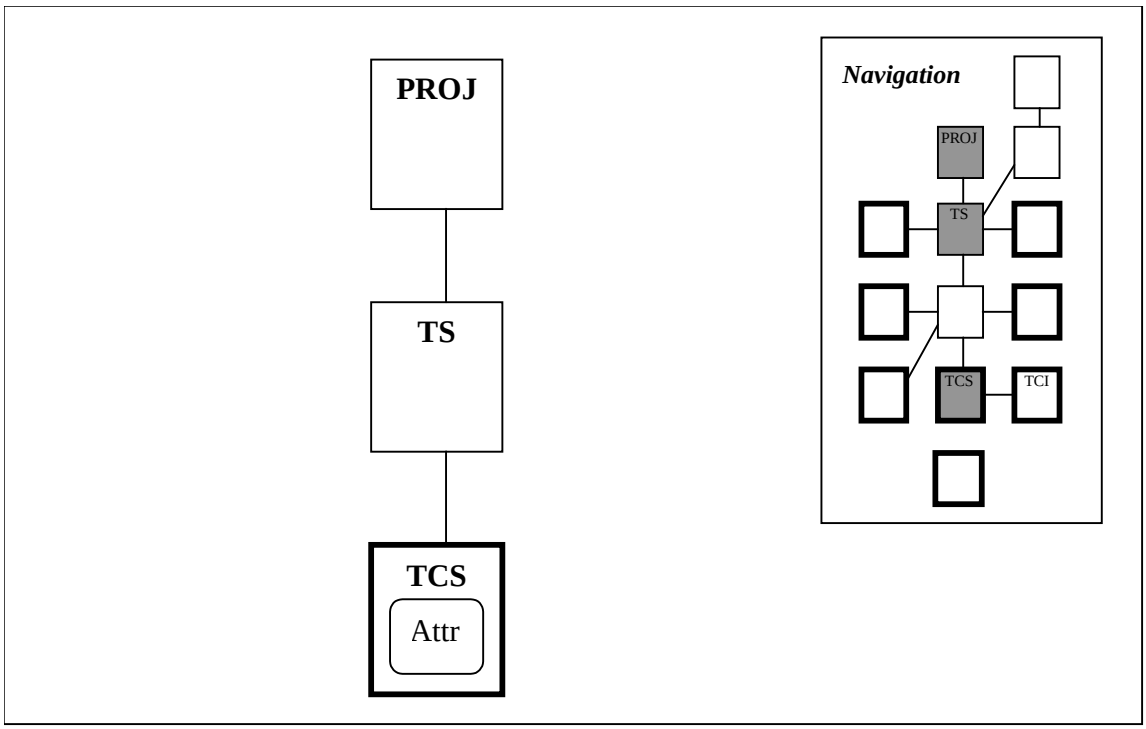


Figure 4 Introduction of test specification (TS) in naïve solution

By introducing a TS (Test Specification) entity between the PROJ and the TCS entities, the problem of supporting different sets of test cases can be solved. This is indicated in Figure 4. The most important contents of a TS entity is a set of references to the test cases included in that TS. This means that the TS is not dependent on the product but only on the project since the TS itself does not contain any product data. Another advantage with this strategy is that it allows arbitrary TS organizations, which may differ between different projects. Furthermore, it even allows several TS items in the same project to (partly) contain the same test cases. Examples when this is desired are when we want to have a separate regression test specification containing a subset of the existing test cases, or when load tests in a separate test specification use some of the test cases defined for functional testing.

Of course it is possible to also include some attributes in the TS entity. Such attributes may for instance be document id, author and revision information. However, an important thing to remember is not to mix product and project related information.

With this solution, where a TS item contains a set of references to test cases and possibly some attributes, it is very easy to automatically generate a paper version of a test specification. By printing the values of the attributes of the particular TS item together with the contents of all the referenced TCS items the TS is complete (REQ_1A). Using the same technique, but following the references from the TCS to the TCI it is also possible to generate a crude version of a test instruction (REQ_1B). In later sections, the contents of the test instruction will be refined, but if manual test execution is dominating, the functionality introduced up to this point is sufficient. However, the introduction of the TS entity, does not solve the problem of results reporting on individual and project specific test cases.

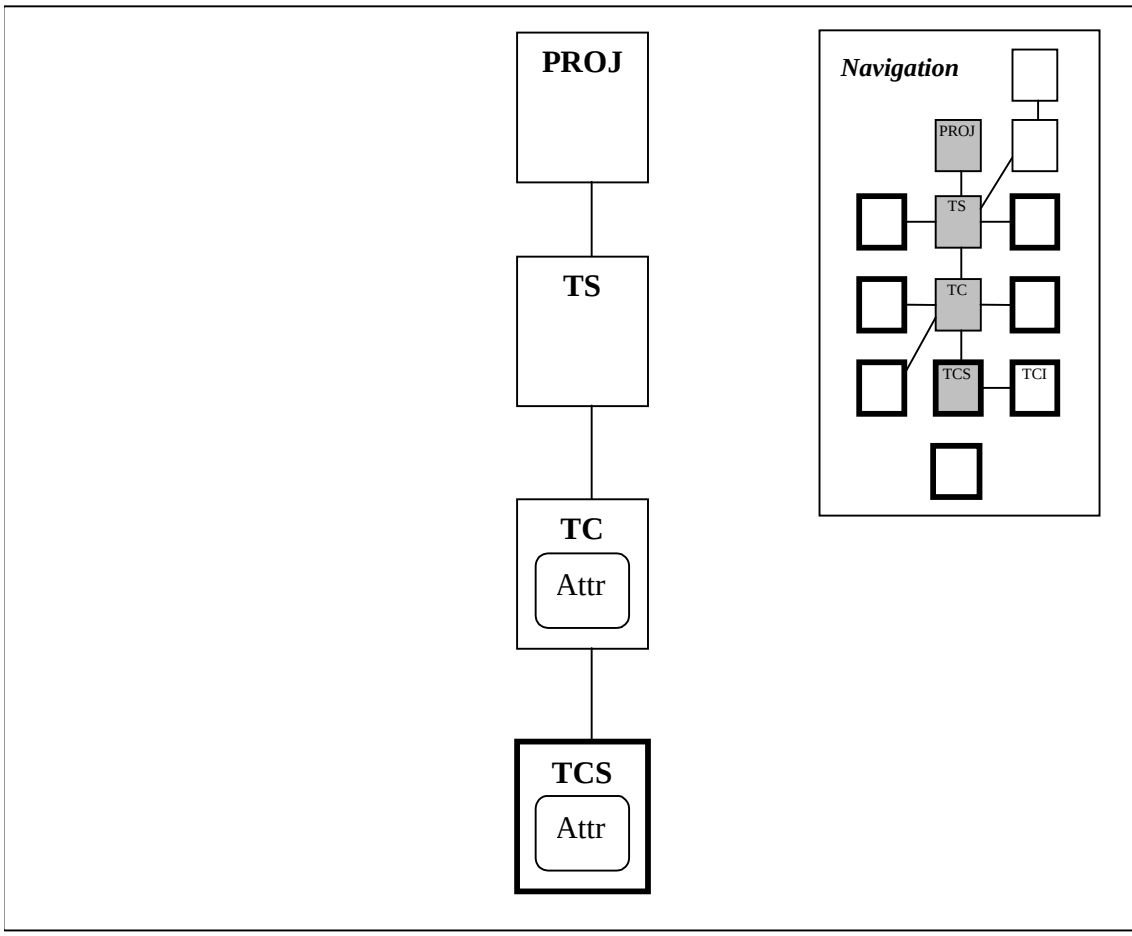


Figure 5 Final solution of relation between project and test case

To solve this problem we need to introduce an extra entity between the test specification and the test case specification. We simply call this entity *TC (Test Case).* The main function of this entity is to keep the project related information of a test case separate from the product related test case information already kept by the TCS entity. This is illustrated in Figure 5. This means that, while test execution results, e.g., pass/fail information (REQ_4) and references to error reports (REQ_5), may be stored as attributes in the TC, requirement references are stored in the TCS, as previously described. Since the TC already contains a number of project specific attributes, this is the perfect place to add some general purpose attributes (REQ_7) to cater for future needs.

The final structure of this part of the test case database model is thus, to have one PROJ item for each project in the company. A PROJ item may contain an arbitrary number of references to TS items, implementing separate scopes of each project (REQ_10). Each TS item corresponds to a test specification. A TS item, in turn, may contain an arbitrary number of references to TC items, where each TC item contains one and only one reference to a TCS. Note that although a TC item may only reference one single TCS item, there is nothing to prevent that several different TC items reference the same TCS item, thus the same TCS can be used in several different test specifications in the same or different projects (REQ_11). Using this chain of references starting from a project item, it is also possible to query the database for information such as planned and, thus far executed test cases (REQ_12). Also note that the introduction of the TC entity in the model does not destroy the possibility of automatically generating paper versions of test specifications and test instructions.

4.3 Elaborating on Test Script

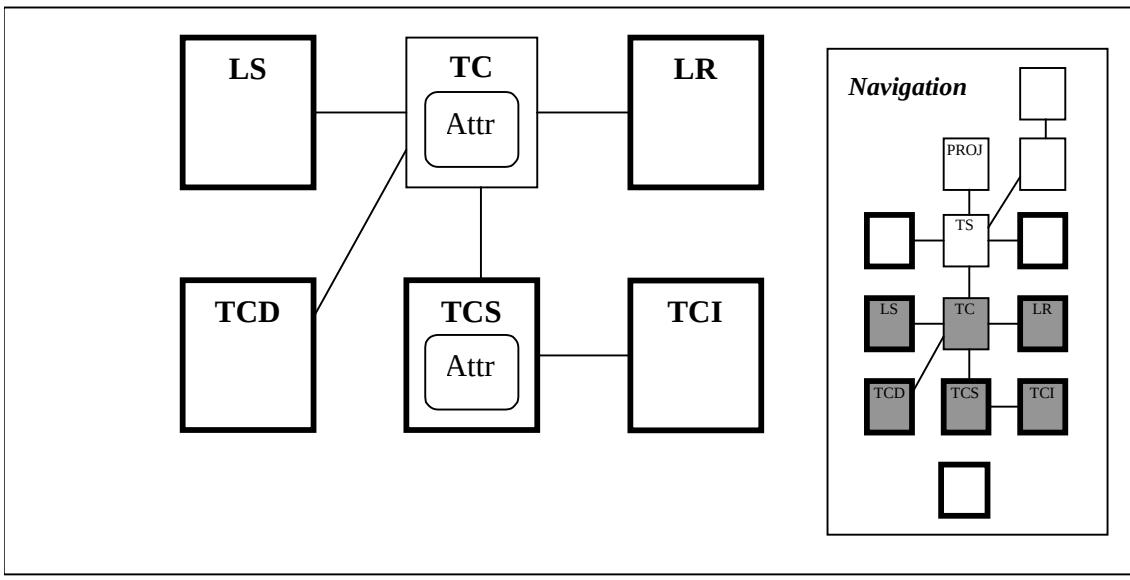


Figure 6 Solution in part for modularized test script

Up to this point, the modularization of test scripts has been ignored. If the test execution is only performed manually, with step-by-step instructions being the contents of the test instructions, it is quite enough using the TCI as described above. However, if the test execution is automated, using some kind of scripting language, then the following extensions of the test database should be considered.

As stated in the requirements there is often a need to parameterize the test cases by the separation of code and data. To solve this problem, a *TCD (Test Case Data)* entity is introduced in the database (REQ_15). The contents of a TCD item can be described as the values of the parameters that are to be used during the execution of a TCI item. As the idea with parameterization is to re-use the code, there can be several different TCD items that can be used with the same TCI item. Here we regard two executions of the same TCI item with different TCD items as separate test cases. Thus, the natural place in the database model for the TCD entity is to associate it with the TC entity, using a one-to-one relationship, which is illustrated in Figure 6.

Sometimes test cases are constructed, to require the test object to be forced into a starting state, before the execution of the actual test case can be commenced. When automatic test case execution is used, this may be accomplished by using a special setup script, tailored to suite the particular test case. In the same

manner, a restore script can be employed after the execution of a test case to restore the state of the test object to basic state, from which the prepare script of the next test case can be started. To illustrate these concepts consider the scenario in Figure 7.

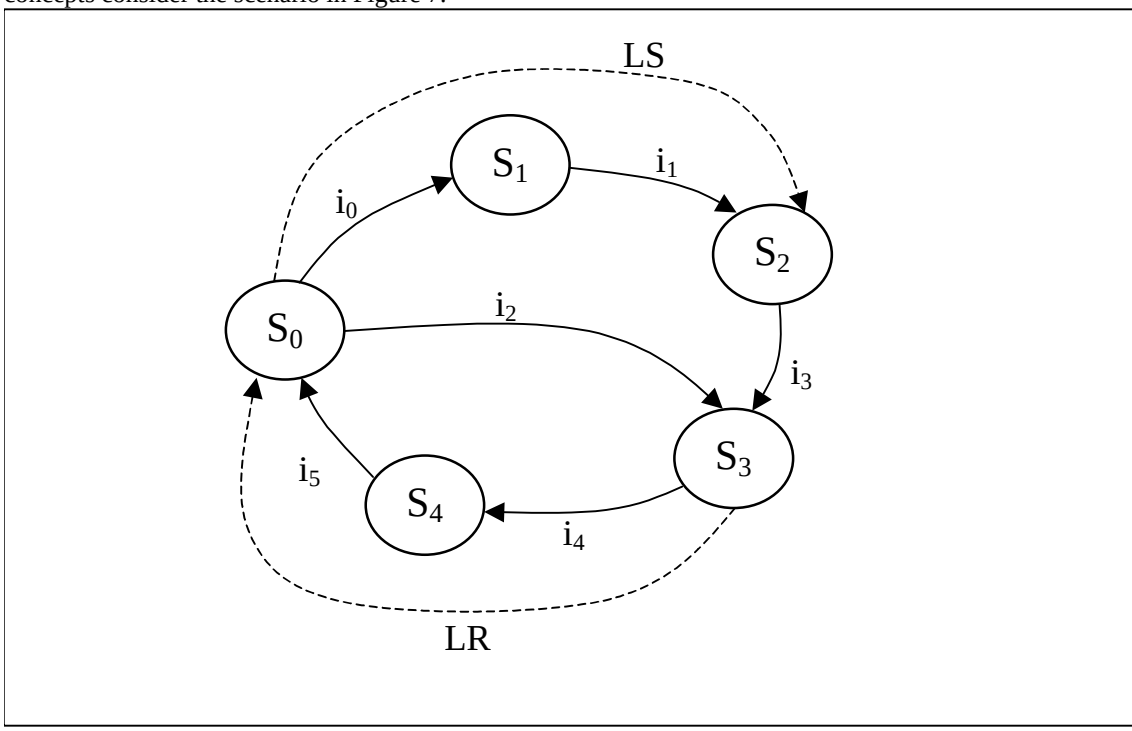


Figure 7 Illustration of the use of Local Setup and Local Restore actions

Suppose we want to test the transitions i_2 and i_3. For testing i_2 we need the test object in state S_0 prior to the test case execution. After the test is executed the test object state will be S_3. Thus, we need to construct a restore script to return the test object state to S_0. For testing the i_3 transition, we need the test object in state S_2. This is accomplished by using the set-up script. To restore the test object state after the test execution, the same restore script as in the previous test case can be used. This illustrates some interesting points. First, by using the same start and end state for all test cases, arbitrary test cases can be executed in arbitrary order. Second, it is evident that setup and restore scripts may be re-used for several test cases, which is an argument for separation of these scripts and the actual test case. Third, assuming that we have test cases and test scripts organized in modules, as described above, then there is an option to substitute some setup and restore scripts with actual test cases, if that is desired. The conclusion is that modularizing the scripts, supports flexibility in the execution. To handle setup and restore scripts in the database, we introduce *LS (Local Setup)* and *LR (Local Restore)* entities.
As the same setup and restore scripts may be used several times with different TCI items, but only one setup and one restore script for each test case, the natural place in the database model for these entities is associated to the TC entity, as displayed in Figure 6.

The effect of organizing the database in the manner just described is that the database does not impose any requirements on how to structure the test scripts. If parameterization of test scripts is not available, or not desired the TCD entity can just be omitted, and if the setup and restore actions are built directly into the test scripts, these entities may also be ignored. The database organization even permits a mix of strategies, where some test cases require explicit TCD, LS, and LR items and other test cases have everything built into the TCI directly.

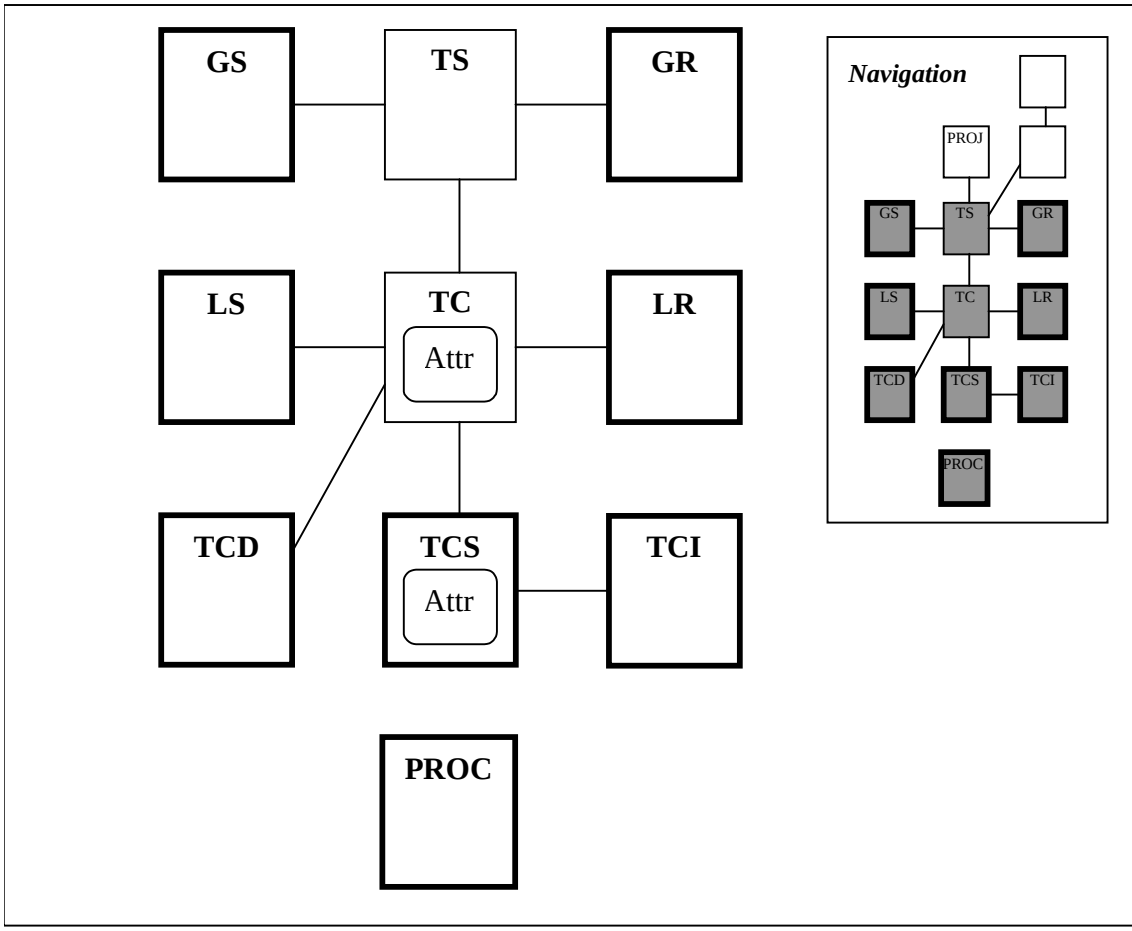


Figure 8 Full solution for modularized test script

In some cases there might be global setup and restore routines valid for all test cases in an entire test specification. Imagine for instance all test cases in a stress test using the same background load. A script to start a load generator for that load would be generic for all test cases in that test specification. Thus, we also introduce the two database entities *GS (Global Setup)* and *GR (Global Restore)*. As these entities are common for all test cases in an entire test specification, we have decided to associate these entities with the test specification, shown in Figure 8.

The final part of the discussion of the modularization of the test scripts concerns procedures. Some scripting languages will allow procedure or function calls within the script itself, that is why it might be convenient to include a separate database entity for *procedures called PROC* (REQ_16). Procedures are used when several test scripts contain identical parts. These parts may then be broken out into separate procedures, which are called from the different scripts at appropriate times. Procedure calls may be used in any script module containing code, i.e., TCI, LS, LR, GS and GR. Procedure calls in most languages, are usually made by name, i.e., the name of the procedure is the unique identifier of the procedure. This means that the relation between the caller and the callee is already implicitly built into the model. Thus, we have included a procedure entity in the database model without any explicit relations to any other entity of the database.

With the entities introduced above, we conclude that the database model fully supports a modular test script organization (REQ_17).

4.4 Test Execution and Results Reporting

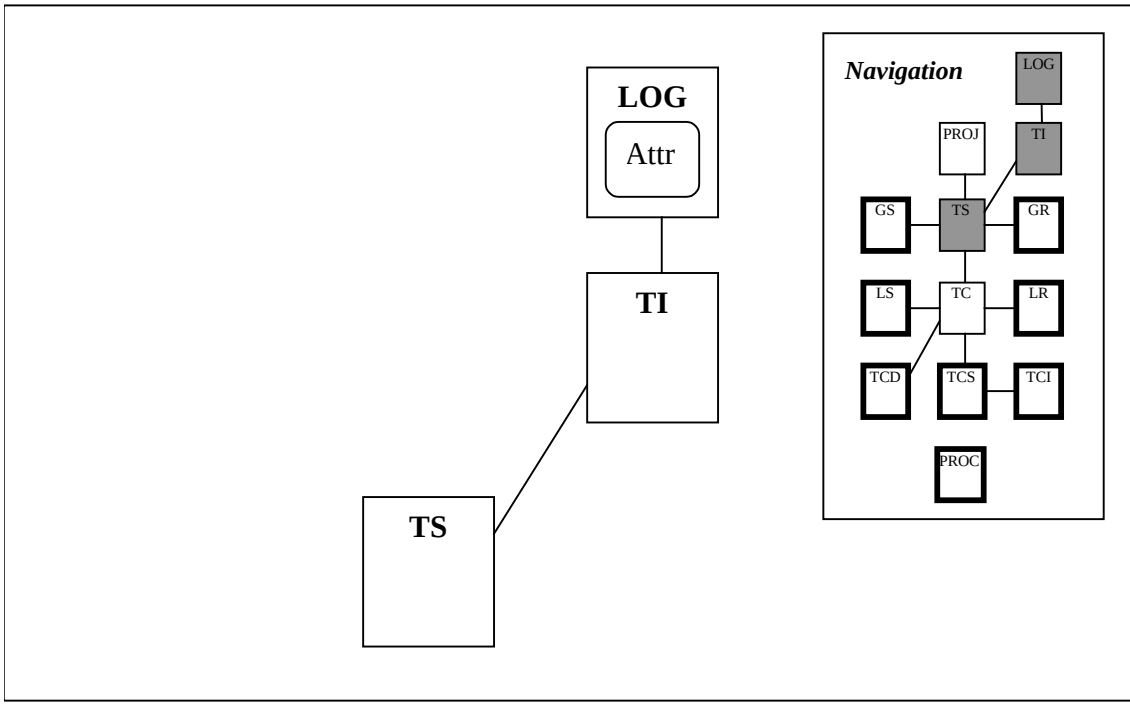


Figure 9 Storage of test execution results in the test case database

As described earlier, the database model supports automatic generation of test instructions, by following the appropriate relations in the model, and possibly resolving procedure calls. If we assume that each test case has implicit or explicit setup and restore actions associated, as described above, then the order of execution of the different test case is potentially independent. This means that, if every test case starts and stops in the same test object state, then we can execute the different test cases in arbitrary order, and we might even chose to select only a subset of test cases for execution. The database model, described in this paper, clearly supports arbitrary selection and ordering of test cases. However, from a test reporting perspective, it is important to keep track of what has been done and also to be able to recreate previous results. Thus, for a database solution to be complete with respect to test case selection and ordering, the database also need to support storage of generated sequences of test cases, and the results from executing these. We have built this functionality into the test case database by introducing a *TI (Test Instruction)* entity and a *LOG* entity. The TI is responsible for storage of test script sequences, where a test script sequence is automatically assembled from the different test script items directly or indirectly referenced by a certain TS. The LOG entity is responsible for storing the results, i.e., log files from the execution of a certain TI.

When deciding upon a selection of test cases to be executed, the basis is the test specification, since it holds all the test cases possible to choose from. The natural place for the TI entity is thus related to the TS entity. A TS item may reference an arbitrary number of TI items since the same test specification can be the base for several test instructions by varying the [order of] the included test cases. An important observation is that a TI item need not contain the actual test script. It only needs to contain the identities of the included test cases, and the order of execution of these, since the complete test script always can be (re-)generated from this information.

When executing a test instruction there are usually several types of results. Pass/fail information and error reports have already been discussed when describing the TC entity. Logs are another type of results that usually contain information generated by the test object during the test execution. A log is thus dependent on the test instruction used, but each time a test instruction is executed, a new log will be produced. From this we decided to let the TI entity relate to the LOG entitiy, with a one-to-many relation. The LOG entity may either contain a simple file reference or the complete log, depending on the storage capacity of the database. The final result is that we have a test case database where it is possible to report and store different types of test result based on the execution of test cases (REQ_13). These results can then be presented either from a project perspective or from the basis of a certain test case (REQ_6).

5 Minimal implementation

In this section we present how some of the ideas presented in this paper can be evaluated at a very low cost, by making a small trial implementation. The only prerequisite of this minimal implementation is access to an operating system supporting links and a version control system, e.g., Unix with ClearCase. The idea is to realize the relations in the database through two parallel hierarchies of directory structures, where the connection between these two structures is implemented by soft links.

The main drawback with this minimal implementation, apart from being crude, is the lack of support for automatic generation of TS and TI documents and the problem of accessing status information. However, since the artifacts of this implementation are the same as for the full implementation, the work invested in such a pilot implementation, is easily regained in a full scale implementation, by just moving the artifacts as they are into the new system.

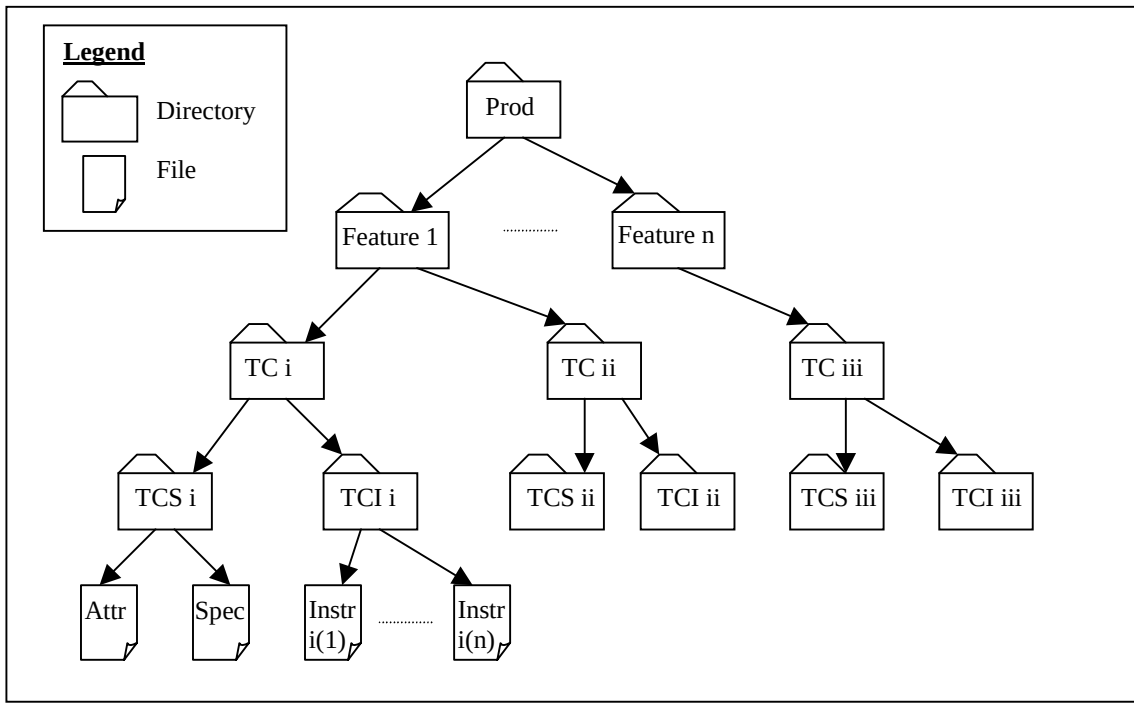


Figure 10 Product related test case hierarchy

The first step is to implement a tree structure where the different product related database entities, e.g., TCS, TCI, LS, LR, etc are maintained in different sub-trees of the structure. Files in this product structure should be kept under version control, and through some basic operating system supported search mechanism desired files can be located whenever needed.

Any product structure existing within the company may form the basis for such a tree structure. In our example, we use a five level directory hierarchy for the product related directory tree. To simplify the picture, Figure 10, shows such a tree with only TCS and TCI entities indicated. However, the idea is the same for all product related entities. The top level of the product hierarchy is a representation of an entire product or product family. The second level set of directories splits the product up into features or functions. The primary motivation of this level, is to narrow the search space, when a certain test case is looked for. The third level in the product hierarchy is the test case level. Each test case belonging to a certain feature is allocated a directory of its own in the sub-tree of that feature. The main motive of the test case level of the hierarchy is to implement the TCS-TCI relation in the database model previously introduced. Thus, a test case directory contains one TCS and one TCI directory, where these two sub-directories implement the what and the how of a test case. Note that the test case directory is not the same as the TC entity in the database model. The TC entity in the database model is project related, while this test case directory only exist for the purpose of keeping related TCS and TCI items together.

In the normal case, the TCI directory contains one text file, implementing the script or the manual instructions of the test case. However, if there are multiple versions of the script, all versions are stored in the same directory.

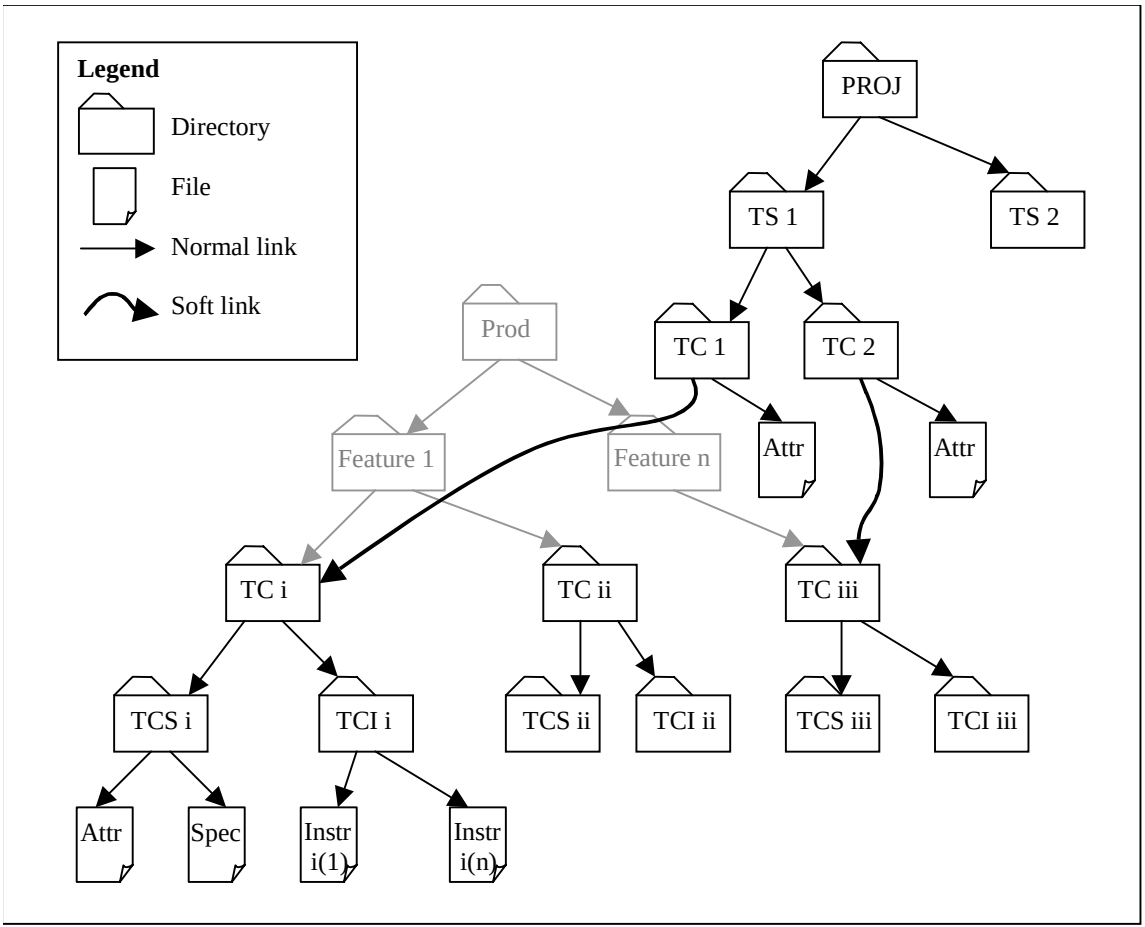


Figure 11 Project related hierarchy superimposed on product related test case hierarchy

We store two text files in a TCS directory. One of the files contains the actual test case, and the other file contains the product related attributes of that test case. Splitting this information into two files is mostly a

matter of taste, although for version handling purposes, it might be convenient to have the text of the test case separate.

The next step is to create a project hierarchy in the file system, which is superimposed on the product structure, using links. The top catalog of the project hierarchy corresponds to the PROJ entity of the test case database, with one separate directory for each project in the company. In the second level of the project hierarchy, folders corresponding to the TS entities are created. Thus, a project may contain a set of test specifications. For each test case in the test specification, there should be a test case directory in the TS directory. These test case directories correspond to the TC entity of the database model. To store the project specific information, e.g., pass/fail results, the TC directory contains a text file. In addition to this text file, there is a link to a test case directory in the product related file hierarchy. This link implements the relation between the TC and the TCS in the database model, and hence, we have succeeded in implementing a file structure where product and project artifacts can be separated, which is one of the main characteristics of this work.

6 Full Implementation

An old incomplete version of a test case database, called Test Case Library (TCL) has been in use at Ericsson Utvecklings AB Y/VF since 1990. The main task of this organization is to perform system tests of computers in large telephone exchanges. Focus in the testing is on stress, performance and other non-functional tests, although some functional testing is also performed. The experiences from the use of TCL, led to the realization that a modernization of the tool was needed. A requirements capturing project was initiated, which in turn led to a full scale implementation project. The result of the implementation project is a custom-design version of the test case database, called TCL 2000 [ER01]. Enea Redina, made the implementation, which roughly consists of 500.000 lines of Java and SQL code, and has taken around one man-year to develop. The contents of the old TCL database were converted to the new format dictated by TCL 2000, and the tool itself was launched in May 2001, together with the start of a new large project.

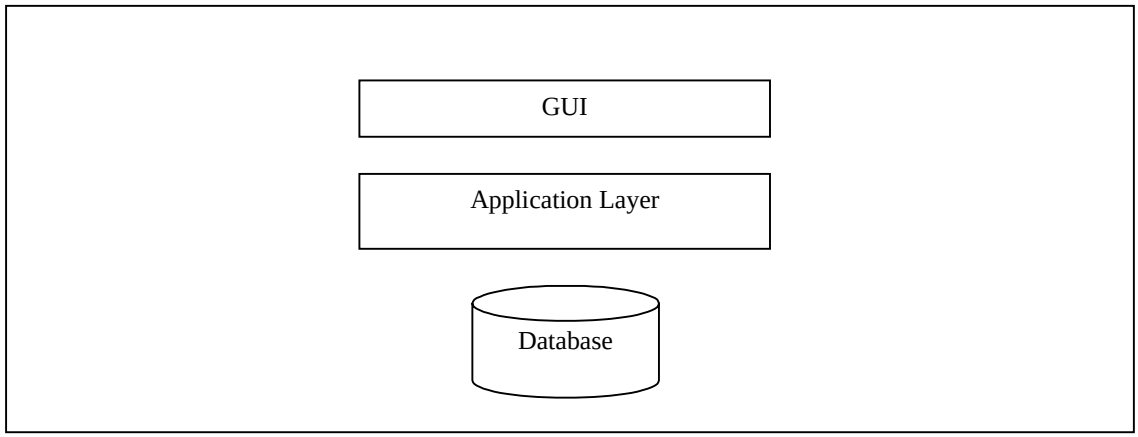


Figure 12 Architecture of TCL 2000

Figure 12 shows an overview of the architecture of TCL2000. Most of the intelligence, e.g., test script generation, is built into the application layer. The database, built on the model presented in this work, is extended with some minor application dependent features. The application layer contains most of the intelligence for transforming the data of the database into desired formats. One such example is the logic for compiling complete test scripts from the different test script modules in the database. Through the use of the Ericsson internal scripting language Autosis, limited support for automatic test case execution has been added to the tool. Another example of intelligence included in the application layer is the support for iterations of test scripts. A key feature in stability testing is to run the same test cases over and over again, to see that the test object can handle continuos load. The application layer contains an iteration facility

allowing the tester to force the test script generated to loop over test cases a predefined number of times. The GUI is supported both in Windows and Unix environments.

7 Related work

7.1 Commercially available tools

A number of commercial alternatives exist, which implement some of the functionality described in this paper. The advantage of the work presented in this paper over all the commercially available products is the separation of the artifacts into project and product artifacts. This is an important difference, since it enables the product related artifacts to be stored in a product structure simplifying the localization, thus the re-use of the artifacts. This division also permits a controlled development of the product related artifacts as the product itself evolves. The following sub-sections contain some highlights of some alternatives to the work presented in this paper. Neither the list of tools, nor the description of each tool is complete. The purpose of the description is merely to give the reader a hint of what exists.

7.1.1 TCS

Test Control System (TCS) by TESTMASTERS [Te01] supports a hierarchical organization of test cases within a project. It also supports the division of test cases into what and how parts. Compared to the work presented in this paper the major down sides of TCS are lack of revision handling and organization of test artifacts that support both inter and intra project requirements at the same time.

7.1.2 Test Case Manager (TCM)

Test Case Manager (TCM) by Pierce Business Systems [PB01] is a freeware tool based on a Microsoft access implementation. It is intended for small to midsized companies. The access source code is available making it possible to make own adjustments to the implementation. TCM supports a hierarchical organization of the test cases, but only with a project focus. Furthermore there is no support for version handling of the test artifacts. TCM is also primarily intended for manual test execution, which is why there is no support for modularization of different test script entities.

7.1.3 TestManager

TestManager by Rational [Rat01] is part of their integrated tool suite for testing. It offers much of the functionality described in this paper. The major disadvantage with TestManager is the lack of distinction of product and project artifacts. There is a possibility to organize test cases in an arbitrary manner within a project, but no natural way of extending this organization across several projects. A consequence of this is that re-use of test cases both within and across projects is only possible by making copies of the original test case, which in turn leads to a multiple source update problem.

7.1.4 TestDirector

TestDirector by Mercury [Mer01] is a test tool, which coordinates support for requirements management, test planning, test scheduling, test execution, and defect tracking. Test case organization is managed in a user defined collapsible subject tree, but without support for either multiple roots of the tree, or version handling. The main strength of TestDirector is the integration of the different functions into one single tool.

7.2 Research

A search in science citation index and other research databases reveals virtually no work done in the area of test case maintenance.

8 Conclusions

In this work we have presented a logical database model for test artifacts. One of the aims with this database model was to make maintenance and re-use of test cases easier than with a document based storage. The single most important aspect of this work is the realization that some test artifacts are project related, while other test artifacts are product related. In particular different attributes of the test case are product and project related respectively, which led to the splitting of the test case into several different database entities with relations between them. Further, to facilitate maintenance, we also introduced revision handling of the product related artifacts. The model also supports the storing of test execution

results including logfiles generated by the test object. This facility helps integrating status reporting into the tool. Status can be reported both on projects and products due to the separation of information in the database.

The database model is kept generic without any conscious coupling to a particular type of test object, nor any assumptions about scripting language or even automation of the testing. This means that an implementation of a tool based on the database model presented, is likely to work in any test environment.

We have also shown that it is indeed possible to build a tool for test case management based on the database model. Although the tool has only been in use some months, there are several positive experiences reported. Some of these experiences are based on several years use of the old tool called TCL. For instance reviews of the test cases have improved since the contents of a test case can be reviewed in isolation with optimal staffing. Once each test case is reviewed and approved, a group of test cases can be reviewed for coverage. In this review, no effort has to be spent on whether or not the test cases themselves are correct. Another positive experience relates to the maintenance aspects of this work. Since it is easy to locate test cases with a certain attribute, testers can use old test cases as templates for new ones during test case development. This makes test case writing more efficient. Furthermore, the structure of the test cases become more homogenous, which in turn helps knowledge transfer between different testers.

Finally, we have presented an inexpensive and simple way of evaluating some of the benefits of the logical test case database model presented in the paper.

9 Acknowledgements

Christian Jacquet and the rest of the staff of testing department at UAB/Y/VF for being very helpful and supportive, and the main sources of knowledge when the requirements of TCL were being captured.
Åsa Grindal for giving me a lot of helpful comments during the writing of this article, and last but not least colleagues at Enea Realtime for the help and inspiration throughout the years.

References

[Bei90] Boris Beizer, Software Testing Techniques, Van Nostrand Reinhold, 115 Fifth Avenue, New York, NY 10003, 1990

[BJ00] Bogdan Bereza-Jarocinski. Automated Testing in Daily Build, Technical Report 3-2000, Sveriges Verkstadsindustrier, 2000

[BS98a] British Standards Institution BS 7925-1 Glossary of Terms Used in Software Testing, 1998

[BS98b] British Standards Institution BS 7925-2 Standard for Component Testing, 1998

[ER01] TCL2000, Enea Redina, att. Anders Pettersson Smedsgränd 9, S-753 20 Uppsala, Phone: +46 (0)18 66 08 00, Fax: +46 0(18) 66 08 02

[Mer01] Mercury Interactive – TestDirectorhttp://www-svca.mercuryinteractive.com page visited July -01

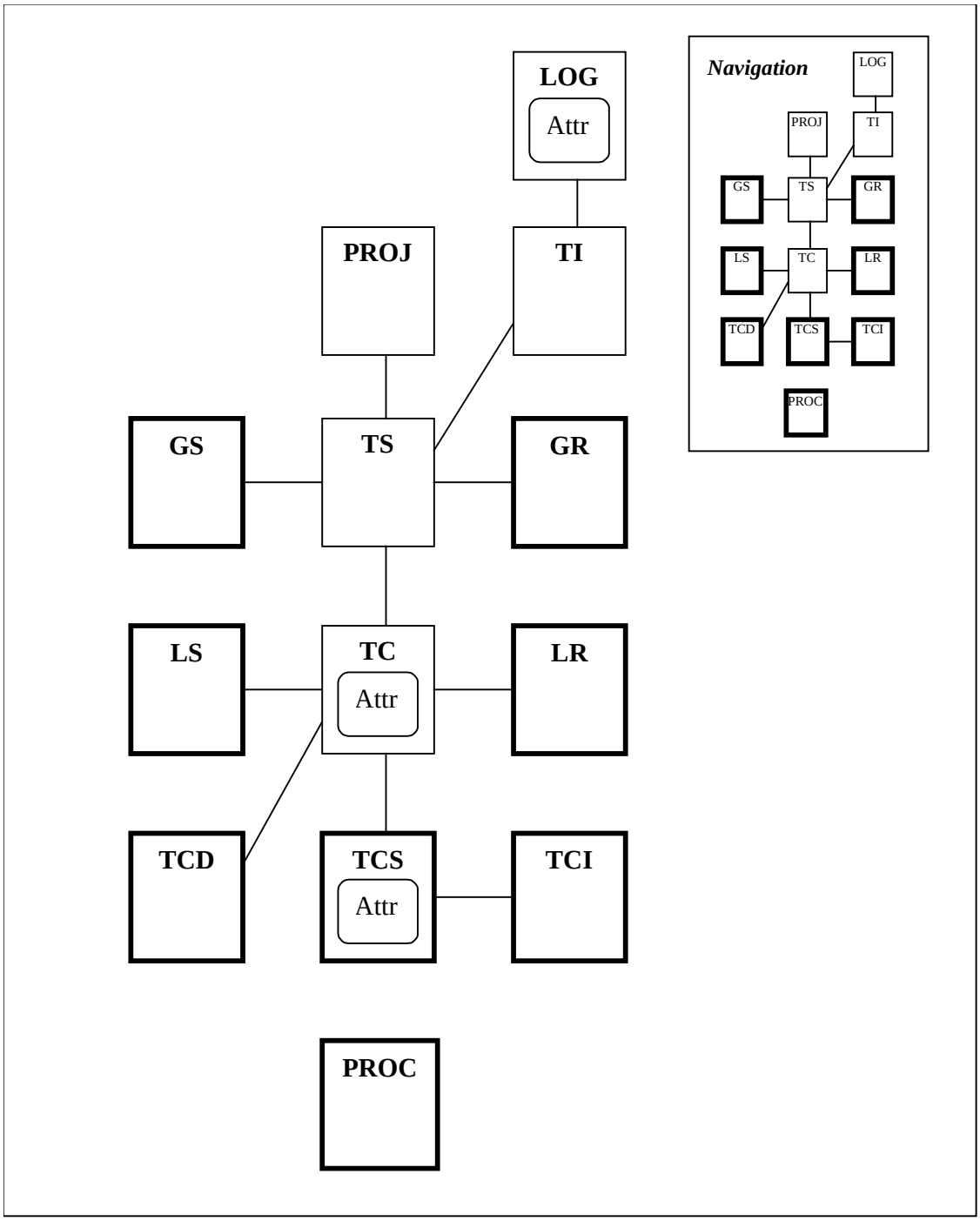
[PB01] Pierce Business Systems - Test Case Manager (TCM) http://jupiter.drw.net/matpie/PBSystems/downloads/freeware/devtools/TCM.html page visited June-01

[Rat01] Using Rational TestManager version 2001.03.00, part number 800-023807-000

[Te01] TESTMASTERS – Test Control System (TCS) http://testtools.com page visited July-01.

Appendix A – Complete Database Model

This is the complete database model introduced in this paper.



Appendix B – Overview of Requirements

The following list contains the requirements made explicit in section 3. Just as in that section the requirements are ordered consecutively. As been pointed out previously, this is just a small subset of requirements of a real implementation. The reason to use this particular set of requirements is to illustrate some of the key features of the test case database.

Req. Id	Req. Text	Page
REQ_1A	There should be support for generating a test specification from the database repository.	3, 8
REQ_1B	A test instruction is project related, and should have the possibility to be automatically generated.	3, 8
REQ_2	All test artifacts stored in the database, should have unique identities.	3, 7
REQ_3	The test case database should include support for traceability between requirements and test cases.	4, 6
REQ_4	There should be possibilities to store a test case result (pass/fail) in context of a project.	4, 8
REQ_5	There should be possibilities to store a reference to an error report for a test case in a project context.	4, 8
REQ_6	A test case should include support for referencing the log(s) where this test case has been part of the execution.	4, 13
REQ_7	There should be some general purpose attributes of a test case.	4, 8
REQ_8	Product related artifacts, e.g., TCS and TCI should be able to exist stand-alone in the test case database.	4, 6
REQ_9	There needs to be a general search mechanism to locate possible test case candidates.	4, 6
REQ_10	Each project should have its own scope.	4, 9
REQ_11	Any product related artifact should be possible to be associated to any number of projects.	4, 9
REQ_12	It should be possible to find the number of test cases planned and so far executed in a specific project.	5, 9
REQ_13	The test case database should provide the means for reporting different aspects of the product status e.g., pass/fail information.	5, 13
REQ_14	The test case database needs to support real version handling of the test artifacts.	5, **Error! Bookmark not defined.**
REQ_15	The test case database should support parameterized test scripts.	5, 9
REQ_16	The test case database should support procedure calls.	5, 11
REQ_17	The test case database should support a modular test script organization.	**Error! Bookmark not defined.**, 11

Managing Test Cases Using Database Concepts

Mats Grindal
Enea Realtime AB
magr@enea.se

Outline

Introduction

The Database Model

Implementations

- Minimal
- TCL 2000

Conclusions

Motivation for Test Case Database

Improvements of test specification phase relatively unexplored

Paper documents hamper maintenance and re-use of test cases and test scripts

Tool support for storage, status reporting, traceability, and modularised test script construction

Background

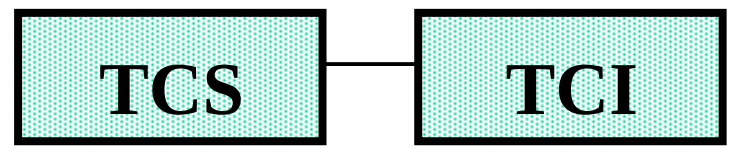
Company

- Large test department
- Mature test organisation
- Old test case database in use since -96

Extensive use revealed potential for improvements

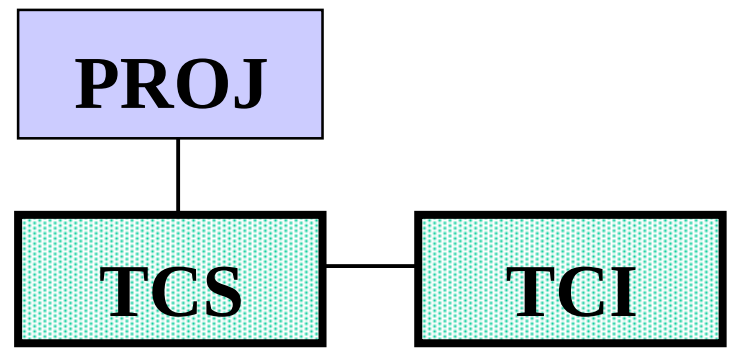
Logical Model 1(9)

Fundamental
Relation



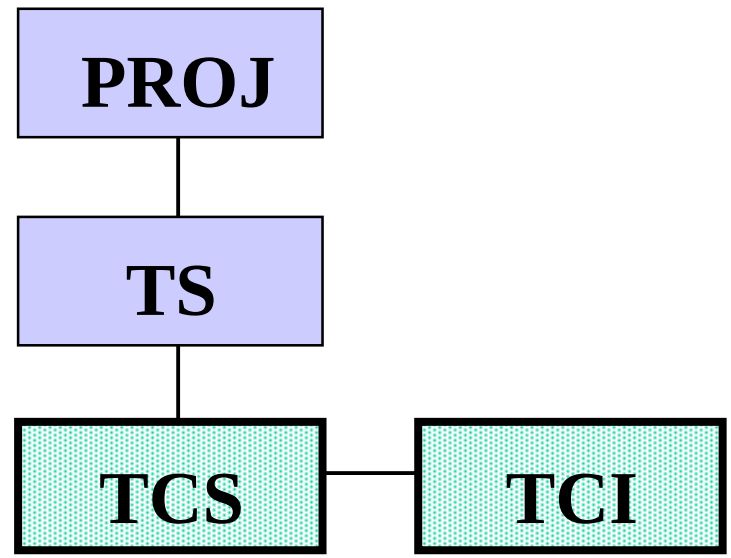
Logical Model 2(9)

Adding a
Project Scope



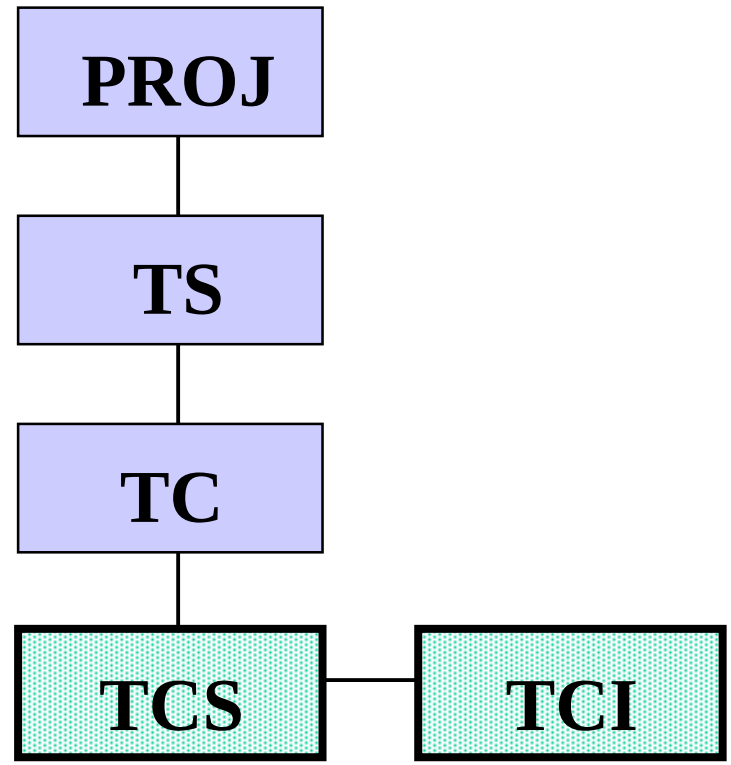
Logical Model 3(9)

Adding a
Project Scope



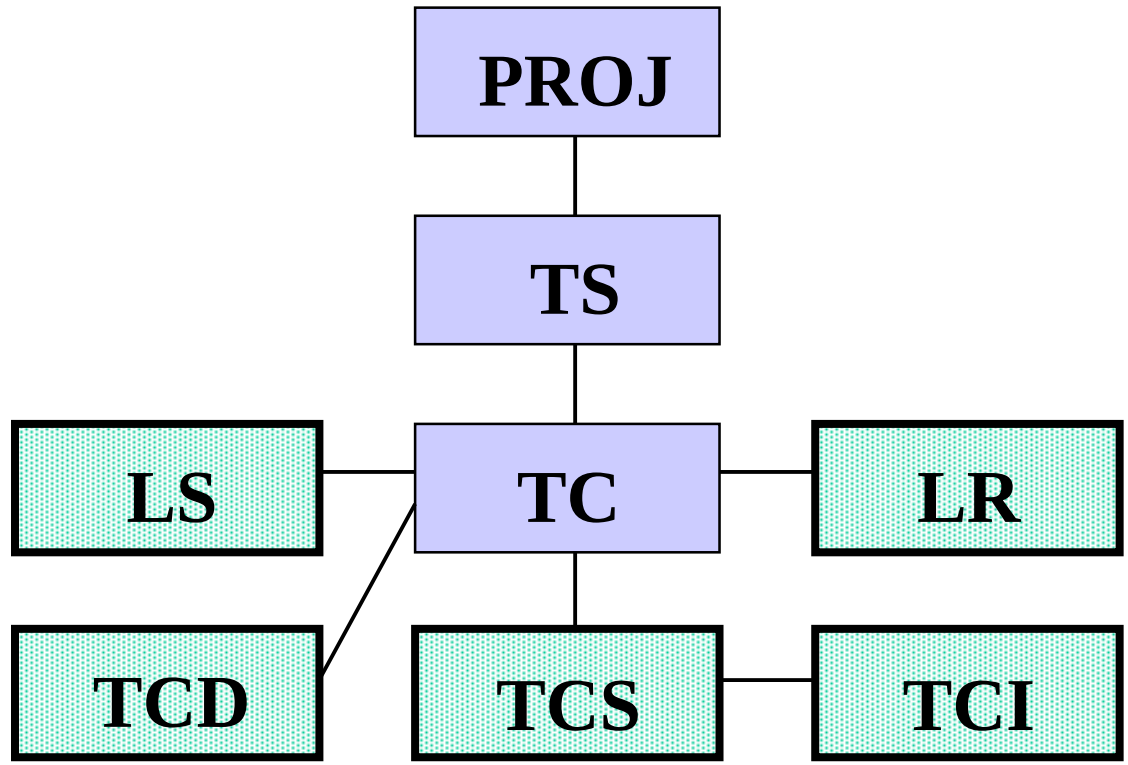
Logical Model 4(9)

Adding a
Project Scope



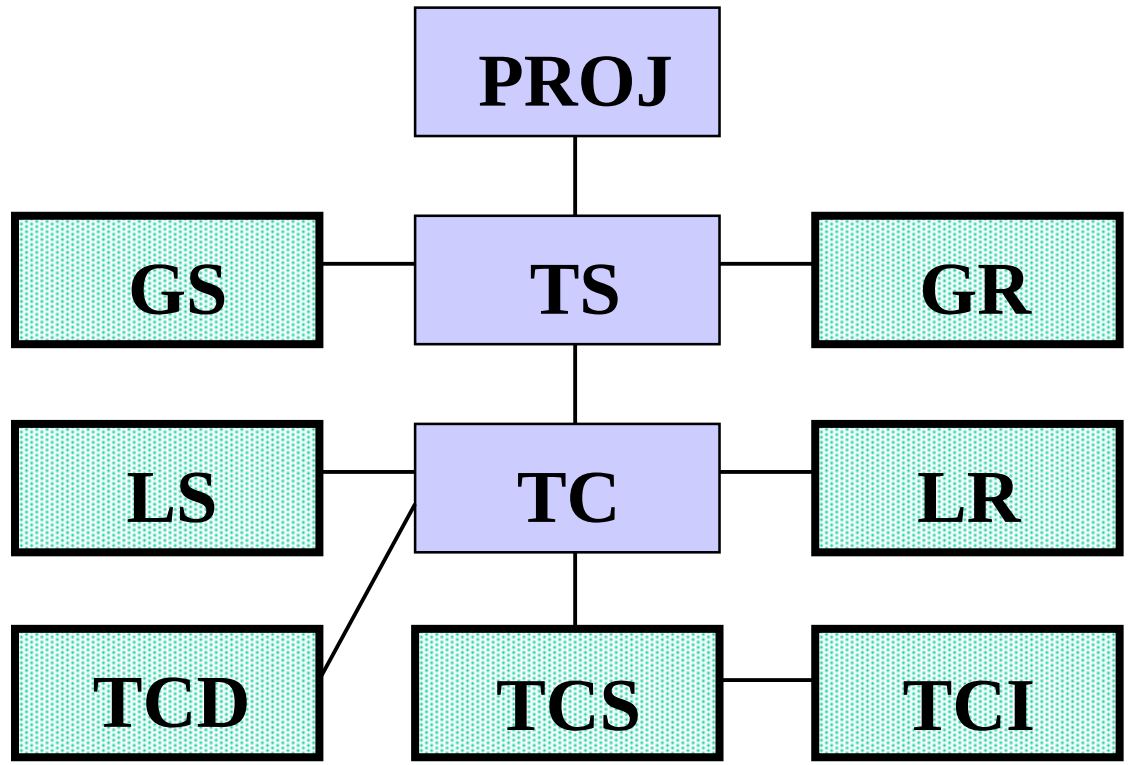
Logical Model 5(9)

Modularized
Test Script



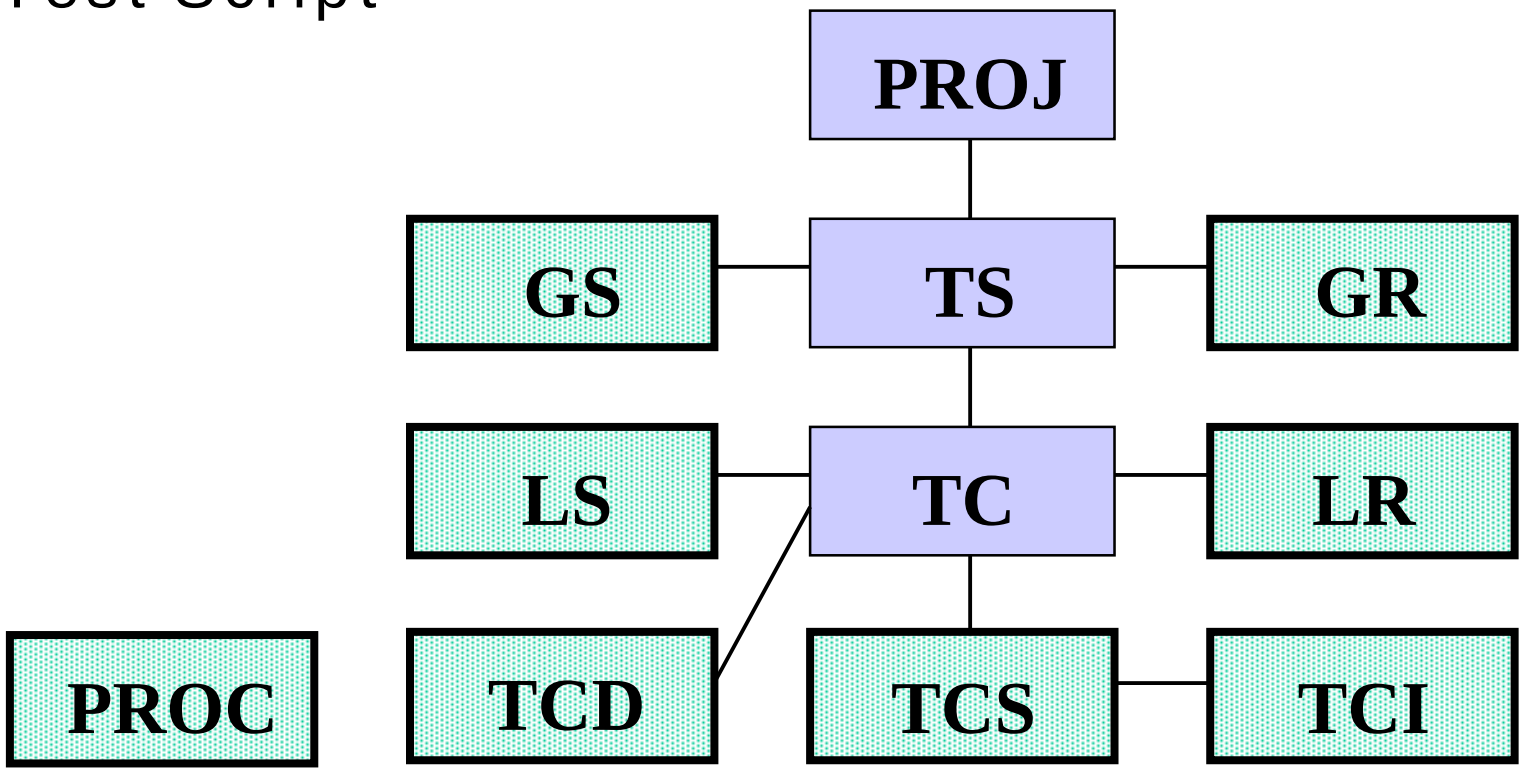
Logical Model 6(9)

Modularized
Test Script



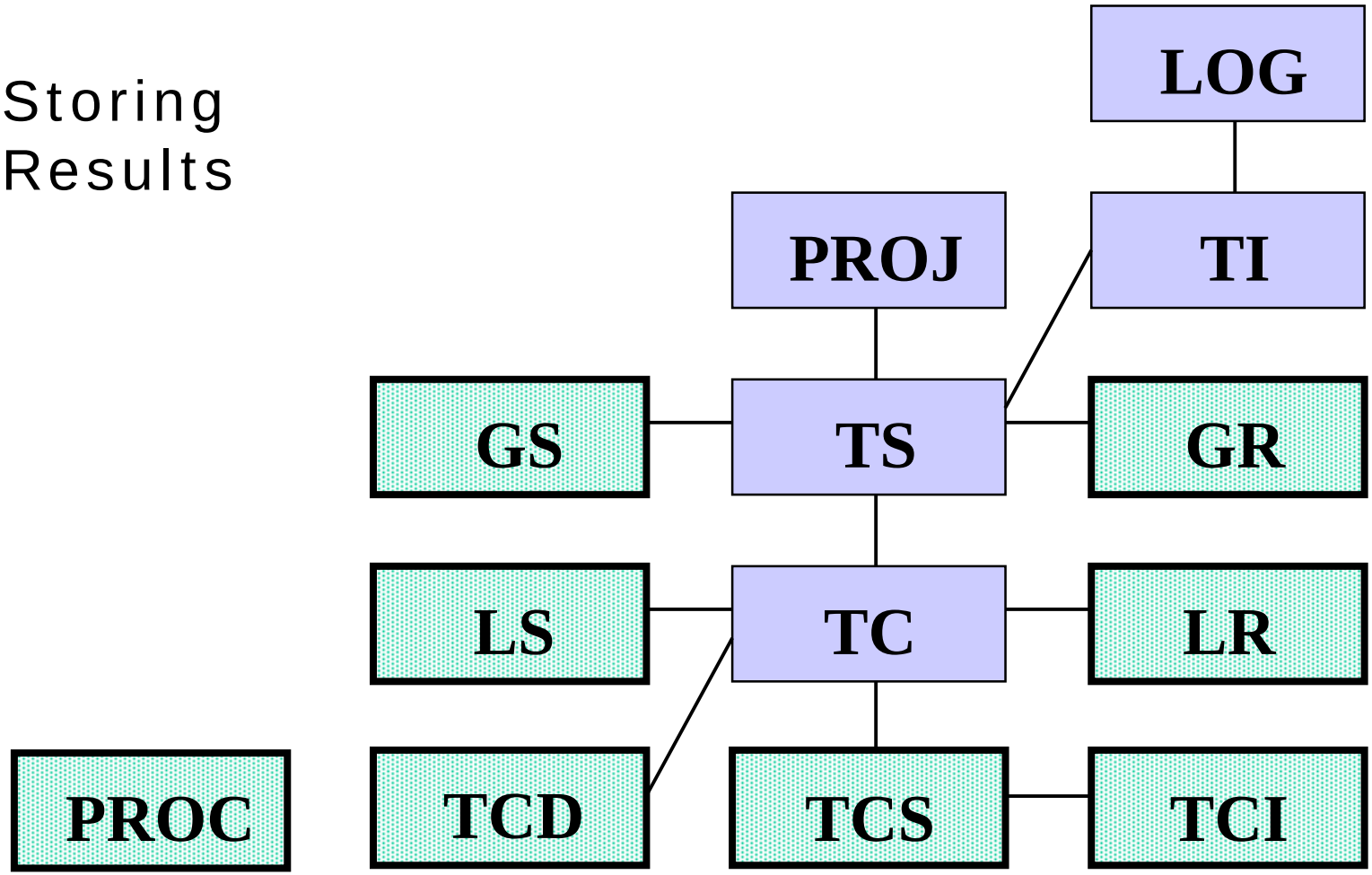
Logical Model 7(9)

Modularized
Test Script



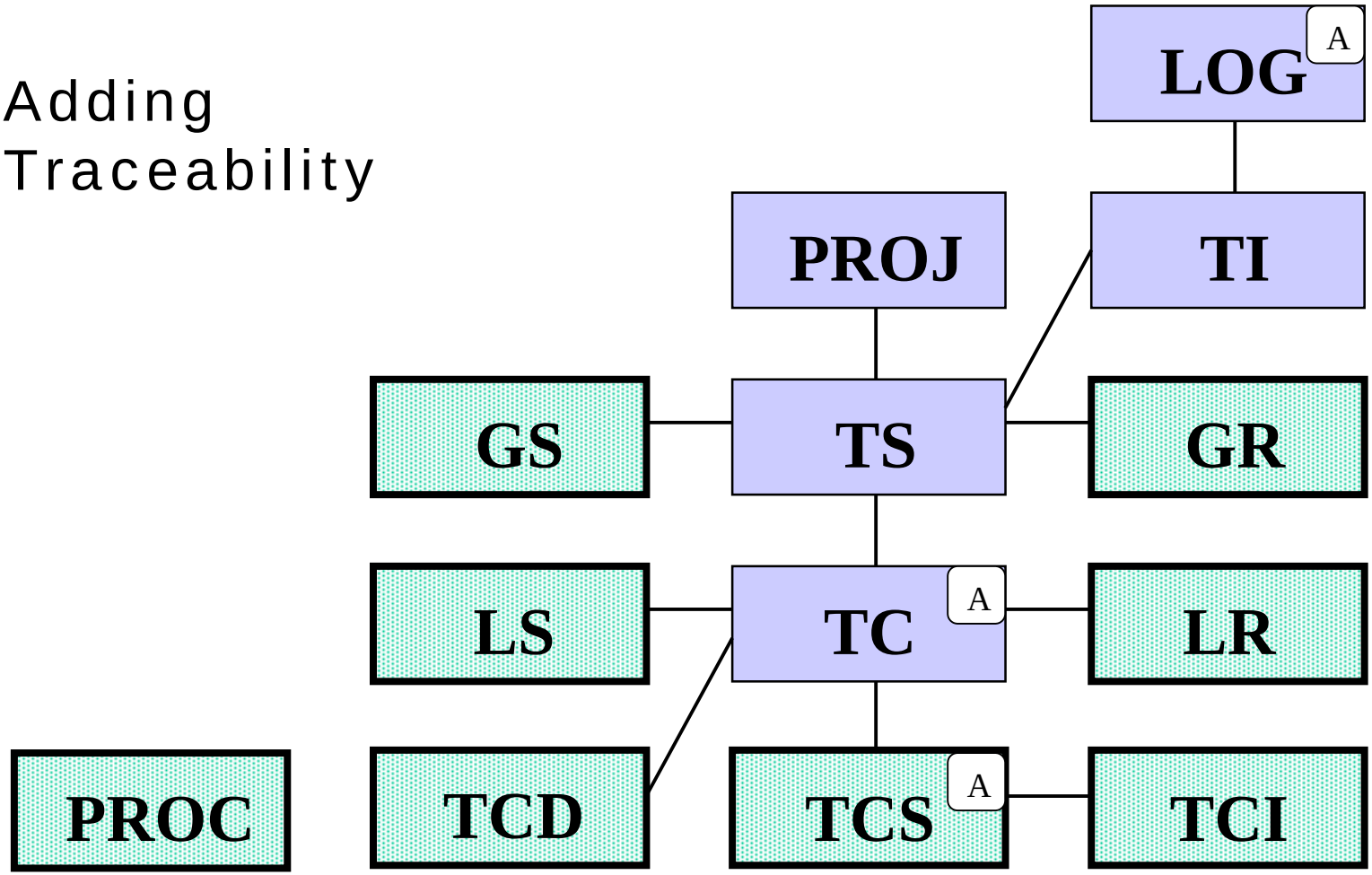
Logical Model 8(9)

Storing Results



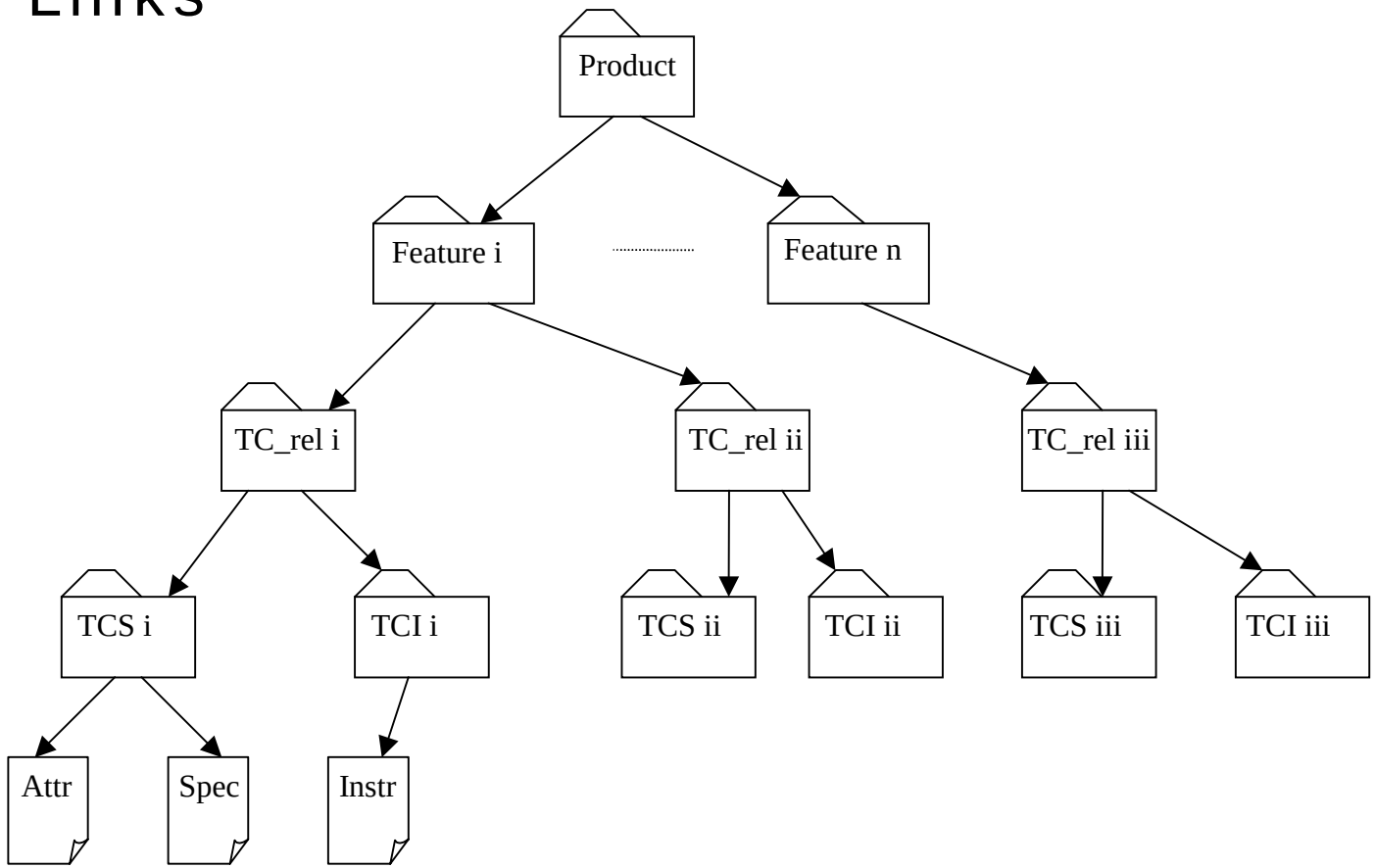
Logical Model 9(9)

Adding
Traceability

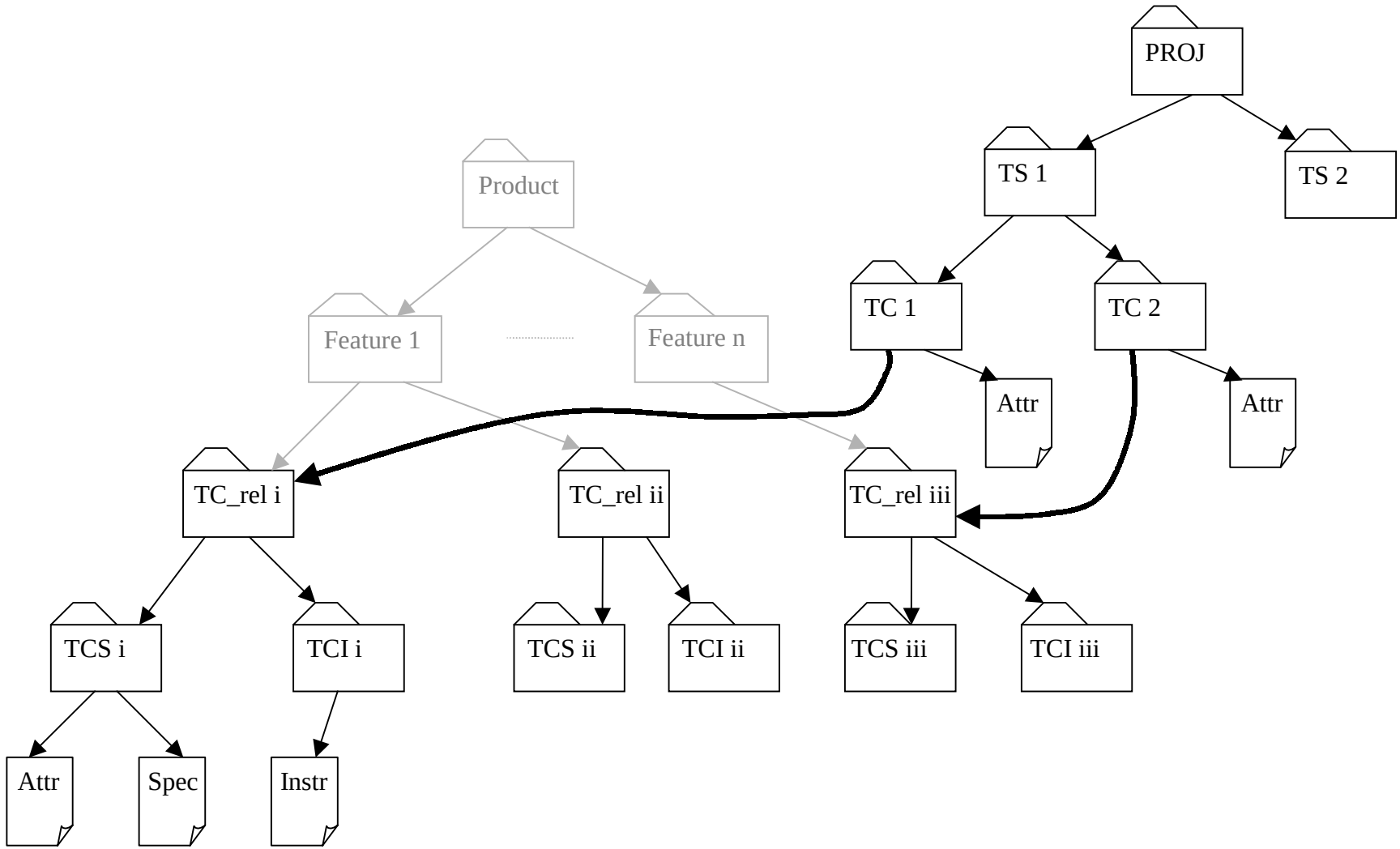


Minimal Implementation 1(2)

Version handling
Links

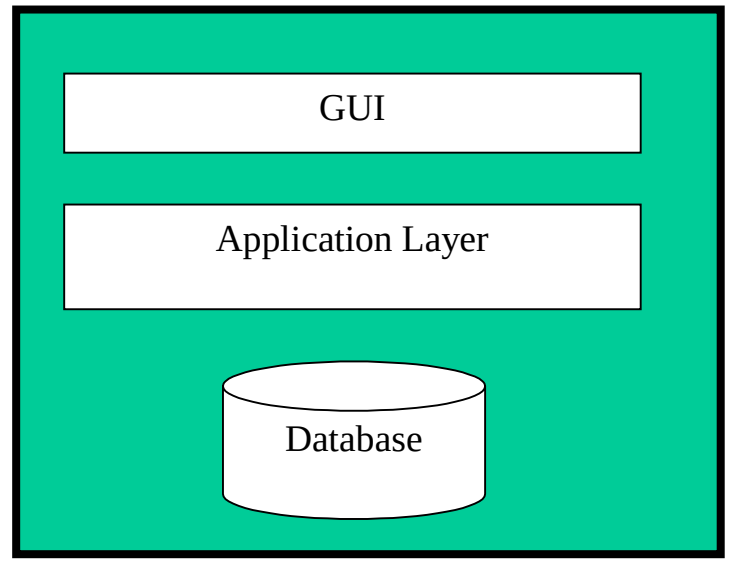


Minimal Implementation 2(2)



TCL2000

Full scale implementation
- One man-year
- 500 KLoC (Java & SQL)
- Operating since May-01



Experiences
- Easy to locate and re-use old test cases
- Good source of ideas
- Version handling no nuisance during ordinary work

Conclusions

Important to distinguish between product and projects related test artefacts

Test case database - a feasible solution for storing test cases and other test artefacts

Parts of the case database model can be emulated inexpensively

Proposed database model works in real life

Acronyms

GR - Global Restore
GS - Global Set-up
LOG - LOG file
LR - Local Restore
LS - Local Set-up
PROC - PROCedure
PROJ - PROJect

TC - Test Case
TCD - Test Case Data
TCI - Test Case Instruction
TCS - Test Case Specification
TI- Test Instruction
TS - Test Specification

Some Requirements - General

- All test artifacts should have unique identities and version handling
- There needs to be a general search mechanism
- There should be some general purpose attributes of a test case
- The database repository should support automatic generating of
  - Test specifications
  - Test instructions

Some Requirements - Product vs Project

- Each project should have its own scope
- Test cases should be possible to associate to any number of projects, including none
- There should be support for status reporting
  - Planned and executed test cases in a project
  - Error report status for project and product
- The database should support
  - parameterized test scripts
  - test scripts with procedure calls
  - modular test script organization

Some Requirements - Traceability

- The test case database should include support for traceability of
  - Test cases and requirements
  - Test cases and test results
  - Test cases and error reports
  - Test cases and logs