PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 08/11

Overview of the Talisman Version 5 Software Engineering Meta-Environment

Arndt von Staa

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 08/11 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena August, 2011

Overview of the Talisman Version 5 Software Engineering Meta-Environment

Arndt von Staa[1]

arndt@inf.puc-rio.br

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
22451-900 Rio de Janeiro, RJ
Brazil

ABSTRACT

This report presents an overview of Talisman's version 5 functionality. Talisman is a computer aided software engineering meta-environment. It focuses strongly on model driven tools. It provides means to build software development and maintenance environments composed of a harmonious collection of representation languages and tools. The set of representation languages and tools may cover a very wide variety of development and maintenance activities.

Talisman operates on a net of workstations, each containing an environment instance providing tools to support some of the activities of a specific software development and maintenance process. The collection of environment instances supports a large portion of the activities of a given development process.

Talisman stores fine grained objects in a distributed repository. The base schema and meta-schema of this repository as well as the definition of user interfaces, representation languages and tools are kept in a definition base. Definition bases are derived from an environment base which contains all facts about supported representation languages and tools. The environment base is used by the environment builder to create and maintain representation languages and to adapt tools to the specific needs of a particular project.

One of the basic aims of Talisman is to compose and maintain code and other artifacts from high level specifications relying heavily on model driven activities. The result of the development using Talisman is a hyper-document interrelating all artifacts that constitute the target system. The construction and maintenance of this hyper-document is achieved by successive transformations, modifications and verifications of a variety of models. To define and fine-tune these tools, Talisman uses an internal programming language, which specializes tools and activities, such as editors, code composers, representation transformers, representation verifiers and hyper-document navigation control.

Keywords: software engineering meta-environment, representation languages, software quality, representation transformation, model driven development, model driven maintenance.

---

[1] Supported by: CNPq grant: 306802-2008-2

RESUMO

Este relatório descreve o meta-ambiente de engenharia de software assistido por computador Talisman versão 5. A ênfase deste meta-ambiente são ferramentas baseadas em modelos. Ele provê meios para desenvolver e manter ambientes compostos por um conjunto harmonioso de linguagens de representação e ferramentas. Este conjunto visa cobrir uma grande gama das atividades de desenvolvimento e manutenção.

Talisman opera em uma rede de estações de trabalho, cada qual sendo uma instância de ambiente que provê ferramentas e linguagens adequadas o sistema objetivo a ser desenvolvido ou mantido. O conjunto de instâncias de ambientes apoia uma grande gama de processos de desenvolvimento de software.

Talisman armazena objetos de pequena granularidade em um repositório distribuído. O esquema deste repositório, bem como as definições das interfaces humano-computador, as definições das linguagens de representação, as regras que governam as meta-ferramentas são armazenadas em uma base de definição. Estas são derivadas da base de ambiente que contém as descrições de todas as linguagens e ferramentas suportadas. A base de ambiente é utilizada para instanciar e adaptar bases de definição específicas para o projeto sendo realizado.

Um dos objetivos de Talisman versão 5 é compor e manter código e outros artefatos derivados de especificações de alto nível de abstração, baseando-se fortemente em desenvolvimento e manutenção dirigida por modelos. O resultado do desenvolvimento e da manutenção realizados com o apoio de Talisman é um hiperdocumento inter-relacionando todos os artefatos que constituem o sistema objetivo em questão. A transformação de especificações em artefatos de baixo nível de abstração tais como código é realizada através de transformações, modificações e verificações sucessivas envolvendo uma variedade de modelos e artefatos. Para realizar estas transformações Talisman utiliza uma linguagem de programação capaz de especializar ferramentas e atividades tais como uma variedade de editores, compositores de código, transformadores, verificadores estáticos, e navegação em hiperdocumento.

Palavras chave: meta-ambiente de engenharia de software, linguagens de representação, qualidade de software, transformação de representações, desenvolvimento dirigido por modelos, manutenção dirigida por modelos.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
e-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Table of Contents

1 Introduction 1
- 1.1 Domains 2
- 1.2 Development models 4
- 1.3 Terminology 5
- 1.4 Overview of this report 7

2 Talisman version 4.4 functionalities 8

3 Functional requirements 12
- 3.1 Software engineering environment user roles 12
- 3.2 Software development activities 13
- 3.3 Representation languages 15
- 3.4 Form programs 17
- 3.5 Representations and artifacts 19
- 3.6 Navigating over representations 22
- 3.7 Cross references 23
- 3.8 Reuse 24
- 3.9 Quality assurance 25
- 3.10 Version control 28
- 3.11 Application generators and transformers 30
- 3.12 Reverse engineering and re-engineering 30

4 Architectural aspects 32
- 4.1 System architecture 33
- 4.2 Environment instance 35
- 4.3 Environment workstations 36
- 4.4 Language category meta-editor interaction 37
- 4.5 Meta-editors 38
- 4.6 Definition base and software base interaction 39
- 4.7 Repository properties 40
- 4.8 Attributes 41
- 4.9 Relations 43
- 4.10 Multi-base 44
- 4.11 Shared software bases 45
- 4.12 Repository integrity 45

5 Concluding remarks 47

6 References 48

1 Introduction

This document reports a research and development initiative aiming at developing version 5 of the computer aided software engineering meta-environment Talisman. Based on the experience gained with an earlier version (4.4) of Talisman [Staa, 1993], an initial draft of this document was written in 1995 [Staa and Cowan, 1995]. Later on it was massively revised several times. This report provides an abridged description of the system now under development.

As shown in figure 1, the goal of a software system is not just being available, but is supporting users to adequately and dependably perform their work. As mentioned by Brooks [Brooks, 1987] the service of the software is its essence, while the way it was implemented (architecture, design, or code) is accidental. The term *user* is employed in a broad sense considering humans as well as hardware or other software systems with which the target software interacts.

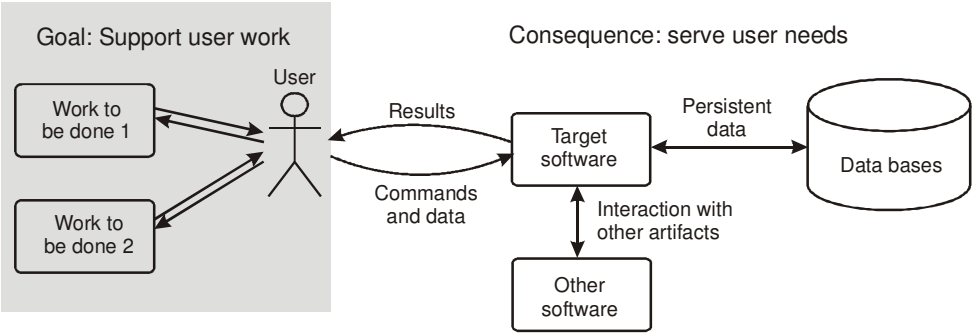

Figure 1. Software service supporting user needs

Software is composed of a collection of artifacts such as specification documents, architecture designs, executable code, initial data bases, source code, test scripts, engineering and user documents, help files, executable tutorials, development plans and many others. As mentioned earlier, this rather large collection of artifacts aims at providing users with adequate and dependable computerized tools to achieve their goals.

Many software systems are long lived, but even when short lived the collection of artifacts must usually be maintained consistent during the whole life cycle of the software. However, several authors report that software maintenance leads to structural deterioration or decay [Eick et al, 2001; Tvedt et al, 2002; Hochstein and Lindvall, 2004]. Hence, to prevent the software deterioration, the whole collection of artifacts should be kept consistent while being changed [Kajko-Mattson, 2000; Arisholm et al, 2006]. Furthermore, many changes will require reengineering the affected artifacts in order to prevent architectural or design degeneration. Unfortunately this reengineering work is often deferred in order to reduce maintenance costs, where such costs are usually due to inexistent or inadequate development and maintenance tools. However, not performing this reengineering may lead to considerable future debts [Sterling, 2011].

1.1 Domains

We call *application domain* the service that software must provide to satisfy the needs of its users as observed by them. The *service* of the *target software* is the set of functionalities and non-functional requirements that it offers to its *users*, see figure 1. The service must support the *target user* to adequately and reliably accomplish hers/his tasks. As mentioned earlier, users are not interested in just using some software; they are interested in accomplishing their work with the aid of adequate and dependable software [Avizienis et al, 2004].

The application domain is not static; user needs evolve during the software's life-time. The longer the life expectancy of the software, the more one should expect that it will change in order to adapt to newly observed service needs and contextual changes, such as software platform and hardware [Beck, 2010; Lehman, 1996; Lehman and Belady, 1985]. Another characteristic of application domains is that its scope tends to increase with the passing of time. Among other reasons, increments may occur due to new capabilities offered by new technology, or due to the integration of several application domains into a larger encompassing one. All these changes imply non trivial modifications of the affected software.

We call *artifact domain* the set of artifacts such as test scripts, test scaffolds, specifications, architectures, designs, and specialized tools created and used while developing and maintaining[2] software. We use the term *artifact* for all *work-products*, and *products* that are needed to *use and maintain* the target software. Though some artifacts of the artifact domain are needed only while the software is being developed (e.g. mock objects, stubs, and simulators for some kind of hardware) several other artifacts are necessary to facilitate maintenance (e.g. architecture and design documents, test drivers, test scripts). In order to assure maintainability these latter artifacts must be co-evolved as long as the software is used. Hence, they should be available and must be evolved over time [Arisholm et al, 2006].

We call *technology domain* the set of representations (e.g. executable code, XML files, and scripts), representation languages (e.g. programming languages, scripting languages), software tools (e.g. libraries, frameworks), software and hardware platforms (e.g. data base management systems, and processor types), required by the target software to *provide its service*. Software is developed envisaging a given technology domain. As time passes the technology domain may change (e.g. the hardware and/or operating systems evolve), consequently, to continue providing the desired service, the target software and artifacts must be adapted to these changes. In many cases the technology domain evolves dramatically during the life-time of the software, possibly rendering the software useless, although its service is still needed. Many software systems are critical for the enterprises that use them, hence while the enterprise exists the service provided by the software must be available too, independently of whether the technology domain has changed or not. Thus, the artifact domain must be kept coherent with the technology domain while the latter evolves over time. Presently such evolution is often performed rewriting the code or encapsulating it in some kind of wrapper. However, not only code must be changed, but also all persistent data stores.

We call *environment domain* or *development environment* the set of representations (e.g. requirement specifications, architecture and other design documents)

[2] We will use the term *maintenance* to denote all kinds of changes after the software has been delivered, i.e. corrective, adaptive, perfective maintenance as well as evolution.

representation languages (e.g. specification and design languages), software tools (e.g. editors, integrated development environments), development techniques and practices[3] employed while *developing and maintaining* the software supporting artifacts. Independently of the maturity of an organization, software is developed and maintained using a given environment domain. Maintaining software requires the availability of at least part of the environment domain used to develop it. However, as time passes the environment domain will change; consequently the previous domains must possibly be adapted to these changes too.

We call *communication domain* the tools and languages needed to establish proper communication between users or clients and the development team. Not necessarily all of these persons are conversant with software engineering languages and tools. When considering a long lived system it is expected that people involved with its maintenance will be replaced. Hence, documentation must exist and must assure proper communication between people that do not know each other. To increase the efficacy of the communication, domain specific languages, or dialects of existing languages, are frequently developed. Such languages are geared towards the user, or *specifier*. However, the artifacts written in these languages must later on be translated into standard software engineering artifacts.

The need to adapt a software environment goes far beyond of choosing among existing representation languages. The new languages and dialects aim at reducing the amount of defects injected into the software due to incorrect communication. Since the people using the target software and the developers that maintain it are substituted during the life-time of the target software, it is expected that the communication domain also evolves over time. The communication domain involves also systems and their users. It is well known that inadequate human interfaces and system context expectancies are one of the major causes of human errors [Reason , 2003]. Hence, it is expected that the target system is changed to improve human interfaces, as well as being adapted to present better human interfaces.

These domains are not orthogonal, instead they are quite interdependent. Furthermore, they evolve over time in a non predictable way. Due to the interdependence, if an artifact in a given domain is changed, other artifacts in the other domains may have to be changed too. On the other hand, it is almost always impossible to create a design that remains unchanged over the life-time of the software. Due to this it is important that maintenance be taken into account while designing and developing the software. No doing so may lead to debts, that is, costs that will show up at later times [Sterling, 2011]. Furthermore, software maintenance is known to deteriorate the structure of the software [Eick et al, 2001; Tvedt et al, 2002; Hochstein and Lindvall, 2004]. Hence, means should be available to reorganize, or better to *reengineer the software*, whenever its architecture becomes inadequate.

Concluding, following interdependent domains must be taken into account when establishing a development and maintenance environment:

1. *Application domain* – establishes the functional and non functional requirements that the software should satisfy to provide an adequate and dependable service for the user. Unfortunately there is often a difference between what the user desires and what the software provides, i.e. the *service of the software*.

[3] We use the term *practice* to denote some human action that should be performed in some prescribed way. Very often, though, artifacts are developed in an ad hoc way, not using defined practices.

2. *Artifact domain* – establishes the set of artifacts (work products) that are needed to provide the service as well as to support the maintenance of the target system.
3. *Technology domain* – establishes the technology on which the target software is based while in use (e.g. languages used to write the artifacts).
4. *Environment domain* – establishes the technology used to develop and maintain the artifacts of the target system (e.g. tools and processes used to develop and maintain).
5. *Communication domain* – establishes the communication needs and restrictions considering the humans that develop, maintain and use the software.

Using a crude analogy, the application domain is the purpose for which a bridge has been built. The artifact domain contains the walkways, ladders, scaffolds, design documents and tools necessary to maintain the bridge. The technology domain is composed of the materials used to construct the bridge, the embedded maintenance support and the maintenance manuals. The environment domain is composed of the engineering techniques, equipment and tools that have been employed or developed to create the artifacts used to build the bridge. The communication domain involves the documents that are used by customers (e.g. transportation system architects), client engineers, construction engineers and workers to describe the bridge to be built and later maintained.

1.2 Development models

Looking from another perspective, software may be considered to be a form of knowledge [Armour, 2003]. Thus, its development process would be a knowledge acquisition process. While using, developing and maintaining software, one acquires knowledge about the application domain as well as about the other domains. Due to insufficient knowledge at the onset of the development, it is highly improbable that specifications, architecture, designs and other artifacts remain unchanged during the software's initial development [Berry et al, 2010; Kemerer and Slaughter, 1999].

All of the different artifacts that compose software are highly interdependent and constitute a hyper-document. As in other engineering fields, the collection of artifacts (e.g. specifications, design, code, tests, etc.) should be kept consistent one with the other. Hence, whenever some element of any of the five domains is changed, other elements must possibly be co-evolved to reestablish consistency. To be of practical use it should be possible to verify consistency at a low cost and also to help developers to reestablish consistency whenever problems are detected.

It is well known that software maintenance is inevitable [Beck, 2010; Lehman, 1996], even while the software is being developed [Berry et al, 2010]. As already mentioned, it has also been observed that the software architecture deteriorates as maintenance is performed [Eick et al, 2001; Tvedt et al, 2002; Hochstein and Lindvall, 2004]. Finally, it has been observed that shortcuts are often taken in order to satisfy contract goals such as schedule and cost. Often these shortcuts lead to inadequately structured artifacts incurring in unnecessarily increased maintenance costs [Sterling, 2011; Glass, 2003]. Unfortunately these shortcuts may persist in the delivered code, polluting its architecture and design. Thus, preventive maintenance [Kajko-Mattson, 2000] and reengineering is expected to occur, aiming at restructuring the architecture and design of the software [Fowler, 2000] and consequently increasing the ability for it to remain useful over long periods of time. However, preventive maintenance is seldom

performed due to its costs and the difficulty to justify the return of investment. A consequence of this discussion is that software should be designed in such a way as to facilitate its evolution considering all five domains.

A major reason for the difficulty of developing and maintaining software is its inherent complexity [Brooks, 1987]. Software is highly abstract in the sense that it is essentially the prescription of the behavior that one or more interacting automata should display when executing these prescriptions. Specifiers, designers and implementers must imagine how the software will behave while being developed. As technological capabilities increase more difficult it becomes for humans to imagine all facets and consequences of this behavior. Hence, effective tools are needed to specify, model, integrate and control different views of the software. A variety of views is necessary, since the complexity of the software often extrapolates the capability of humans to fully understand its specification and design. This understanding is necessary to enable developers to write and maintain adequate and dependable code. However, this approach may lead to the "blind men and an elephant" syndrome. Finally, a set of tools or practices is required to transform specifications into code, user documents and other artifacts that are required to enable users to properly interact with the software.

Since software possibly contains defects and almost certainly will evolve during its life-time, the development environment must support not only maintenance, but also reverse engineering, reengineering and refactoring. These practices should be supported assuring that they are cost effective as well as require short time spans to be properly performed. The relevance of these practices grows as the expected life-time, complexity and risk grow. Purely manual practices are not sufficient since they often do not reduce costs nor do they assure sufficient quality. Hence, specialized software tools become more and more necessary as software complexity and usage risks increase. Hardly the same set of tools will provide adequate support for a large majority of software being developed and maintained. In other words, the development environment containing a set of tools used to specify, model, design, inspect, test and other activities must be adaptable to the needs of specific software systems.

In addition to supporting openness in the face of constant change, the development environments must conform to the other domains of the target software. For example, the environment used for developing a large command and control system significantly differs from the environment required to develop a simple information system. Thus, development environments must not only adapt to some technology domain, they must also adapt to the target application domains. Also here it should be expected that in many cases this domain may change during the life-time of the software.

1.3 Terminology

In this section we introduce several terms that will be used throughout this document. Later many of them will be described in depth.

The artifacts composing the software being developed or maintained are stored in a repository. This repository contains *hyper-objects*. Artifacts are built composing attributes of hyper-objects according to a prescribed rule, the representation language of the artifact. Many representation languages are graphic. A diagram is a hyper-object; however, the graphical elements of a diagram are similar to hyper-objects too in

the sense that these elements contain a well defined set of attributes that are specific for each kind of element. Hence, hyper-objects may contain zero or more *sub-objects*. Hyper-objects and sub-objects are instances of *hyper-classes*. Only one sub-object level is supported. Deeper structures can be achieved using recursive relations, such as *composition* and *decomposition*, or *specialization* and *generalization*. In fact, a sub-object could be viewed as a named and well characterized sub-domain of a given hyper-object's attributes. For example, a diagram (a hyper-object) contains several attributes for each of the items (boxes, links, labels, adornments) that compose this diagram. Each of these items corresponds to a sub-object and may refer to another hyper-object that contains its descriptive details. For example, in an UML **`class diagram`** boxes (sub-objects) refer to the **`classes`** (hyper-objects) that contain the facts (e.g. specifications, interfaces, attributes, methods) of these **`classes`**. Conversely, each **`class`** type hyper-object should refer to all places where it appears as a sub-object in some diagram. This latter relation allows navigating from any occurrence of a class to any other occurrence in a fashion similar to that of hyper-documents.

Artifact – is any tangible result (work product) of the development or maintenance of the target software. Artifacts may be composed of other artifacts. Artifacts may contain several different representations. Examples of artifacts are: documentation files; compilable program modules; findings documents generated while performing some quality control activity. Example of an artifact composed of several representations is a design document containing a data flow diagram and the specifications of the elements that appear in this diagram.

Definition base – is a special purpose data base that contains interpretable descriptions of representation languages and instantiations of meta-tools.

Environment – is an interconnected collection of tools, representation languages and practices that is used to develop or maintain the target software.

Fact – is any elementary data used in some representation. Examples of facts are: boxes, edges and labels contained in some diagram; code fragments (e.g. statement lists) that are part of some algorithm; declarations of methods; composition lists of some document. In SQL terminology a fact would be a line of a table, however not all data is stored in a SQL compatible fashion.

Hyper-class – is a class like element from which hyper-objects and sub-objects can be instantiated.

Hyper-object – is an object like element that is persisted in a software base. A hyper-object contains several attributes, each of which corresponds to an object in an object oriented programming language.

Maintenance – is any activity that changes part of the target software aiming at eliminating defects (correction); adapting to new context conditions; adding or improving functionalities or quality characteristics; reorganize the software for the purpose of easier future maintenance.

Meta tool – is a generic tool that can be instantiated to perform a specific task. For example, the diagram meta-editor can be instantiated to edit a variety of different diagram representation languages.

Repository – is a collection of software bases and files that contain all facts of one or more target systems.

Representation – is any document or part thereof that describes some aspect of the target software. Examples are diagrams, architecture or design descriptions, and code.

Representation languages – are natural or artificial languages used to write a given representation. Each representation is written in a specific language. Examples are: UML class diagrams; entity relationship diagrams; C++; Java; make scripts; JUnit scripts.

Software base – is a special purpose database used to store facts of the target software or part thereof.

Target software – is the software, component, of framework that is the object of development or maintenance activities.

Tool – is any software, component or framework that is used for the purpose of developing or maintaining the target software.

1.4 Overview of this report

In section *2 Talisman version 4.4 functionalities* we will provide an overview of several of Talisman's version 4.4 capabilities. We will also describe several of the problems identified while using that system. Solving these problems will be one of the aims of the new version.

In section *3 Functional requirements* we will describe some of the services provided by software engineering environments instantiated with Talisman. We will use a narrative style instead of a more formal style frequently found in requirements documents.

In section *4 Architectural aspects* we will describe several architectural aspects of the Talisman meta-environment. The architecture of a meta-environment must be process, representation language and tool independent. Data contained in a definition base establish the specific behavior required by a specific instance of the meta-environment. This organization allows the construction of a large set of environments, without needing to reprogram any of Talisman's components.

Finally, in section *5 Concluding remarks* we present a brief wrap up of this report.

2 Talisman version 4.4 functionalities

In this section we will provide an overview of several of Talisman's version 4.4 capabilities. We will also describe several of the problems identified while using that system. Solving these problems will be one of the aims of the new version.

The development of the Talisman Software Engineering Meta-Environment started in 1987, and its version 4.2 was finished in 1992 [Staa, 1993]. It was developed for a PC-XT running MS-DOS. Several small modifications were made that lead to its last version 4.4 finished September 1995. The following are some of Talisman's 4.4 features:

- It is an integrated *meta-environment* prototype supporting several *meta-tools* required for developing software such as: creating and editing composite text, diagrams, and structure charts. It also provides means to transform, verify, import and export representations, as well as to generate code or other textual files.
- Representation languages can be dynamically adapted without destroying or losing already performed development. New representation languages can be added at any time. New representation languages editors based on some kind of diagram can usually be created in less than two days. Model checking tools, transformation tools and generators may take more time due to the intrinsic complexity of the new language.
- The meta-environment is instantiated by means of a *definition base*. This is a special purpose data base containing a set of binary tables that direct how the meta-tools should operate. The definition base is created from a set of interdependent specifications contained in several text tables.
- The meta-environment has been instantiated for a set of representation language families such as: organization analysis, goal specification, requirements specification, data flow based design, entity relationship based design, state transition graph design, modular design and implementation, work break down based planning, and document generation. It has been successfully used in several industrial projects.
- After a bootstrap version (version 2) had been built, all module design and implementation files were imported to Talisman repositories (see below), allowing to continue development using Talisman itself. Presently all of the modules which comprise the Talisman 4.4 system are composed using the contents of Talisman repositories.
- All tools interact via a *repository*. This repository is a special purpose object oriented data store. It stores lists containing *hyper-object* attributes of *small granularity*. *Hyper-objects* correspond to the syntactic elements of representation languages, and attributes correspond to conventional objects. Examples are: classes, methods, code blocks, states, transitions, processes, data elements and data stores. Coarse granularity objects, or better *representations*, are recomposed and then rendered whenever they are accessed. Examples of coarse granularity objects are: source code files, data flow diagrams, entity relationship diagrams, state transition diagrams, structure charts and user documents. Representations are built by selecting and formatting attributes contained in the repository.

- It is capable of generating many different kinds of text that could be used as input to some other tool. Furthermore, it is possible to import data from other tools, as long as it is contained in a text file adhering to the import syntax. This usually requires the development of a specialized conversion tool. This approach has been used to reverse engineer C code to structured design. Unfortunately, Talisman 4.4 is not capable of importing diagrams.
- It uses an internal programming language, the *form programming language*. Among other uses, this language allows to define how to compose source code from the repository's contents, how to define model checkers capable of generating quality control reports, how to implement representation transformers used to convert from one representation to another.
- It allows linking hyper-objects using a large variety of relations. As a result it is possible to establish a trace trail from requirements items to code fragments and vice-versa.
- Using its internal programming language, code composers can be built. These are programs capable of exploring (navigating through) diagrams and other content of the software base, picking some of the attributes (code fragments), composing them intermixed with generated code and producing a sequential text. If desired this text may be sent to a file, or to the composite text meta-editor. Code composers may generate files that can be sent to other tools. For example the composer may generate compilable code, or input for other text processing software. When sending the composed code to the composite editor, the attributes (text fragments) extracted from the software base can be edited. The visual aspect of the text is quite similar to the one a programmer expects to see. When closing the edited document, each modified fragment will be persisted in its proper place in the repository.
- It allows building code generators that use diagrams as input. Hence, part of the representations, especially code, can be maintained and generated using diagrams. This reduces development effort since a significant part of the code is correctly derived from the models represented by the diagrams. Furthermore, it allows to develop programs incrementally and to maintain programs editing design diagrams. Using entity relationship diagrams, we have successfully generated working Web systems containing more than 100.000 lines of correct Java code [Franca, 2000].
- The set of meta-editors (composite text meta-editor, structure chart meta-editor and diagram meta-editor) are all definition base driven. This allows for the addition and evolution of representation languages without having to change the code of the meta-environment itself. Furthermore, existing representations are usually not affected when such additions or changes are made.
- The set of editors provide a very powerful hyper-document navigation capability. For example, given that the current position in a composite text refers to some fragment, it is possible to navigate to a diagram that contains an instance of the hyper-object that contains this fragment. Alternatively, it is possible to navigate to another composite text that contains the specification (or other text) of the hyper-object that contains the fragment.
- The Talisman 4.4 engine and supporting tools were developed using an approach similar to contract driven development. Structure verifiers [Staa, 2000] have been developed for all elaborate data structures, in particular for the

repository structure. These verifiers run in a multi-programmed fashion, monitoring the development on the go (over the shoulder). Since Talisman was developed using itself as a development tool, most of the testing was done using the system and relying on the power of verification tools. This approach proved to be very successful and has contributed to a substantial increase of productivity when compared to industrial benchmarks available at development time. A similar approach has been used successfully to develop supervisory systems [Magalhães, 2009]

The experience gained with the development and use of Talisman 4.4 in many projects led to the proposal and development of this version 5 of Talisman. Among the problems observed we can mention:

- For the meta-environment approach to be successful a meta-programming environment is required. The instantiation of the meta-environment using a collection of interdependent tables is very cumbersome and error prone. However, meta-environment programming could be performed using an instance of the meta-environment itself. Thus a bootstrap period will be required until the meta-environment development tool is available. This two step approach must be well designed to reduce startup effort.
- Meta-programming should be partially rule based – aiming at lexical and syntactical aspects – and partially based on some procedural or object oriented programming language – aimed at semantic aspects.
- To allow more control when evolving a representation, version control should not be performed by a separate textual version control tool, but should be structural [Araújo, 2010] and should be part of the environment itself [Pietrobon, 1995]. Structural differences could be stored as attributes of the affected hyper-objects. This structural difference data allows building tools that selectively propagate changes to related representations.
- Diagrams should be considered first class hyper-objects. Hence, each diagram can be handled as a unit. Diagram hyper-objects should maintain an interface descriptor. This descriptor allows interface items to be associated with items contained in other diagrams or with other hyper-objects. This would provide means to establish interdependencies among the several diagrams that compose a system. It also provides means to perform model checking over the boundaries of a specific diagram, as well as transformations involving diagrams.
- It should be possible to export and import diagrams from XMI (or similar) files. Currently Talisman 4.4 exports and imports name, string, text and relation attributes, but not graphical ones.
- It should be possible to generate diagrams by means of some transformation operator. The placement of the diagram elements should be performed by some algorithm and, if necessary, hand improved afterwards. This transformation would facilitate reverse engineering diagrams. It would also provide means to reuse many architectural and design elements or to verify the coherence between textual and diagrammatic representations. Finally, it would allow developing or maintaining a complex system using several representations (development steps) each of which leading to a lower abstraction level or involving a variety of viewpoints.

- A consequence of being able to generate diagrams is the possibility to cut part of an existing diagram and move it to a new diagram or to another existing diagram. It allows also composing a larger diagram from elements contained in other diagrams. While moving or creating a new diagram from parts of existing ones, many of the relations between diagrams should be created automatically. This should contribute positively to the use of model driven development, reengineering and maintenance based on existing models.
- A large integrated repository containing all systems of an organization[4] creates risks and several problems such as: difficulty to share components among different organizations and difficulty to transfer components to other organizations. On the other hand, partitioning the overall repository into several independent repositories creates other problems such as: difficulty to establish verification tools that cross boundaries of the elementary repositories. It also hampers maintenance since several copies of an elementary repository might exist. A possible way to solve these problems is to partition the overall repository into several *interdependent* repositories.
- Hardly a single meta-environment will provide support for all possible development tools. Hence an effective means to interoperate with other tools is needed, such as plug-ins. The internal programming language provides means to specialize the tool for specific purposes. For example, navigation rules, composite text construction rules. This internal programming language should be an extended version of a general purpose language. We are considering to use Lua [Ierusalimschy, 2004], since this language is a very efficient and powerful extension language and also contains a powerful library of pattern matching functions.
- A large part of the software development effort is directed to maintaining existing legacy systems. Many of these systems suffer from inadequate architecture and design. However, these systems tend to be critical for the enterprise owning them. Hence, an effective way to reverse engineer and later on reengineer these systems might be an adequate form to increase life-expectancy of legacy systems.
- A training infrastructure must be available. Environments that use model driven development and maintenance require a fair deal of expertise in modeling since. Due to the use of representation transformers, they end up being part of the coding effort. Users with too little proficiency will find it much harder to work with such an environment than with the traditional integrated development environments that focus on programming in a given set of programming languages [Parker, 2001]. Furthermore, the aim of a meta-environment is allowing the creation of domain specific languages and the necessary transformers. Again a fair deal of expertise is needed to develop these artifacts.

[4] An organization could be as small as a Scrum development team and as large as a software house.

3 Functional requirements

In this section, we will describe some of the services to be provided by software engineering environments instantiated with Talisman. We will use a narrative style instead of a more formal style frequently found in requirements documents.

Talisman 5 is a meta-environment that provides meta-tools to create specific environment workstations. Environment workstations are used to develop or maintain artifacts that compose software systems. Artifacts may be diagrams, composite text, text files and others. The data from which artifacts are rendered are kept in a repository. The repository is composed of one or more special purpose object oriented data bases. Instantiation data is kept in a definition base, which is also an object oriented data base.

3.1 Software engineering environment user roles

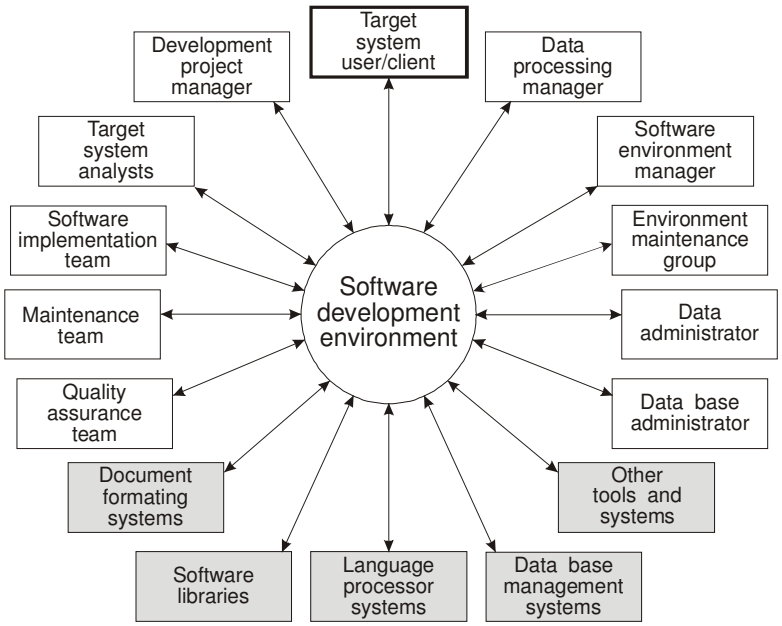

Figure 2. Typical environment user roles

Figure 2 depicts some of a wide variety of *environment user roles*. These roles may be enacted by humans, or may be performed by tools (shaded boxes in the figure). A same person may enact several of these roles. In contrast, a given role may be enacted by several individuals. Since the roles depicted are fairly well known, we will not describe them in more depth here.

As mentioned earlier software environments must adapt to the needs of specific environment users, to the technology domain and also to the application domain of the target system. However, it is virtually impossible to identify the set of all possible user roles, since new or slightly changed user roles could be required when developing or maintaining a specific system. The environment must then be adaptable to a variety of user roles and to persons enacting these roles, where the set of possible roles is not known when building the meta-environment. Given the difficulty of

defining user roles a priori, Talisman does not impose a set of predefined user roles. Rather, it allows creating and improving roles, as well as role supporting representation languages and tools, assuring a harmonious development environment.

Software systems must provide adequate and dependable support for the enactment of each user role. The goal of every environment user role is to contribute in a defined way to the effective development or maintenance of one or more artifacts all of acceptable quality. As shown in figure 1 the goal of an environment user is not just using the environment. Instead it is to perform the role's duties with the aid of the environment in a productive and dependable way. Thus the environment must provide an adequate set of tools and representation languages for each of the user roles. Furthermore, user interfaces should be adjustable not only to the needs of each role and but also to the needs of the person enacting the role.

User roles are interdependent. For example, database standards established by the database administrator must be strictly followed by at least the development teams, the maintenance teams and the quality assurance groups. Thus software environments must provide efficient and effective means to support communication among the different environment users. Much of this communication relies on documents, i.e. representations. Even when informally discussing aspects of some target system, several representations are used to support this discussion. Some of these representations are ad hoc, but usually are quite expressive. Furthermore, many issues of such discussions can receive some automated support. For example model checkers and design evaluation tools can identify design anomalies, reducing the amount of work spent in reviewing or, even worse, in useless rework [Macía, 2009]. In order to establish communication among environment users, Talisman allows the assembly of a variety of different representations from the facts contained in the repository. To fine-tune quality control to the needs of a specific project, model checkers and measuring tools can be developed or adapted.

3.2 Software development activities

Developing a system with Talisman corresponds to *populating* and *updating* the repository with facts. The order in which the repository may be populated should be process independent. This assures adaptability of the meta-environment to a variety of specific development processes.

Of course, when following a well-defined development process, the amount of possible inconsistencies among representations will be small and could be removed soon after a representation has been built or changed. Consequently fewer inconsistencies will be detectable when performing quality control, less corrective effort will be necessary in order to assure consistency among a collection of representations. Furthermore, fewer defects will remain in the target system. However, imposing a predefined work order often hampers development due to unnecessary bureaucracy.

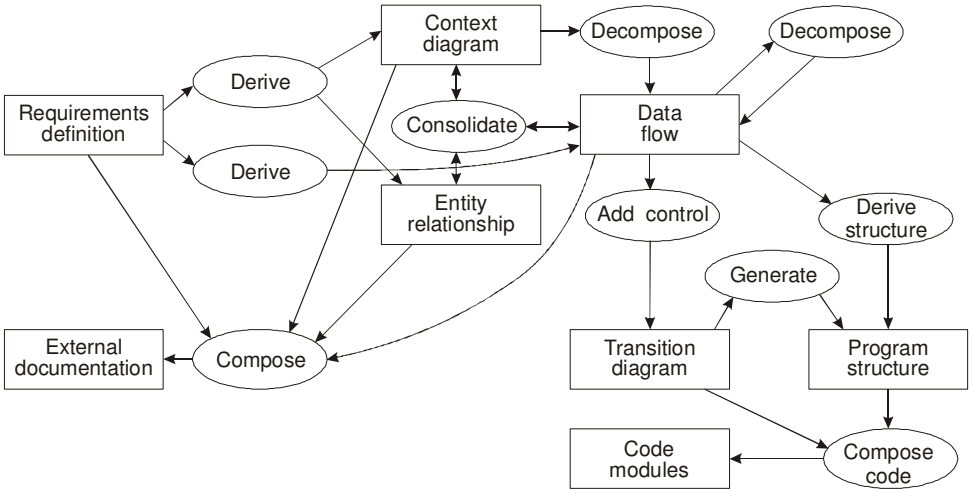

Figure 3. A partial language transformation graph

The set of available representation languages can be organized as a directed graph, see Figure 3. The picture shows the interdependencies of data flow, entity relationship, transition diagrams, structure charts and code. In this graph representation languages are the rectangles, whereas the ellipses identify the transformation from a *predecessor representation language* to a *successor representation language*. This transformation is applied when transforming a representation written in the predecessor language into another representation written in the successor language. Transformations may require several sources, or produce several results. For example, to compose documentation for the user it may be necessary to extract facts from several representations such as requirements, entity relationship and dataflow diagrams. Since the set of diagrams must describe exactly the same system the contents of several artifacts may have to be consolidated. For example entity relationship and dataflow diagrams must pass joint model checking operations.

The *language transformation graph* defines part of a development process. In particular it defines the pre- and post-conditions, i.e. representations in a given representation language, which specify each transformation. There may be more than one transformation from a given representation language to another one. For example, one might compose code from transition diagrams as well as from program structure definitions. Any specific process using these representation languages and transformations should obey the constraints established by the language transformation graph.

However, the graph could be traversed in an inverse order. In this case the transformations correspond to reverse engineering like operations. Such operations are necessary when maintaining legacy systems. Their availability assures that the development may be performed at any point and propagated back or forth as needed [Antkiewicz, 2006].

In one extreme, a transformation could be fully automatic — a *total transformation*. For example, a compiler is a total transformation. In this case the result of the transformation does not need any additional information in order to build an adequate and dependable artifact. In the other extreme, transformations might require intensive human participation — a *manual transformation*. In this case there is only some text telling people how to perform the transformation. For example, when transforming textual requirement definitions into system specifications, usually a purely manual

transformation is performed. Most transformations fall somewhere in between these two extremes — a *partial transformation*. In these cases, part of the transformation can be performed automatically, producing a partially completed representation. Completing the representation requires human intervention. This kind of transformation allows a good separation of abstraction level concerns, but introduces maintenance problems, since the added information might be lost whenever the representation is changed and than transformed again.

The ability to establish partial transformations is important when considering step-wise development, since it allows data that is germane to a given level of abstraction to be added only when a representation at this very level is being filled out. Hence, high level abstraction representations need not to be polluted with low level of abstraction data, since these will be added only when a representation at the corresponding abstraction level is completed. This editing is preceded by one or more transformation steps from the higher level of abstraction representation to the lower level one. However, to implement such representations requires attention to how maintenance and reverse engineering operations are performed.

When transforming a representation written in some language to another representation, several cross-references involving these representations are established. These cross-references define the information dependencies of facts contained in the two representations. These cross-references establish transformation trails that allow an inspector to verify where requirements have been reified, as well as given a code fragment identify the reason why it has been written.

When editing a lower level of abstraction representation, elements derived from higher level representations may be edited. This induces loss of consistency between the two representations. Talisman uses point-wise propagation due to the use of small-grained interrelated hyper-object attributes. When modifying a given representation, all existing relations to hyper-object attributes contained in some derived representation are kept and remain correct, except, of course, for those affected by the modification. However, it is expected that only few of these are modified. When navigating back to the higher level representation, these changes will be shown, allowing to adjust the representation in order to be defect free. Once these changes have been performed, forward transformation may be performed again, which will only affect the elements derived from the changed parts of the higher level representation. All these changes are under structural version control embedded in the Talisman repository. It is expected that such changes ripple through the collection of artifacts ebbing out once this collection returns to a consistent state.

3.3 Representation languages

Every representation is written in some *representation language*. Each representation language defines:

1. *Rendering formats* (pretty printing rules) involving properties such as shapes of graphic elements, or style sheets, fonts, and margins of textual elements.
2. *User interface*, defining how to create or modify representations written in the language.
3. *Syntax*, defining hyper-object classes and their attributes which can be manipulated and how these hyper-objects may be combined.

4. *Semantics*, defining the meaning of hyper-objects and of the relations between them. Semantics establish how representations may be verified, transformed into or combined with other representations.
5. *Composition*, defining how a specific representation is built from a starting hyper-object using the content of the repository.

Each representation language contains several *hyper-classes*. *Hyper-objects* of these classes will be created when designing a specific representation using this representation language. Hyper-classes relate to other hyper-classes or to themselves by means of a *relation*. All relations are bidirectional and establish cross-references between hyper-objects.

Considering a given representation language, its hyper-classes and their relations may be defined by means of a diagram such as the one shown in figure 4, which defines graphically a simplified version of the "*modular programming*" representation language. This language contains most of the C++ features. In this diagram boxes correspond to hyper-object classes and labeled edges correspond to relations between these classes. Labeled edges identify a pair of forward and inverse relations. All relations are *0..n* to *0..n*. Names of relations may be equal, as long as the source hyper-class is different.

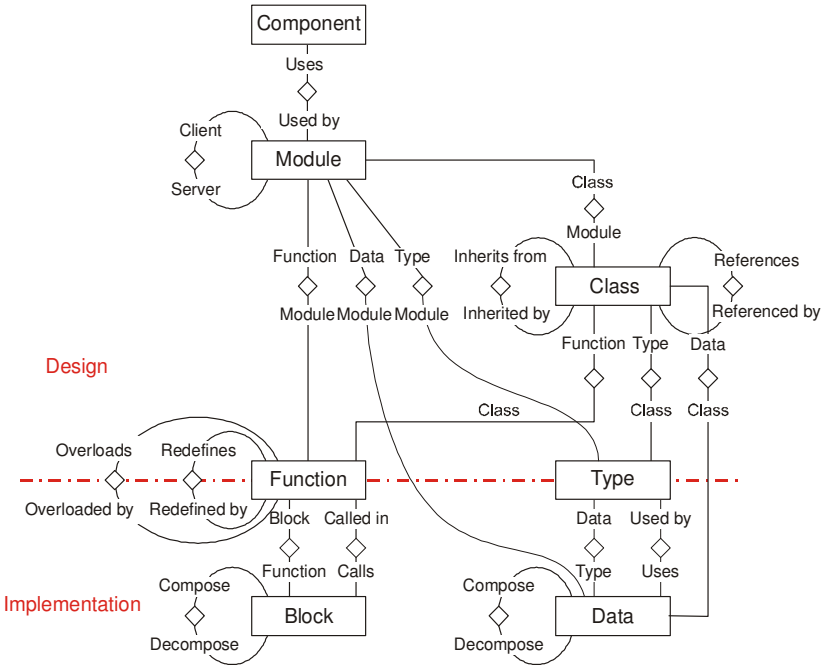

Figure 4. Example of a graphical representation language definition

Each hyper-class defines several *attributes* which its hyper-objects may contain. Hyper-object attributes correspond to objects in some object oriented programming language. For example: a "*Module*" class may contain an "*Informal description text*" and a "*Structural invariant specification text*". Both attributes are objects of the class "*Plain text fragment*". Talisman groups attributes into several *attribute categories*, such as "*names*", "*strings*", "*plain text fragments*"[5]. Once extracted, they are rendered according

[5] A text fragment is a text of unbounded size. In the case of code it corresponds usually to a statement list that is part of a method, but not necessarily corresponds to the whole

to the meta-editor rules being used. The extraction may be performed navigating over several hyper-objects accessible from the referred one.

The diagram used to model a representation language is written in a representation language in its own right. Hence it can be defined as a representation language contained in the repertoire of the representation languages supported by the meta-environment. This feature allows Talisman to define itself.

Talisman groups representation languages into:

1. Representation language families, and
2. Representation language categories.

A *representation language family* defines a collection of strongly interrelated representation languages. For example, the *DataFlow* representation language family contains among others the *DataFlowDiagram*, the *ProcessSpecification*, and the *DataStoreSpecification* representation languages.

Each *representation language category* is edited by a *meta-editor*. Each meta-editor handles all representation languages of the corresponding category. For example, the diagram meta-editor handles all representation languages based on diagrams; the structure meta-editor handles all representation languages using hierarchical structures; the composite text meta-editor handles textual representation languages. An example of a meta-editor is a diagram editor capable of editing any diagram containing nodes (boxes), hyper-nodes (a sub-graph) and edges (arrows). Meta-editors are instantiated for a specific representation language by means of descriptors contained in the definition base.

3.4 Form programs

Talisman builds and verifies all representations by means of *form programs*. These programs are developed by the environment builder and are stored in the environment base. Later on they are compiled and stored in a definition base. Each representation language may be bound to several form programs, each of which aimed at an activity such as: editing attributes, verifying properties, transforming a representation, composing code to be compiled and generating reports. Talisman always considers a specific hyper-object to be the current focal hyper-object. Form programs may explore, verify, transform and render representations departing from a focal hyper-object.

---

method. In the case of text it may correspond to one paragraph or part thereof. Considering the other extreme, a text fragment may correspond to a whole document.

```
BeginForm void EditModule( void )
   Title "Module name: " ;
   NoNewLine ;
   Name ;
   If Exists( [ Text ChangeRequestText ] )
   Then
      Title "Change requests" ;
      Text ChangeRequestText ;
   EndIf ;
   If Exists( [ Text ReportText ] )
   Then
      Title "Findings report" ;
      Text FindingsText ;
   EndIf ;
   Title "Module description:" ;
   Text DescriptionText ;
   Title "Functions exported by this module" ;
   Relation ExportsFunction ;
   Title "Functions contained in this module" ;
   Relation Functions ;
EndForm ;
```

Figure 5. Example of a specification form program fragment (Talisman 4.4)

Each specification form, when applied to an acceptable focal hyper-object, explores the repository and builds the representation view corresponding to this focal hyper-object. Different tools are defined by means of form programs. For each of the hyper-classes of a representation language, there must be several form programs, one for each of the macro-actions which may be triggered by the environment user. Examples of such macro-actions are *EditRepresentation*, *VerifyRepresentation*, and *TransformRepresentation*.

Figures 5 and 6 illustrate respectively a specification form and a small fragment of a simple-minded verifier. The language used in both examples is derived from the language used in version 4.4 Talisman. However, in version 5 the extension language Lua [Ierusalimschi, 2004] is being considered as the form programming language to be used.

```
BeginForm bool VerifyClass( void )
   If Not Exists( [ Relation Methods ] )
   Then
      Title "Class does not contain methods" ;
   EndIf ;
   If SelectPublic( [ Relation Methods ] ) == empty
   Then
      Title "Class does not contain public methods" ;
   EndIf ;
   ForAll Methods Do
      If Not Exists( [ Text PreConditionText ] )
      Then
         Title "Method " ;
         NoNewLine ;
         Name ;
         NoNewLine ;
         Title " does not define a pre-condition." ;
      EndIf ;
   EndForAll ;
EndForm
```

Figure 6. Example of a verification program

3.5 Representations and artifacts

When developing software, several *representations* are rendered, edited, verified, inspected, transformed and maintained. Examples of representations are: source code files, requirement specifications, design diagrams, algorithm structures, help files, target system user documents, and *make* or *Maven* files. Welsh and Han [1994] even depart from the premise that software development is a documentation-based process. These composite concrete representations usually consist of a number of elementary fragments such as text paragraphs, code fragments, diagram items and the relationships among them.

Artifacts are work-products. An artifact may contain several representations. For example, a design artifact may contain graphical and textual representations. Artifacts appear in some machine readable form and a basic goal of an artifact is to serve as a communication means between the different environment users, regardless of whether they are humans or software systems.

One may extend the concept of artifact to include several reports and work files. For example, the findings report generated while performing model checking could be considered as a special kind of artifact. Similarly *make* or *Maven* script files and test script files, such as *JUnit* files, could also be viewed as being some sort of artifact.

Artifacts may vary from the very large and complex, e.g. a user manual, to a very small and simple one, e.g. a data element specification. *Coarse grained representations* such as diagrams or program code, are usually composed of a large quantity of facts, whereas *fine grained representations* contain one or very few facts. Talisman stores very fine grained facts in the form of *hyper-object attributes* in its repositories. It always reconstructs coarse grained artifacts whenever they are accessed.

In order to display an artifact all its component representations will be displayed. To display a representation, i.e. a coarse grained object, a *form program* explores the repository, reading and formatting all attributes which compose the representation to be rendered. Talisman defines a representation by following tuple:

< *RLF*, *RLC* , *ACT*, *FP* , *OBJ* > where:

RLF identifies the representation language family to be used. The family allows bundling several form programs together, reducing the effort of configuring the specific environment.

RLC identifies the specific language category to be used.

ACT identifies the *action* that is being performed. Examples are: create a hyper-object, edit a representation, verify a representation, and transform a representation into some other one.

FP identifies the *form program* to be used. There may be more than form program for a given action, leaving to the environment user the choice of which one to use.

OBJ identifies the *focal hyper-object* from which the representation will be rendered.

As an example consider the editing of a data flow diagram and its corresponding data dictionary. In this case the representation language family is *DataFlow*. When editing a data flow diagram the language category is *Diagram*. When editing data dictionary

specifications the language category is *CompositeText*. When editing a data flow diagram using bubble charts, the form program selected by the environment builder would be *BubbleChart* [DeMarco, 1979]. However, the same diagram could be rendered using the conventions defined by Gane and Sarson [1978], in which case the form program selected would be *GaneSarson*.

Assume now that the user wants to see the data flow diagram corresponding to the decomposition of the process ***A-DFD***. In order to display the diagram the user selects the hyper-object ***A-DFD*** and triggers the action *ShowDiagram*. Now the identifiers *RLF=DataFlow* and *RLC=Diagram* are used to select the form program *BubbleChart*. Usually there will be just one form program. If there is more than one, the user is prompted for a choice. From the *BubbleChart* program, the function *EditDiagram* is selected. Now the repository will be accessed retrieving the facts that fit in the viewport and that are related to the diagram ***A-DFD***. These facts are stored in a workspace and then the diagram meta-editor is invoked to display and edit the diagram in an appropriate window.

If the user chooses to see and edit the specification of some process ***A-PR*** contained in the displayed diagram, she/he points to the object in the diagram and triggers the action *EditSpecification*. Now the identifiers *RLF=DataFlow* and *RLC=CompositeText* are used to select the form program *BubbleChart* and then the function *EditProcessSpecification* is invoked. Using this function the repository will be explored retrieving all facts related to the specification of the process ***A-PR***. These facts are stored in a workspace, the composite text meta-editor is invoked, which displays and edits the composite text.

The rendered representations may contain attributes of hyper-objects that are part of several different representations, independently of their family or category. For example the composite text of a *process* may contain attributes extracted from the data flow diagram, as well as from state transition diagrams, or even from other representation languages. For example, the representation could list all incoming flows, where they are coming from, and also tell what data flows on them. Furthermore, a process could be bound to a transition in a state transition diagram. It might be important for the developer to know aspects related to this transition. Once displayed, the user may edit this specification. When she/he closes the window containing the representation all changes recorded in the workspace are stored in the repository and, if necessary, a *RepositoryModified* message is issued. This message triggers the refreshing of all open windows (MVC pattern).

Artifacts are not generated just for the purpose of documenting aspects of the target system. They are an *active part* of the development and maintenance processes, guiding these processes. One of the goals of Talisman is to compose code from specification and design documents. For example, when using a state transition diagram to specify some user interface, one wants to be able to systematically transform this diagram into code structure skeletons. After this transformation, these skeletons are filled with code fragments. Then the diagram and code fragments are *linearized* (flattened, serialized) into compilable sequential code. When changing the diagram, independently of the amount and nature of changes, all derived elements which are not related to the changed portion remain as they are and still allow composing correctly the corresponding source code.

Talisman users may perform following operations on representations:

1. *addition*: This occurs when an artifact is written from scratch, or when an existing artifact is enlarged, or when a generated skeleton of an artifact is filled with

information. For example, when drawing a state transition diagram that refers to a process already defined in a data flow diagram one can search and select this process to insert in the state transition diagram. Afterwards one can fill information that is germane to the diagrams in which the object is referred to. The composite text corresponding to the process may contain all fragments added considering all places where this process is referred to. State transition diagrams can be converted into compilable code. To allow this the elements contained in the diagram must possibly contain code fragments. When the diagram is changed, all these code fragments remain attached to the diagram elements. Thus, all portions of the diagram that is not affected by the change will yield the same code text as before of the change. Diagrams may be converted to editable composite texts. If a text fragment in this composite text is changed, the corresponding place of this fragment is changed in all diagrams as a consequence of the fact that Talisman refers to fragments contained in the repository instead of copying them into the diagram.

2. *extraction*: This occurs when an artifact is displayed, when a hard-copy is generated, or when a file is generated to be used by some foreign tool. Representation languages should convey the writer's intent and should be understandable by the reader, even if this is some other software. Extraction is performed by form programs, which may explore the repository in any way they find necessary.

3. *modification*: This occurs when an artifact is updated. Typically, a modification of an already accepted fact leads to a new version of this fact. However, modifications of facts which have already been modified but not yet accepted will not lead to a new version of this fact.

4. *propagation*: This occurs when a consequent artifact is generated or changed in conformance to a given antecedent artifact. A consequent artifact is an artifact which, in accordance to the normal work flow, is to be created after the antecedent artifact. Propagation does not affect already existing facts in the consequent representation. In fact propagation maintains and possibly establishes references to these facts. Propagation is performed by form programs, which are capable of generating new content in the repository. Again these programs may explore the repository and create fragments or even new hyper-objects whenever and wherever needed.

5. *reverse propagation*: This occurs when an antecedent artifact is built or adapted to conform to a given consequent artifact. Reverse propagations are usually performed during reverse engineering activities. Propagations and reverse propagations are *transformations*.

6. *search*: This occurs when the environment user examines the repository in order to find a set of hyper-objects satisfying some search criteria. Since all facts are stored in the repository, it is possible to create form programs that explore the repository searching for some text and/or property. The result is a list that might be displayed and used to navigate or explore starting at one of the selected desired objects.

7. *reuse*: This occurs when the environment user incorporates an already existing fact, or even artifact, as part of some artifact being built or modified. Talisman promotes as-is reuse relating artifacts to reused artifacts.

8. *composition*: This occurs when several artifacts, elementary or not, are combined to form a composite artifact. Conversely, *decomposition* occurs when disassembling an artifact into several other artifacts.

3.6 Navigating over representations

As already mentioned, Talisman does not keep representations in its repositories, instead whenever necessary it reconstructs representations from some focal hyper-object applying to it a function contained in a form program belonging to a representation language category and family.

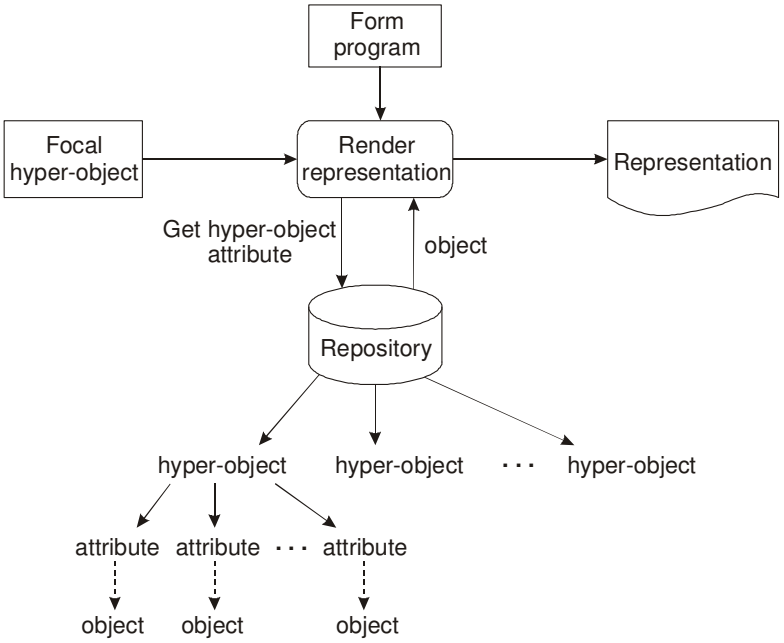

Figure 7. Rendering architecture

Whenever a representation is required, the facts that compose it are retrieved from the repository and then rendered. If a required datum is not found, the form program can render default data or error messages allowing the environment user to correct these defects. Missing data may be due exploring partially completed work; due to incorrect design; or due to interrupted connection to another computer. Talisman assures continually structural correctness, but not semantic correctness, since this would easily lead to deadlocks. Hence, the repository should be validated upon user request. A consequence of this is that use, or exploration, of the repository is independent of the order in which the repository is populated during the development process. This is a desirable feature since it assures that the environment does not impose a fixed development process, allowing environment builders to tailor development processes to specific needs.

The set of all representations corresponds to a hyper-document [Bigelow, 1988; Conklin, 1987]. Whenever a representation containing a given *focal hyper-object* is to be displayed or manipulated it is rendered using the form program of the target representation language applied to this hyper-object. This assures that representations always reflect all changes made anywhere to one or more of its constituent facts. This capability of reconstructing representations is very important since hyper-objects and attributes, i.e. facts, may be shared by several representations, consequently may be

edited in several different contexts, and, also, a given representation may display several times the same attribute or hyper-object.

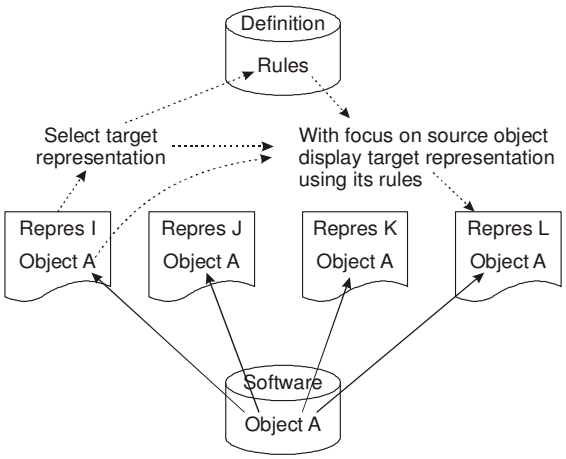

Figure 8. Hyper-document navigation

As shown in figure 7 rendering is performed by a form program associated with the target representation language and is applied to a focal hyper-object. While performing the rendering, the repository is explored in search of attributes that are returned as objects of the programming language used. As shown in figure 8 this allows to explore the set of representations navigating from one representation to another. Since rendering is governed by a form program, the content and complexity of the rendered representations as well as navigation details can be adjusted to the needs of the development stages and processes of a given project.

Talisman does not interrelate representations, instead it interrelates hyper-objects. This architecture supports a wide range of navigation facilities using explicit links, i.e. relations between hyper-objects or edges in a diagram. Since the construction rule applies to all artifacts regardless of their complexity, this architecture allows navigating between quite complex document sets. Finally, when editing or creating relations all navigation operations will adapt to these new relationships.

Different from many conventional hyper-text systems [Conklin, 1987], in Talisman links and frames have types, where these types convey semantic information. For example, when examining a specific requirement (a fact) in a requirements definition (a frame), one might want to see how this requirement influenced the system design. Thus, one may want to see all the entities (facts) in the set of data models (frames) which reify this requirement. In another occasion we might want to see all user interfaces (frames) which reify this same requirement.

3.7 Cross references

Representations contain *cross references* linking one representation to another. Cross references define semantic interdependencies. They are in fact relations involving hyper-objects. For example, the specification of a given **`class`** (a representation) should be cross referenced to all representations which make use of this **`class`**.

Cross references establish traceability involving several representations. Building such relations by hand is error prone and tedious; i.e. omissions and semantically inexistent relations might occur frequently. Thus some assistance should be provided

to at least partially generate and maintain the set of relations by automatic means. Talisman creates or updates relations as a response to user actions, or to editing, or to assignments performed by some form program.

Cross referenced representations must be *consistent*. That is, the information content of one representation must not conflict with the information content of the other representation linked by a cross-reference. In order to mechanically reduce the chance of inconsistencies Talisman uses transformers and verifiers. Given the facts of an antecedent representation, a transformer creates or updates facts of a corresponding consequent representation. Given a collection of representations, a verifier determines whether they are mutually consistent. Any discrepancy will be noted and stored in a defect attribute of the offending hyper-object.

3.8 Reuse

Reuse is assumed to be one of the most effective ways to reduce development effort. Talisman emphasizes specification, design and fragment reuse, as well as conventional class or function reuse. It provides two mechanisms for sharing facts:

1. Representation languages share hyper-classes; hence a given hyper-object may be part of several representations, as well as attributes of it may appear in several places in a same representation.
2. Representations are built while navigating over relations between hyper-objects.

The first mechanism allows hyper-objects of shared hyper-classes to appear in different representations (specification and design reuse). For example, assume that we are specifying a system using entity relationship diagrams (ERD) and data flow diagrams (DFD). Both the ERD representation language and the DFD language share the class *Record* (e.g. similar to a **`struct`** in C or a line in a SQL table). Thus a given entity, defined in an ERD, may contain records which in turn are contained in the DFD and appear in several data flows and data stores. These Records will later be complemented with code fragments allowing the composition of compilable code.

The second reuse mechanism is based on relations between hyper-objects. The related objects may belong to different hyper-classes. Relations permit form programs to navigate to hyper-objects belonging to any hyper-class even those which do not belong to the set of hyper-classes contained in the focal representation language. Continuing the same example, it is possible to establish a relation between a data store in a DFD and an entity in an ERD. Now when building the specification of the data store, it is possible to navigate to the ERD and extract relevant information to be displayed in the data store specification. It is also possible to perform model checking of both diagrams reporting inconsistencies that possibly occur in one of them. When correcting the shared element both diagrams are changed. Now both have to be checked again. Hence changes in one propagate to the other diagram.

These two mechanisms, together with the ability to share software bases and partially locking the contents of repositories against change, provide means to reuse standard specifications and designs. For example, a corporation could have defined a standard corporate data schema (see figure 17). This schema can now be used to build several different applications. Since the schema is protected, no application developer can accidentally or willfully change it. Furthermore, if the data administrator changes the data schema, all applications bound to it can be notified.

Different representations may have non empty intersections. This redundancy is a positive property since it increases the ability to control consistency among different representations, and it also makes it easier for readers to understand a complex system since several representations, i.e. views, can be used to understand the system. This redundancy is also necessary since different readers want to see the same information from different points of view. For example, the programmer of a function wants to see its source code. The user of this function wants to see the interface specification, the corresponding definition code, and some examples of its use. The user of an application containing this function wants to see help information relative to its specific operational aspects.

In Talisman artifacts share facts as a direct consequence of the way how they are built from the contents of the repository. Sharing facts between artifacts eliminates repetitive writing effort, since different artifacts can be generated from the repository containing these facts. For example, consider the development of a class library. There are at least five documents, the source code, the interface definition code, the test data files, the help files, and the user manual. All these documents share a large amount of information. Generating all these documents from a common base significantly reduces the inconsistencies which could build up whenever some maintenance is performed. It also contributes to the reduction of the co-evolution effort required to maintain consistency of all these representations.

In order create and maintain interrelated artifacts, Talisman:

- allows facts to be acquired and updated in any of the artifacts where they occur,
- provides verifiers to control and propagate changes, and
- provides fine-grained version and configuration control.

Obviously Talisman supports also the traditional copy and paste kind of reuse.

3.9 Quality assurance

When finishing the creation or modification of a representation its quality should be controlled. Whether performed by tools or humans quality control generates a *findings report*. This report contains error and warning messages relative to the controlled representation. Since the findings report is a fragment of some hyper-object, it might be bound to versions of the corresponding hyper-object. In Talisman text fragments related to the findings report are associated with the hyper-objects that expose the related defects. Thus, when controlling a representation, findings will be associated to the hyper-objects contained in this representation. The findings report of the representation corresponds to a composite text containing the set of findings texts of the hyper-objects contained in this representation.

Several causes may lead to defects found in a representation. However, every defective representation must eventually be modified in order to eliminate all causes of error messages and most of the causes of warning messages. In Figure 9, we show the set of actions and their interfaces which are performed when developing representations.

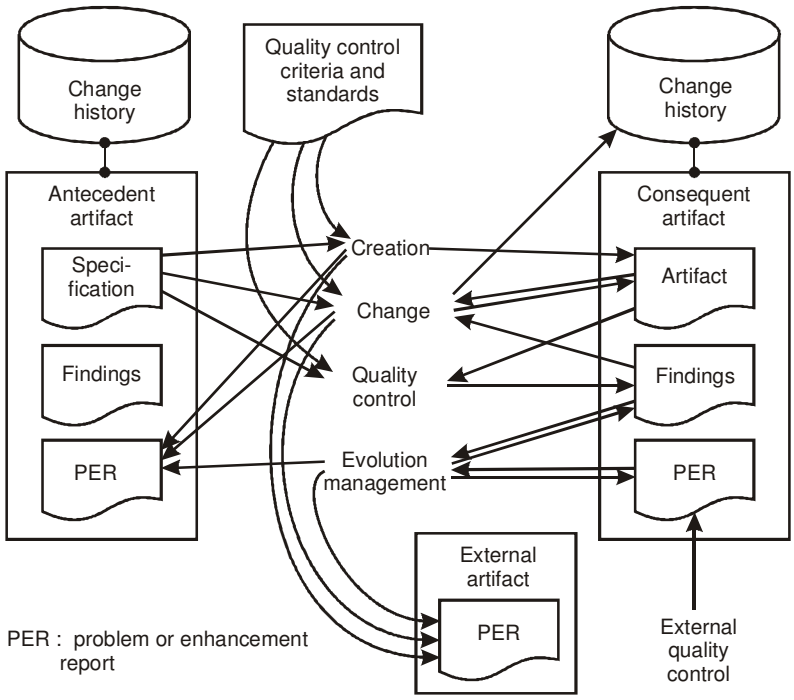

Figure 9. Representation development activities

When controlling the quality of a consequent representation, it is possible that defects are uncovered in some of its antecedent representations. For example, when implementing a class one may find that a method is missing in its specification. It is unreasonable to enforce a zero specification defect rule, such as "all implementations must be strictly consistent with their *original* specifications", since all sorts of errors and omissions may occur when writing the specification [Berry et al, 2010].

Enforcing specifications to be corrected immediately whenever a defect is uncovered may hamper work progress. On the other hand, letting quality control go unenforced is risky and may lead to large technical debts [Sterling, 2011]. Finally, letting a specification be changed by the person deriving the consequent artifact is a source of trouble, since specifications are often antecedents to several artifacts. Thus, due to the lack of uniformity when treating inconsistencies between specifications and their consequent representations, Talisman allows quality enforcement policies to be adjusted to the needs of the projects and progress states in these projects.

In Talisman antecedent and consequent representations may be under development at the same time. For example, when developing class diagrams, the detailed specification of classes and the diagrams may be created in a concurrent way. Thus it is unreasonable to expect that all antecedent specifications (the diagrams) have necessarily been accepted before starting the development of a consequent artifact (the class details). Instead, negotiation should be supported allowing the antecedent and consequent artifacts to evolve until they stabilize on a feasible and acceptable point.

Talisman allows attributes of a hyper-object to be *frozen*, i.e. locked. Once an attribute is frozen it may only be changed if it is first *unfrozen*. Freeze and unfreeze are configuration control operations similar to check-in and check-out available to the project manager. If a defect report is directed to a frozen attribute it will be kept as a problem or enhancement report, otherwise it will be kept as a findings report. In order

to achieve this, Talisman allows the existence of several findings texts and problem report texts associated with hyper-objects.

Changing an accepted artifact may impact several simultaneously ongoing activities. Furthermore, due to schedule pressure some changes and verifications may have to be delayed. Thus, changes may have to be postponed until an adequate opportunity arises. Thus, problems found in antecedent artifacts should be kept separate from conventional findings reports. These problems will be kept as *problem or enhancement reports* – PER – of the antecedent representation. As already mentioned, some problems found in the consequent artifact may have to be delayed. These PERs will be associated with the consequent artifact.

The set of PERs which affect a given artifact can be explored navigating through the artifact and displaying all findings found while visiting a constituent hyper-object. While a given activity is being performed, or when getting more insight about the problem to be solved, problems or suggestions may be uncovered that affect the whole project. These reports are added to the PER lists related to the project. Later on they while being analyzed, they can be transferred to the PER list of the affected artifact.

Creating or updating representations may take considerable time. During this time the representation will contain several defects due to incomplete or even incorrect modifications. Continually applying quality control tends to be more of nuisance than an effective aid, since defects will be reported for parts of the representation which the environment user knows are wrong. Thus, quality control should be applied at the environment user's discretion. On the other hand, sometimes the environment user may find it helpful to obtain findings reports even for still incomplete representations. For example, a findings report could be generated for a not yet finished consequent representation. Such a findings report may help to avoid specification errors that will later lead to rework. These reports will help her/him to complete and correct representations she/he is developing. In order to reduce noise, the quality control rules should be adaptable to the level of completion selected by the environment. This can be achieved selecting among the available verifiers the one which is most adequate considering the development stage. However, full acceptance will only be granted if the representation satisfies the specified quality requirements.

Talisman defines three levels of rules to be verified by tools:

- The first level contains rules which are applied before committing the result of an individual edit action to the workspace. Rules of this level may allow editors to validate properties such as "data stores may not be linked to other data stores".
- The second level contains rules which are applied before committing the workspace to the repository. Rules of this level may validate properties such as "text fragment must satisfy syntax X".
- The third level contains rules which are applied to part or all of the content of the repository. Rules of this level may validate statements such as "an entity must define at least one key data item". Rules belonging to the third level generate finding reports. These rules are applied upon user request.

The first two levels are handled while editing since they are context independent. They assure a minimal degree of syntactic and semantic integrity of the repository content. This degree depends on the control rules being used. Lax rules lead detect few defects while editing and leave a large volume of possible defects in the repository. Strict rules lead to a large amount of defects detected while editing, and

leave a smaller amount of defects in the repository. However, verification rules at the editor level are often a nuisance. Furthermore, they cannot be as strict as desired, since rules may require context attributes that not necessarily are available when their evaluation is triggered. The third level examines the contents of the repository. Part of the repository will be explored in accordance to the verification rules.

In addition to verification performed by tools, representations may undergo inspections performed by humans. The recording of the defects and suspicions found during inspections follows exactly the same rules as the recording of defects found by means of tools. Inspections may be aided by tools, as for example check lists. However, the findings are reported by human action. To simplify reporting, Talisman permits the linking of the report generators with inspection support tools.

The rigor of assured quality may evolve in time as the development progresses from a very abstract description of the target system to a more concrete description. For example, when designing a class, the methods which compose this class are specified. At the beginning only the input and output data must be specified for each of them. Later on, decisions are made determining which of these data items are passed by parameters, which ones are object attributes, and what their code names are. Finally, code bodies are built for these methods. Obviously, each of these three states corresponds to a different level of completion and should satisfy different quality control rules.

At given times, pending problem or enhancement reports will be analyzed. This *evolution control* will select some PER to be implemented. This corresponds to moving messages contained in the PER lists to the corresponding findings reports. Since PERs may have been originated outside of the development group, they may have been attached to the wrong hyper-object. Evolution control should move these improperly filed requests to the proper hyper-objects.

Once a modified representation is accepted, all of its consequent representations must have their quality checked again. This quality control may generate new findings reports for several of the objects contained in this representation. Thus, the modification of some representation is propagated to all of its consequent representations, from these to their consequent representations, and so on. While propagating changes to other representations, already accepted representations may become inconsistent with other representations. These defects will lead to new change requests in order to recover consistency. Thus a change may trigger a *change ripple* that traverses back and forth the set of representations, possibly repeatedly returning to a same representation. Eventually this ripple must ebb out, that is, all representations must achieve a mutually consistent state, otherwise a stable set of representations cannot be established, which in turn means that no stable system can be developed. That does not mean that all of the causes leading to change requests and to findings reports have always been completely eliminated. It may be acceptable for a given target system that pending requests are kept for several of its components, but all these requests must be known.

3.10 Version control

Several different versions of a same attribute may exist. Each versioned attribute has its own access key, which takes the version in account. However, the version derivation history is global for each individual repository. Some relations define both the target object and the version of that object. These relations can be used to define

configurations. For example, in a versioned context, a given version of a module is composed of specific versions of its component functions. Since in Talisman a relation is an attribute, the configuration of the module is itself versioned. Thus, the composition of the module in one version may be quite different from that in another version. Each individual repository has its own version history. This version history is not related to the target system version history, that is, a target system version is built from a configuration of attributes possibly contained in several different individual repositories.

Talisman's development paradigm assumes that representations cannot be correctly written from the onset. Rather they must *converge to correctness*, where correctness is defined by means of quality requirements and verification rules applied to the representation. Defects uncovered by these activities stimulate feedback leading to changes in the representation set. Obviously, if representations are always written in order to conform to the defined quality control rules, little effort will be spent processing change ripples, thus, less rework will have to be performed.

It is known that quality control operations usually uncover only part of the defects contained in a representation will be uncovered by the quality control mechanisms. Thus, considering two representations, one containing many defects and the other few, after correcting these representations, the former will probably still contain more defects than the latter [Fenton, 1994].

Once an artifact, i.e. a well defined collection of representations, has been found to be of an acceptable quality, a new *version* of it may be established. In Talisman, a new non modifiable version is created only after the artifact has been accepted. While the change is being developed, a temporary version will be under development. All attributes changed or added while this modification is in progress will receive the same version id. After accepting the temporary version, the version id is set to permanent. When accessing a given version of an attribute, the derivation path of this specific version is used to retrieve the most recent permanent version of this attribute and that is compatible with the version of the artifact being rendered.

When changing a representation, other representations may become inconsistent with this new version, although they are consistent with some older version of it. In other words, for each representation A related to a representation B, *version correctness* must be stated. Hence, version and configuration control cannot be performed adequately by foreign tools; they must be part of Talisman's repository management system. In Talisman the version information is an inextricable part of any given representation.

We may conclude that a representation is composed of three parts:

1. the *representation proper*, containing the versioned attributes of the target system. These attributes may be shown to the environment user by means of representations written in some representation language.
2. the *findings report*, stating all defects found among the attributes of this representation. The findings are stored as attributes of objects which are accessed by the representation.
3. the *change requests*, stating all defects found among frozen attributes used by this representation, or identified by external observations.

3.11 Application generators and transformers

A transformer propagates or reflects information contained in some representation to some other representation. Transformers are defined in terms of the representation languages used. For a given pair of representation languages there may be several transformers, of which one should be selected when performing a transformation.

Application generators are a special class of transformers. Application generators have been proposed as specification driven prototype generators [Balzer, 1981]. Once a prototype has been accepted, the target system is developed from the corresponding accepted specification. Typically, an application generator produces applications directly from specifications (models) containing little, if at all, implementation information. For example, application generators could be used to generate code directly from state transition diagrams. Associating a few source code fragments to states, transitions and labels (processes), it is possible to generate full operational graphical user interfaces [Hübscher, 1995]. Changing the diagram may entail a significant change to the user observed interface, in general requiring almost no changes to the added code fragments. Hence, models can be used not only to generate the first version, but may also be used to maintain the code [Franca, 2000].

Application composers are special kinds of application generators. Typically, they organize the contents of the repository producing adequate input to some foreign tool. For example, consider an editor capable of editing a structure diagram and linking code fragments to this diagram. An application composer would traverse the structure and output the code fragments in the appropriate order required by a compiler. While composing, application composers may also insert code fragments that are not in the structure, or skip parts of the structure (e.g. instrumentation that is compiled only for testing).

Transformers should be capable operating with selected versions of a same artifact. For example, when changing the schema of a database, database files may have to be regenerated. In order to perform the regeneration, some database management systems write the database contents out to a sequential file extracting data in accordance to the new model. The generated file is then read in, rebuilding the data base in accordance to the new structure. The definition of what has to be written and how to read in the temporary file depends on the old and the new version. Often the rules can be derived from the two models, but only if the transformer has access to these versions.

3.12 Reverse engineering and re-engineering

Most organizations already own and use a large amount of software. Independently of how this software has been built and of user satisfaction, this software is usually a valuable asset of the organization and cannot simply be removed or substituted by another without causing major disruption. It follows immediately that, from a pragmatic point of view, any computer aided environment will be successful only if it is capable of absorbing legacy code, designs and specifications, independently of how they have been built. Thus pragmatic environments must be capable of supporting *reverse engineering*.

In its most simple form reverse engineering attempts to recreate engineering documentation from already existing code. For example, a code composer could generate code from a repository using structure diagrams decorated with code

fragments. These structure diagrams correspond essentially to parse trees, where parts of the tree have been coalesced to a single node. What the composer does is to flatten this parse tree, generating sequential source code. Obviously, given existing code this code could be analyzed building the parse tree and store it in the repository. Later on this very detailed tree could be rearranged coalescing some of its parts.

Once an artifact is contained in the repository, the environment user may change it assuring that it corresponds to an acceptable design. This is a form of *re-engineering*. As a result the repository will contain a well organized and documented program structure, from which an enhanced target system code can be generated [Guedes and Staa, 1993].

4 Architectural aspects

In this section we will describe several architectural aspects of the Talisman meta-environment. The architecture of a meta-environment must be process, target representation language and tool independent. Data contained in a definition base establish the specific behavior required by a specific instance of the meta-environment. This organization allows the construction of a large set of environments, without needing to reprogram any of Talisman's components.

Software engineering environments are systems in their own right. They must be populated with several interdependent tools and representation languages editors. They must be specified, designed, developed, tested and maintained, i.e. evolved, adapted and corrected. The environment's characteristics and performance must be continually evaluated and upgraded. They must be reliable and continually available, since they are the basis for the development and maintenance of target systems used by some enterprise. Some of these systems may be long lived and mission critical, hence, their maintenance cannot wait long periods for the tools to be corrected or upgraded. Finally, environments must be long lived, since they should also be used to maintain target systems throughout their life-cycle. A consequence of this longevity requirement is the need for a powerful maintenance infrastructure built in to the meta-environment.

Environment users must be trained in the proper use of tools and languages. Independently of the adequacy and effectiveness of the tools and languages users with insufficient proficiency (knowledge, training and experience) will be unable to produce satisfactory target systems [Parker, 2001]. Thus a fair amount of effort must be spent providing training support as well as on-the-go help for users. Furthermore, tools and languages must be adjustable to the needs and capabilities of the environment users instead of the other way round. This entails a fair amount of experimentation until a proper environment configuration is achieved.

Software engineering meta-environments are expected to be an adequate solution to many of the problems described in the previous sections. Meta-environments are composed of a collection of meta-tools for creating, maintaining, assembling and fine-tuning specialized environments that are adequate considering the five domains identified in the *Introduction*. However, such meta-environments must support target system evolution; otherwise they will not solve long lived software's essential problem. In addition, since much has still to be learned, such a meta-environment should also support experimentation with new tools and representation languages. Ideally the learning curve needed to learn and effectively use the meta-environment should be short and shallow in order to encourage its use while developing or maintaining software, or while experimenting with new software tools or processes.

4.1 System architecture

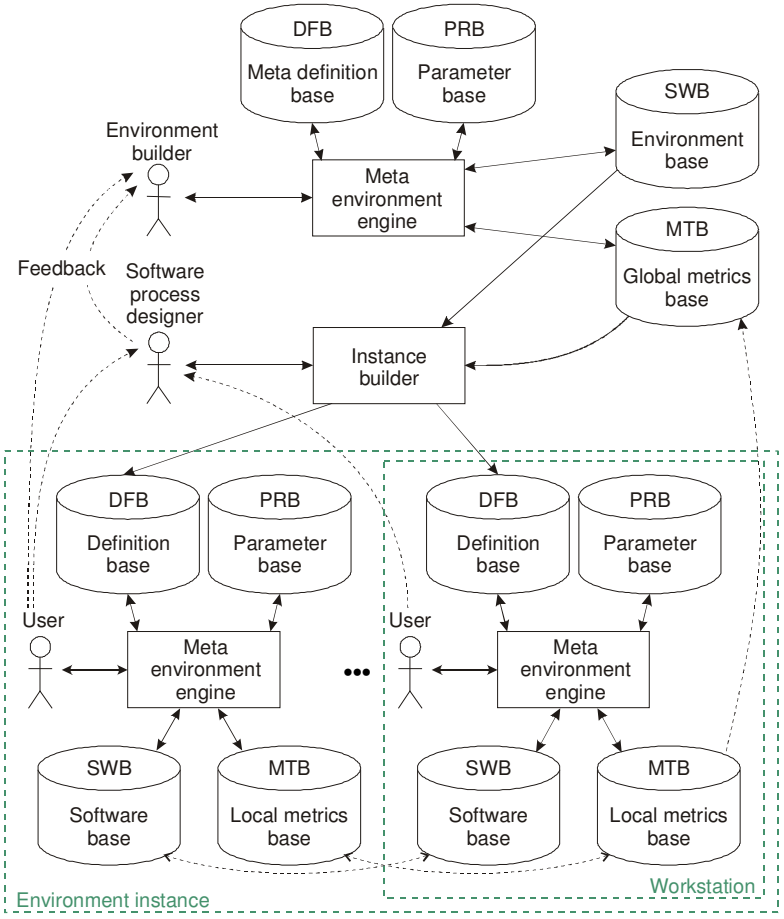

Figure 10. Talisman version 5 overall system architecture.

Figure 10 depicts Talisman's system architecture. The *meta-environment engine* is controlled by two special purpose data bases: the *Parameter base* and the *Definition base*. Essentially the meta-environment engine is capable of interpreting a plethora of specifications that instantiate the available meta-tools, where these specifications are contained in the parameter and definition bases. To assure better response times for the user, these two bases must be in an interpretable format.

The parameter base contains engine specific descriptions, which are invariant for all instances of a same version of the meta-environment engine. It defines all frozen elements that govern the functionality of the Talisman system. Examples of these elements are: the meta-schemata of the definition base and of the software base. Other examples are static symbol tables that define symbols such as: user interfaces, messages and representation language symbols. The parameter base may have to be redefined whenever the engine evolves.

Every *Environment instance* is defined by a set of *Definition bases*. Typical data contained in the definition bases are interpretable (binary) definitions of representation language, editors, verifiers, transformers, generators and other development tools. An environment instance contains one or more *Environment workstations*. Each of these workstations is driven by its own definition base. Furthermore, it is used by an *Environment user* and should be configured for the needs of the role this user plays while developing or maintaining artifacts.

A special kind of definition base is the *Meta-definition base*. This definition base enables the Talisman engine to create and maintain the *Environment base*, which contains all tool and representation language descriptions in an editable format. In fact the meta-definition base is just a normal definition base that instantiates the meta-environment engine to edit the environment base.

The *Environment builder* uses the meta-environment instance containing the interpretable definitions of *meta-environment representation languages* and *meta-environment tools* that are needed to create, edit and verify the contents of the environment base. Notice that this architecture allows creating and maintaining the meta-definition base using the same meta-environment engine as the one used to develop target systems. After an initial bootstrap development, this architecture allows maintaining the meta-definition base using the Talisman system itself.

Using the *Instance builder* tool, the *Software process designer* selects the representation languages, tools and possibly workflows that should be used while developing or maintaining specific software. This selection is necessary since the environment base contains all available language and tool descriptions, where some of them may be mutually exclusive when considering a specific environment. While performing the selection, the coherence of this selection must be verified. The instance builder typically transforms the symbolic descriptions contained in the environment base to engine interpretable descriptions contained in the definition base. The result of the instance builder is a set of one or more definition bases. Several definition bases are generated if the software process designer wishes to create different environment workstations, one for each environment user. The instance builder is in fact part of the bootstrap support enhanced with some meta-environment tools and representation languages.

The collection of definition bases establish a specific *Environment instance* that is geared towards the actual environment users and supports the development of target systems in some specific application domain, using specific technology and environment domains for its development and maintenance. The collection of definition bases describe all the representation languages specifications (rendering, user interfaces, syntax and semantics) and tools (verifiers, transformers, code composers, measurement tools among others) to be used while developing or maintaining a target system.

Several different environments instances may be assembled. For example, one might want to create a specific environment for developing information systems, and another one for developing embedded control systems. The criteria used to select the components of a specific environment instance from the environment base are typically: quality requirements of the target systems to be developed, software development standards to be obeyed, technology and application domains of the software to be developed, and environment user proficiency. While composing a definition base possibly new development knowledge is acquired. Similarly, while adapting the environment to its users knowledge may be acquired too. Both cases lead to the need of updating the environment base.

The *Global metrics base* contains all measurement data about all projects developed. The measurements are collected by each of the environment stations and then integrated into the global metrics base. This base gathers information that can be used to enhance the environment tools, languages and workflows.

4.2 Environment instance

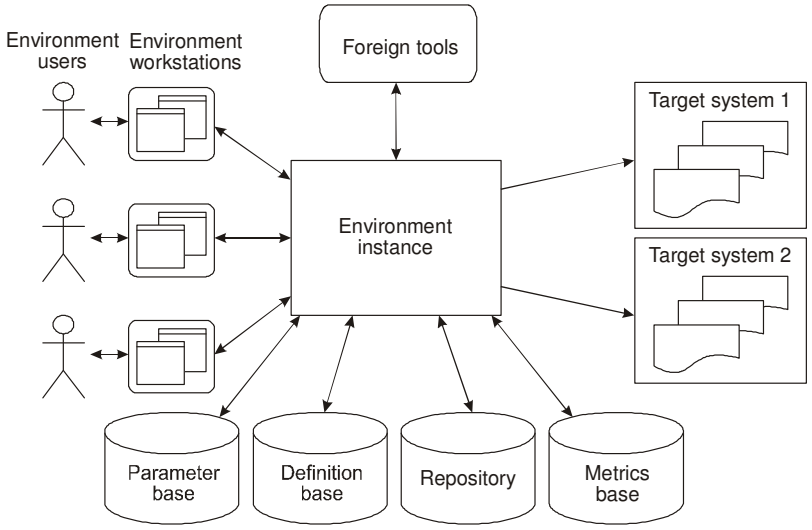

Figure 11. Interfaces of an environment instance

Using an environment instance one or more *Environment users* cooperatively create and maintain *Target systems*, see figure 11. The facts of these systems are kept in a *Repository*, which consists of a collection of interconnected *software bases*. Whenever desired environment users may export files to be used by other tools (e.g. code to be compiled) or import files generated by other tools (e.g. XML or XMI files). Imported files are decomposed and saved in a software base.

The development of target systems is performed by populating and maintaining the repository. The structure of the repository is controlled by the definition and the parameter bases. On the environment administrator's discretion, software bases may contain all facts about all target systems of the enterprise, or may contain facts about one or few of these target systems, or even of a part thereof.

One of Talisman's aims is to provide interoperating interfaces with existing tools. In particular, Talisman should support development, maintenance and quality control right down to code and documentation. Thus, environment instances must be capable of exporting source code to foreign tools. Examples of exported files are source code files directed towards some language processor, model or code verifier, and target system source documentation files directed towards some text formatter.

Conversely, other tools might be used to create and maintain facts about target artifacts and, hence, their output must be imported by the environment instance. For example it may be interesting to import the list of compile errors and bind them to facts contained in the software base. Thus, Talisman could act as a front-end of several other tools. Foreign tools may interface with the environment by means of sequential files, e.g. some XML file. These files typically adhere to a syntax dictated by the foreign tool. Talisman provides pattern matching tools that help building interactive interfaces. These interfaces are implemented by means of form programs. As an example consider the case of cross code generation and compiling. Here the environment is used on a given development platform to produce code which will run on some different target platform. The target system could run in a virtual machine sharing files with the base machine. Typically the language processors and test support tools will reside on the target platform. In many cases, especially when not using virtual machines, it is cumbersome to continually transfer files between different

platforms. Due to this, programmers will tend to compile and test programs and, possibly change the program, in the target platform. However, any changes to the generated programs made at the target platform will make the program inconsistent with the repository. Thus, unless there is some mechanism to recover changes made outside of the environment and insert them back into the repository, the user may complain that the environment hampers effective development.

4.3 Environment workstations

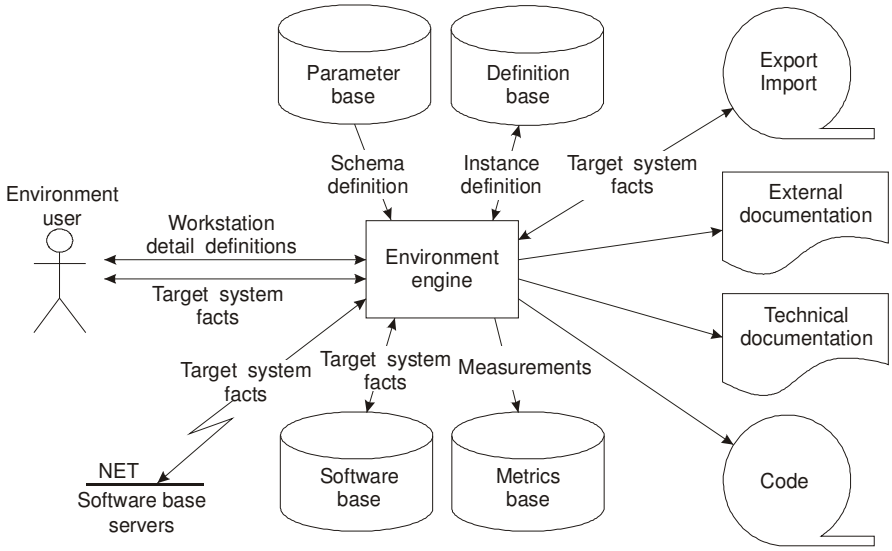

Figure 12. Interfaces of an environment workstation

An environment workstation is a partition of the environment instance used to develop and maintain one or more components of a target system. Each environment workstation is tailored towards a specific environment user (user role) supporting the *development activities* that this environment user is allowed to perform. Figure 12 shows the typical interfaces of an environment workstation.

Every environment workstation is used by an environment user. This person explores, adds, changes and deletes *target system facts* about the target system component being developed. These facts are kept in the software bases. Each instance allows several tools to be used. However, tools may restrict which facts may be changed. The access permissions are contained in the *definition base*.

Talisman provides two forms of maintaining definition bases. The first form, as described in section *4.1 System architecture*, requires updating the environment base and then deriving the specific definition base using the instance builder tool. The evolution of the environment instance should have a minimal effect on the ongoing work. To reduce the risk of loss of work this tool provides version and configuration control of the environment itself. This form is needed for creating a new representation language, or when architectural changes are required in some existing language.

The second form of changing the definition base is to allow the environment user to fine-tune the environment instance to her/his particular needs. These changes must not modify representation language schemas, or even software base schemas. They may modify the way artifacts and representations are composed and rendered.

While the environment workstation is being used, several *process metrics* and *artifact metrics* may be gathered, such as number of objects created, changed and deleted, usage of tools and error counts while using tools. Other metrics may be extracted from the repository, such as number of hyper-objects per hyper-class, existence and size of selected attributes. All these metrics can be used to improve the processes and tools used by the environment workstation. They can also be used to point out possible design or implementation anomalies (*bad smells*) [Macía, 2009].

4.4 Language category meta-editor interaction

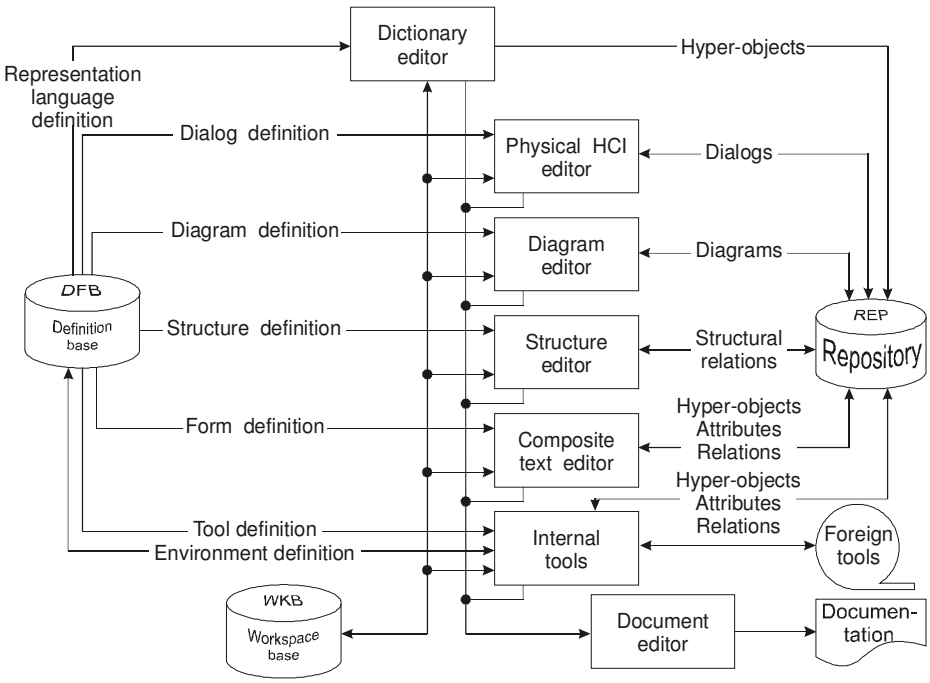

Figure 13. The structure and components of Talisman

Figure 13 shows some of the language categories supported and how the corresponding meta-editors interact. As already mentioned, the set of meta-editors is open-ended. However, each time a new meta-editor is added, the programs which compose the Talisman engine must be changed and possibly the storage schemas of the repository and of the definition base will have to be changed too.

In the sequel we will briefly describe these meta-editors. The meta-editors shown are: a dialog meta-editor for designing human interfaces; a diagram meta-editor for graphical representations; structure meta-editor for hierarchical representations such as structure charts; and a composite text meta-editor for textual representations. The parameter and the definition bases specialize these meta-editors for specific representation languages.

The dictionary editor is a specialized processor capable of browsing all hyper-class directories. It triggers actions on a selection of hyper-objects. As discussed before, representations are always reconstructed from a focal hyper-object, a language category identification, a language family identification, a form program and an action. For example, the dictionary editor may be used to generate code using the structured programming language family. In this case, the composite text meta-editor is selected, the code generating rule set corresponding to the programming language

is chosen, and the code is generated for each of the selected objects. If the target of the editor is a viewport, the user may edit all text fragments of the generated code which have been fetched from the repository. Otherwise, if the target of the editor is a file, the code will be written out onto this file and may then be submitted to a compiler.

Each representation language operates on some level of abstraction such as the system, module or code level and establishes an interpretation for its assembled collection of repository facts. For example, the interpretation could encompass activities such as functional modeling, user interface modeling, data modeling, and test data definition.

The storage schema of the repository depends on the representation language categories which the meta-environment is capable of supporting. Thus, if a new category is implemented, possibly the storage schema must be adapted in order to facilitate composition and decomposition of representations written in languages of this new class. It is expected that this kind of evolution is not frequent.

4.5 Meta-editors

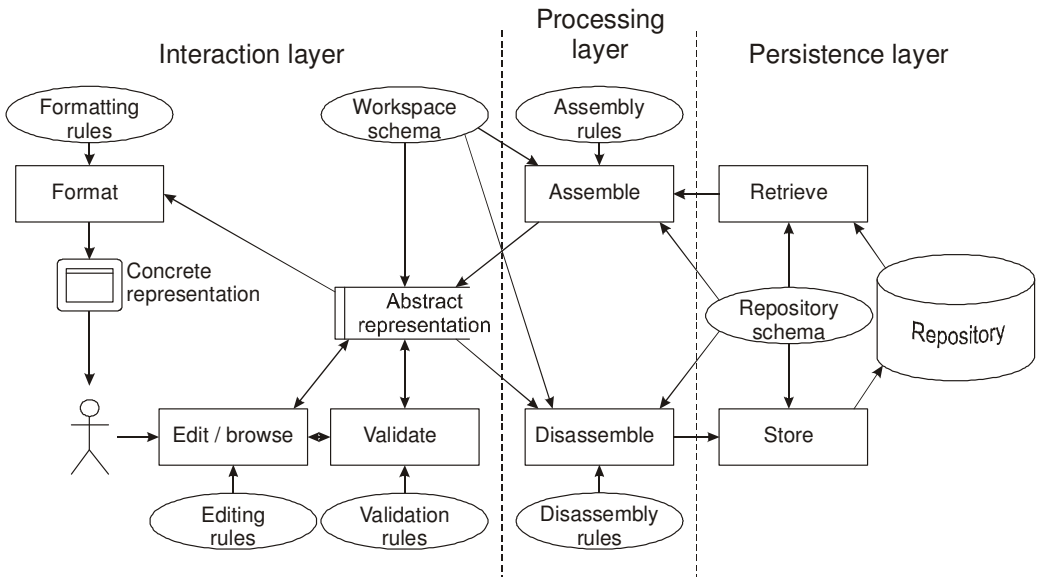

Figure 14. Meta-editor architecture

Each concrete representation is edited by some meta-editor instantiated for this representation. All these meta-editors satisfy the generic architecture shown in figure 14. The behavioral model contains both concrete and abstract representations. The user operates on the concrete representation, which is rendered from the corresponding abstract representation through a set of formatting rules. While editing the user interacts with the concrete representation changing the underlying abstract representation.

The abstract representation is similar to a syntax graph of the representation. This syntax graph contains tags identifying all attributes brought in from the repository, tags defining access rights, structure tags and formatting tags. The first two groups of tags define the structure of the representation at the repository attribute level, whereas the third and fourth groups of tags contain data for the rendering rules. The representation language is used to define how to format and display the abstract

representation. This may be achieved by means of a style sheet, where this style sheet is part of the representation language definition.

The repository schema provides a definition of the organization of the data elements in the repository. The data elements are stored in or retrieved from the repository using this schema. When a representation is to be rendered the processing layer fetches chunks of data from the repository and assembles them forming the abstract representation, which is stored in the workspace base. When the abstract representation should be saved the processing layer disassembles the abstract representation and saves its elements in the repository. While editing context independent verification rules are applied as described in section *3.9 Quality assurance*.

For each language category a specific set of program components has to be developed. The *formatter* renders and displays the concrete representation corresponding to the contents of the abstract representation. The *editor* receives user commands and modifies accordingly the contents of the abstract representation. The *verifier* verifies the correctness of the changes made on the abstract representation. As already seen, context insensitive actions may be validated before changing the contents of the abstract representation or before committing the abstract representation to the repository. In this way the abstract representation acts as a data gathering device, where these data will be used by a repository updating transaction whenever the abstract representation is saved.

There is a need to translate between the organization of the repository and the abstract representations with which the editor interacts. The *assemble* and *disassemble* components translate the organization of the data contained in the repository to the organization required by the abstract representation and back.

These five components are meta-components and provide means to edit and browse all representation languages of a given representation language category. While generating the specific environment, the representation language definitions are converted into a set of rules. These rules are represented as ellipses in figure 14. The formatting, editing and validation rules connect the abstract representation to the concrete representation manipulated by the user, whereas the assembly and disassembly rules define how to convert the contents of the repository to the abstract representation and back. There is a set of rules for every representation language. Experience with Talisman 4.4 has shown that writing a wholly new representation language can be accomplished in a couple of days or less. It has also been shown that it is worthwhile to invest few hours specializing tools for a given project.

4.6 Definition base and software base interaction

Figure 15 shows a fragment of the definition base and software base schemata and their interaction while rendering a fragment of a diagram. The parameter base stores both definition and software base schemata. The environment engine is tightly bound to these definitions; hence any change in the diagram handling component may entail a change in these schemata and vice-versa.

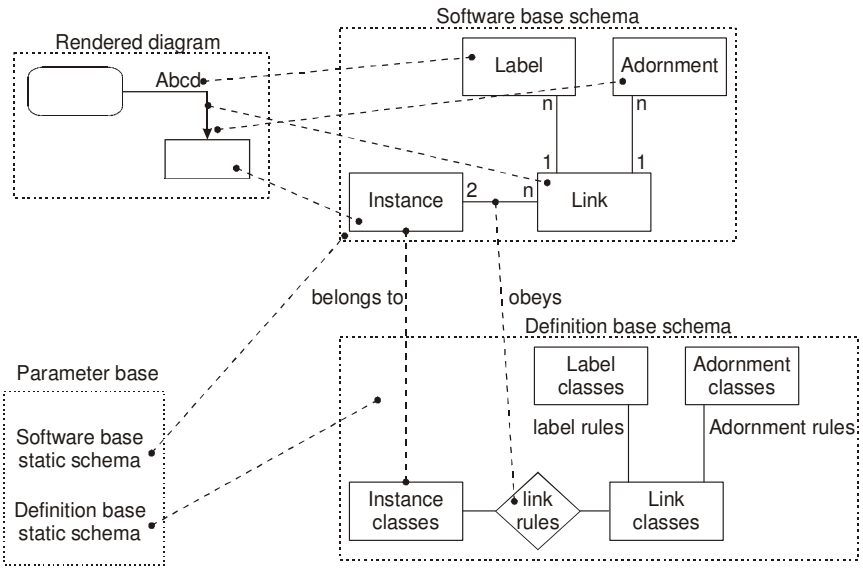

Figure 15. Definition base and software base schemata interaction

The definition base defines the hyper-classes and the relations between them. Instance classes correspond to instance hyper-objects (sub-objects) which are rendered as some box. The rendering rules are kept in the definition base whereas the data is kept in the software base. For example, the instance classes (rounded box, rectangle, relations) define how the hyper-object instances are to be created (Box 1, Box 2, coordinates in the diagram, sizing of the boxes, relationships). Since hyper-object instances may correspond to some hyper-object, the instance may also contain a relationship to this object (not shown). In this case the name is usually extracted form the hyper-object instead of from the instance. Instances may be linked by an edge. The link hyper-class defines what hyper-classes may be linked, the aspect of the linking line, the set of labels and adornments that may be associated with the link.

A diagram is stored as a list of references to hyper-object instances, links, labels and adornments. When a diagram is to be rendered the view-port where it will be rendered and the focal instance must be given. The diagram meta-editor selects the instances and links that can be displayed. While rendering it also renders the labels and adornments if they can be shown in the viewport.

4.7 Repository properties

All Talisman development activities interact with the repository. Developing a system corresponds to populating and/or updating this repository. The repository contains all *facts* (hyper-object attributes) about the target systems being developed. It may also contain standard facts which have a wider scope than a particular system. For example, corporate data models and dictionaries are often defined in order to assure integration between different target systems, possibly being developed in different environments by different teams.

Repositories may be acquired from third parties. For example, a corporation providing some interactive service for other corporations could distribute repositories containing components of this service. This would allow distributing not only the source code but also all maintenance support documents and tools. Finally, the

repository and tools should encourage generalized reuse. In other words, it is desired that the different target systems share large quantities of facts.

Since Talisman is a meta-environment, the hyper-classes and hyper-object attributes may change at any time, as long as the underlying schema is not changed. For example, at any time new hyper-object attributes may be established. One of the basic characteristics of the data contained in the repository is the large quantity of relations between these data. Typical development using several levels of abstraction and view-points induces several relations that establish how a high abstraction level fact evolves until it reaches the corresponding lowest level facts. Obviously it should be possible to show this evolution in some specialized representation.

When developing a new work-product, the local software base starts empty. When a legacy system is to be recovered, the local software base may be populated by means of a reverse engineering operation, or by accessing data extracted from a foreign software base that contains its design and implementation. Another way of integrating the contents of software bases is achieved by means of links between objects contained in different software bases. In this case no copy operation will be needed. These mechanisms allow the integration of all facts about target systems developed or re-engineered with the support of the environment.

Talisman uses a proprietary object oriented data base. Such databases seem to be more appropriate to CAD/CAE and by extension CASE applications [Nyström, 2004]. We could have followed the Object Definition Standard (ODL) [Cattell et all, 2000], however departed from it for several reasons, the main one being the existence of a large portion of code that would have to be rewritten.

4.8 Attributes

The repository contains *hyper-objects*. Hyper-objects may contain zero or more *sub-objects*. Hyper-objects and sub-objects are instances of *hyper-classes*. Only one sub-object level is supported. Deeper structures can be achieved using recursive relations, such as *composition* and *decomposition*. In fact, a sub-object could be viewed as a named sub-domain of a given hyper-object's attributes. For example, a diagram (a hyper-object) contains several attributes for each of the items (boxes, links, labels, adornments) that compose this diagram. Each of these items corresponds to a sub-object. However, sub-objects may refer to the hyper-objects that are instantiated in the diagram. For example, in an UML **`class diagram`** boxes (sub-objects) refer to the **`classes`** (hyper-objects) that contain the facts (e.g. specifications, interfaces, attributes, methods) of these **`classes`**. Conversely, each **`class`** type hyper-object should refer to all places where it appears as a sub-object in some diagram. This latter relation allows navigating from any occurrence of a class to any other occurrence.

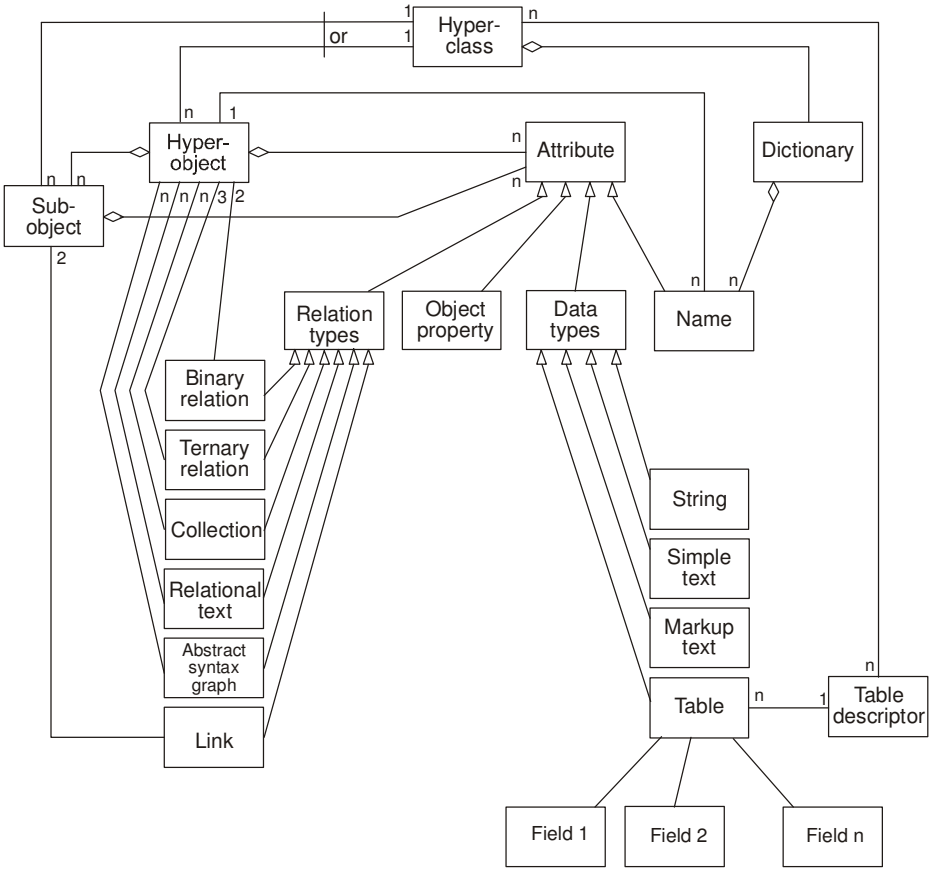

Figure 16. Attribute categories

Each hyper-object and sub-object contains several attributes. Each attribute belongs to an *attribute category*, see figure 16. Examples of attribute types are: "name", "simple text", "string", "binary relation", and "link". Attributes can be unlimited large, e.g. a text attribute. Within each category, a virtually unrestricted set of attributes can be defined for each hyper-object. Each attribute has its own access key. From a strict object oriented language view, attributes correspond to objects and attribute categories correspond to classes. Hyper-class descriptors are hyper-objects too, but are kept in the definition base instead of in the software base.

Hyper-object attributes are *fine grained*. *Coarse grained objects* such as diagrams, program code, or other representations are usually composed from a large quantity of fine grained attributes and are recomposed every time they are required by extracting attributes contained in the repository as discussed in section *3.6 Navigating over representations*. In fact a diagram is a hyper-object that contains several sub-objects, each of which referring to another hyper-object.

A *dictionary-class* is a special type of hyper-class. It contains the set of all dictionaries. A *dictionary* is an attribute of this class. Dictionaries contain an alphabetized list of hyper-object names, where these hyper-objects are instances of the hyper-class corresponding to the dictionary. Each hyper-class may refer to several dictionaries. This allows a hyper-object to own several names (aliases), each of which aiming at a specific use. For example a method name may be a string containing several words in some natural language, as well as code names used in a given programming language. The latter can be used by editors to mark up code text

allowing navigation from the text fragment to the hyper-object corresponding to the method name found in the code.

Another special attribute is a *table descriptor*. A table is similar to a Pascal record or a SQL table definition. A table contains one or more fields. Field descriptors have a name (e.g. column name in SQL) and a type identifier. Examples of field types are integers, floating point numbers, date, time and fixed size strings.

Hyper-objects contain also attributes that are system defined and are kept in *object descriptors*. Among them are data that define properties of the hyper-object. Examples of such properties are the type identifier of the hyper-object, and coordinates and size parameters of a graphical sub-object of a diagram. Object descriptors may also contain system defined relations, as for example the list of sub-objects that refer to a given hyper-object.

4.9 Relations

Each hyper-object or sub-object[6] may contain several relations to other hyper-objects or to itself. Relations are attributes that link the owning hyper-object to zero or more other hyper-objects. They may link hyper-objects that are part of different representation languages. There may be several relations involving the same two hyper-objects. For example, consider the **`work break down structure`** representation. Here an **`activity`** ***`decomposes`*** into several other **`activities`**, but it also ***`depends on the completion`*** of other **`activities`**.

Relations may require specific information for each of its relationships. For example, consider the ***`method of a class`*** relation. Here it may be necessary to discriminate whether this method is ***`public`*** or ***`private`*** with regard to the corresponding class. However, the same method may be related to more than one **`class`** each defining *visibility* in a different way. Thus *visibility* is not a property of the method itself, but is a property bound to the ***`method of a class`*** relationship.

Talisman implements inheritance using relations (*inherits_from* and *is_inherited_by*). Similarly *composition* and *decomposition* are relations as are *contains* and *is_contained*.

Mark-up texts are text containing formatting information (small subset of HTML). A *Relational text* is a mark-up text that contains references to hyper-objects. These references refer to attributes of other hyper-objects that should be gathered when rendering a representation containing the relational text. For example in a program code instead of inserting literal method names one could insert a reference to the hyper-object describing the method. When rendering the program code the reference can be substituted by the appropriate code name of the method, and which can be chosen depending on the programming language used. Furthermore, since the text contains a link to a hyper-object, this link may be used to navigate to other representations using the referred hyper-object as focus.

A *collection* refers to the hyper-object attributes refer to several hyper-objects of a variety of hyper-classes. Collections may be specialized to *stacks*, *queues*, *lists*, *sets* and *bags*. A *link* is a graphical relation between graphical elements of a diagram. Links are always binary. In diagrams where links may interconnect more than two elements, a special element *connector_node* must be defined.

---

[6] To reduce text repetition we will use the term hyper-object to denote both hyper-objects and sub-objects.

An abstract syntax graph is a directed graph that describes the syntax structure of some text (usually a code fragment). Nodes of an *abstract syntax graph* (ASG) may refer to hyper-objects or to strings in a symbol table associated with the ASG. The procedure used to compute the ASG is contained in the definition base.

4.10 Multi-base

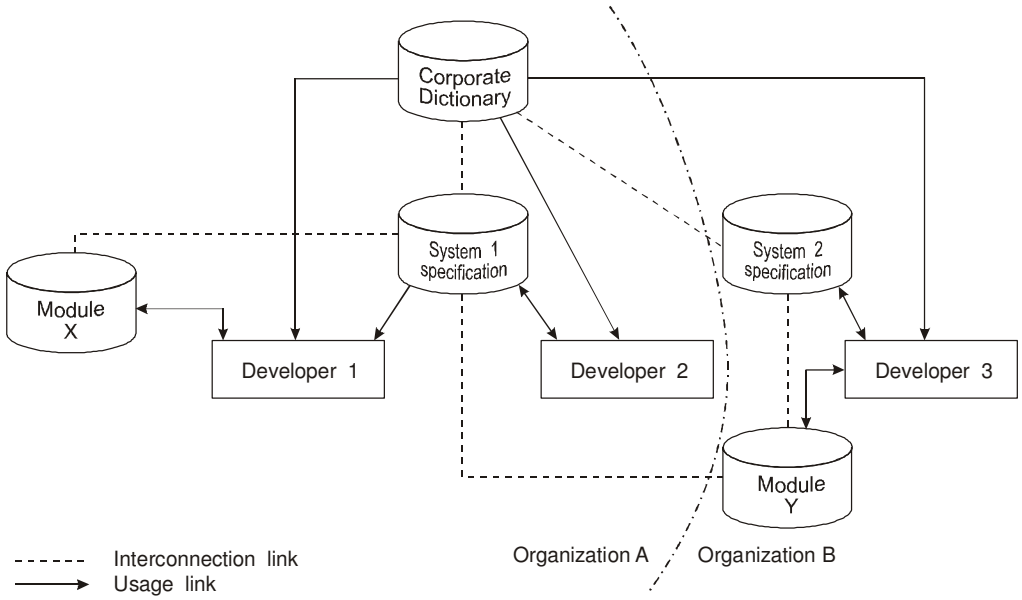

Figure 17. Logical partitioning of repositories

Talisman's repository is a collection of interdependent software bases, where these software bases may be distributed over a network, see figure 17. Hyper-objects in one individual software base may relate to hyper-objects contained in another software base. For every remote hyper-object a local indirect object must exist. Such objects establish the link with the remote object. Since all relations contain an inverse, also indirect objects will be referred to by the remote object.

The partitioning of the overall repository into the set of software bases is arbitrary. This multi-repository structure allows using semantical (e.g. financial aspects, personnel aspects), or abstraction-level (e.g. requirements specification, architecture, code, test), or other criteria to define the partitioning. It allows also partitioning the repository with respect to the organization that maintains it, as shown in figure 17. This allows sharing of software bases across organization boundaries and, hence, provides means to establish verbatim reuse even when several organizations collaborate for the development or maintenance of a target system.

4.11 Shared software bases

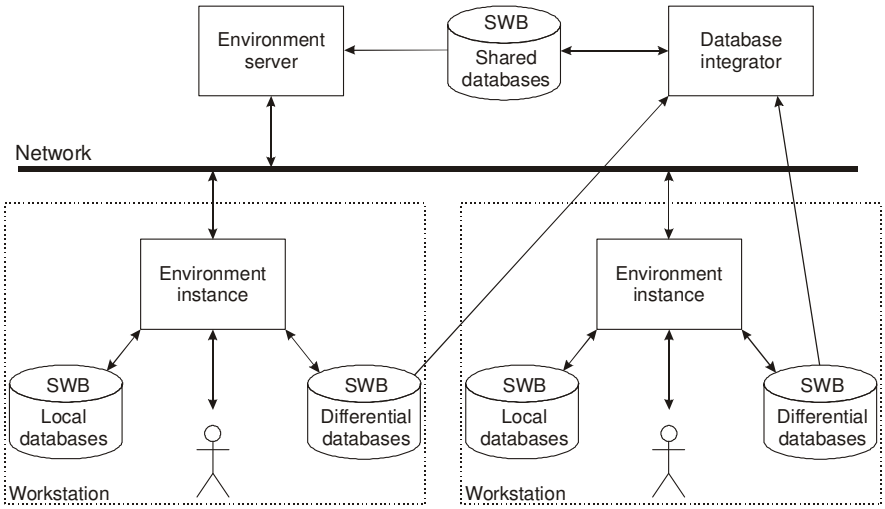

Figure 18. Composing an environment from several instances

As illustrated in Figure 18, every environment instance may interact with shared *public software bases* as well as with several local (not shared) software bases. Public software bases are always read only.

Private software bases allow locking of hyper-objects or attributes. For example, when developing a target system that uses a specific library, the symbolic interface definition of this library could be included in the software base. Obviously these definitions may not be changed; otherwise it would be impossible to assure correct interfacing between target systems code and the pre-compiled library.

When editing a representation, public objects may have to be updated. For example, when establishing a reuse link, the reused object must be updated in order to identify all places where it is used. When a public repository needs to be updated, it may either be checked out or a *differential software* base may be created. Checking out the software base entails assigning it to the user who will update it, while this updating is in progress no other user may access the software base. Since development transactions might turn out to be quite long (days), checking out is often felt as a hindrance to work progress.

The differential software base contains all changes to be made to a public software base. When the artifact being developed has been accepted, the differential software base can be integrated with the public repository. This integration creates a new version for each of the modified attributes and also adjusts the version references contained in the private software base. If several users update simultaneously the same public software base, conflicts may arise. Talisman's integration tool will provide mechanisms to help users to resolve these conflicts by negotiation.

4.12 Repository integrity

The repository is a very critical component of a computer aided software development environment. If it gets damaged, months of work may be lost, even if a backup procedure is in place. Hence, tools must be provided that are capable of verifying and reconstructing a valid repository if its structure is corrupted. These tools must be

capable of reconstructing the repository using available meta-data in the definition base, as well as static meta-data available in the parameter base.

All data structures, both memory resident and persistent, should satisfy robustness criteria [Taylor et al, 1980; Taylor and Seger, 1986; Staa, 2000; Demsky and Rinard, 2003]. To allow verification and recovery with little loss, selective redundancy will be included in the schemata and models, as well as in the code. Furthermore, multi-threaded structure verifiers should be available. These verifiers should be capable of operating while the work station is being used. Whenever they detect a failure, execution should be interrupted in such a way as to prevent persistent data corruption.

5 Concluding remarks

In this report, we presented an overview of the functionality of the Talisman software engineering meta-environment. A prototype, Talisman version 4.4, already exists and has been in use for more than 15 years. This prototype and a plethora of available papers have allowed us to identify new requirements as well as establish several design issues.

Talisman version 5 is being designed and implemented using Talisman version 4.4. Once a bootstrap version of the new system is available, the development will continue using Talisman version 5 itself. We hope that this approach will allow us to iron out several specification and architectural defects, leading to a better initial version.

For the purpose of developing Talisman version 5 several new tools have been added to the Talisman 4.4 environment using its meta-environment capabilities. Furthermore, an automated testing environment has been created. In this way we are not only developing Talisman, but are also using this development effort to assess the stated requirements for tools and representation languages, which are ultimately to be satisfied by Talisman.

Acknowledgements

Special thanks are due to the early reviewer of this report: Thiago Araújo.

6 References

[Antkiewicz, 2006] ANTKIEWICZ, M.; "Round-Trip Engineering of Framework-Based Software using Framework-Specific Modeling Languages"; 21st IEEE International Conference on Automated Software Engineering; Los Alamitos, CA: IEEE Computer Society; 2006; pags 323-326

[Araújo, 2010] ARAÚJO, T.P.; **SDiff: Uma ferramenta para comparação de documentos com base nas suas estruturas sintáticas**; Dissertção de Mestrado; Departamento de Informática, PUC-Rio; Rio de Janeiro; 2010; in Portuguese

[Arisholm et al, 2006] ARISHOLM, E.; BRIAND, L.C.; HOVE, S.E.; LABICHE, Y.; "The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation"; IEEE Transactions on Software Engineering 32(6); Los Alamitos, CA: IEEE Computer Society; 2006; pags 365-381

[Armour, 2003] ARMOUR, P.G.; **The Laws of Software Process: A New Model for the Production and Management of Software**; New York: Taylor & Francis; 2003

[Avizienis et al, 2004] AVIZIENIS, A.; LAPRIE, J-C.; RANDELL, B.; LANDWEHR, C.; "Basic Concepts and Taxonomy of Dependable and Secure Computing"; IEEE Transactions on Dependable and Secure Computing 1(1); Los Alamitos, CA: IEEE Computer Society; 2004; pags 11-33

[Balzer, 1981] BALZER, R.; "Transformational Implementation: An Example"; IEEE Transactions on Software Engineering SE-7(1); Los Alamitos, CA: IEEE Computer Society; 1981; pp 3-14

[Beck, 2010] BECK, K.; "The Inevitability of Evolution"; IEEE Software 27(4); Los Alamitos, CA: IEEE Computer Society; 2010; pags 28-29

[Berry et al, 2010] BERRY, D.M.; CZARNECKI, K.; ANTKIEWICZ, M.; ABDELRAZIK, M.; "Requirements Determination is Unstoppable: An Experience Report"; 18th IEEE International Requirements Engineering Conference; Los Alamitos, CA: IEEE Computer Society; 2010; pags 311-316

[Bigelow, 1988] BIGELOW, J.; "Hypertext and CASE"; IEEE Software; Los Alamitos, CA: IEEE Computer Society; 1988; pp 23-27

[Brooks, 1987] BROOKS, F.P.; "No Silver Bullet - Essence and Accidents of Software Engineering"; IEEE Computer 20(4); Los Alamitos, CA: IEEE Computer Society; 1987; pags 10-19

[Cattell et all, 2000] CATTELL, R.G.G.; BARRY, D.K.; BERLER, M.; EASTMAN, J.; JORDAN, D.; RUSSELL, C.; SCHADOW, O.; STANIENDA, T.; VELEZ, F.; **The Object Data Standard: ODMG 3.0**; Morgan Kaufmann; 2000

[Conklin, 1987] CONKLIN, J.; Hypertext: "An introduction and survey"; IEEE Computer vol 20 no 9; 1987; pp 17-41

[DeMarco, 1979] DEMARCO, T.; **Structured Analysis and System Specification**; Upper Saddle River, NJ: Yourdon Press; 1979

[Demsky and Rinard, 2003] DEMSKY, B.; RINARD, M.; "Automatic Data Structure Repair for Self-Healing Systems"; 2003 ACM SIGPLAN Conference on Object-

Oriented Programming Systems, Languages, and Applications; New York, NY: ACM Association for Computing Machinery; 2003

[Eick et al, 2001] EICK, S.G.; KARR, A.K.; MARRON, J.S.; MOCKUS, A.; GRAVES, T.L.; "Does Code Decay? Assessing the Evidence from Change Management Data"; IEEE Transactions on Software Engineering 27(1); Los Alamitos, CA: IEEE Computer Society; 2001; pags 1-12

[Fenton, 1994] FENTON, N.; "Software Measurement: A Necessary Scientific Basis"; IEEE Transactions on Software Engineering vol. 20 no. 3; Los Alamitos, CA: IEEE Computer Society; 1994; pp 199-206

[Fowler, 2000] FOWLER, M.; **Refactoring: Improving the Design of Existing Code**; Reading, Massachusetts: Addison-Wesley; 2000

[Franca, 2000] FRANCA, L.P.A.; **Um Processo para a Construção de Geradores de Artefatos**; Tese de Doutorado Departamento de Informática, PUC-Rio; Rio de Janeiro; 2000; in Portuguese

[Gane and Sarson, 1978] GANE, C.; SARSON, T.; **Structured Systems Analysis: Tools and Techniques**; Upper Saddle River, NJ: Prentice Hall; 1978

[Glass, 2003] GLASS, R.L.; **Facts and Fallacies of Software Engineering**; Reading, Massachusetts: Addison-Wesley; 2003

[Guedes and Staa, 1993] GUEDES, L.C.; STAA, A.v.; "Um processo de reengenharia econômico e eficaz"; 7o. SBES Simpósio Brasileiro de Engenharia de Software, Rio de Janeiro, 1993; Porto Alegre, RS: Sociedade Brasileira de Computação; 1993; pags 77-91; in Portuguese

[Hochstein and Lindvall, 2004] HOCHSTEIN, L.; LINDVALL, M.; "Diagnosing Architectural Degeneration"; 28th Annual NASA Goddard Software Engineering Workshop; Los Alamitos, CA: IEEE Computer Society; 2004; pags 137-142

[Hübscher, 1995] HÜBSCHER, P.J.E.; **Processo de desenvolvimento de programas C++ utilizando Talisman e MFC**; Dissertação de Mestrado; Departamento de Informática, PUC-Rio; 1995; in Portuguese

[Ierusalimschy, 2004] IERUSALIMSCHY, R.; **Programming in Lua**; Rio de Janeiro: Lua.org; 2004

[Kajko-Mattson, 2000] KAJKO-MATTSON, M.; "Preventive Maintenance! Do we know what it is?"; 16th IEEE International Conference on Software Maintenance; Los Alamitos, CA: IEEE Computer Society; 2000; pags 12-14

[Kemerer and Slaughter, 1999] KEMERER, C.F.; SLAUGHTER, S.A.; "An Empirical Approach to Studying Software Evolution"; IEEE Transactions on Software Engineering 25(4); Los Alamitos, CA: IEEE Computer Society; 1999; pags 493-509

[Lehman and Belady, 1985] LEHMAN, M.M.; BELADY, L.A.; eds.; **Program Evolution: Processes of Software Change**; London: Academic Press; 1985

[Lehman, 1996] LEHMAN, M.M.; "Laws of Software Evolution Revisited"; 5th European Workshop on Software Process Technology; Berlin: Springer, Lecture Notes in Computer Science 1149; 1996; pags 108-124

[Macía, 2009] MACÍA, I.; **Avaliação da Qualidade de Software com Base em Modelos**; Masters Dissertation; Informatics Department; PUC-Rio; 2009; in Portuguese

[Magalhães et al, 2009] MAGALHÃES, J.A.P.; STAA, A.v.; LUCENA, C.J.P.; "Evaluating the Recovery Oriented Approach through the Systematic Development of Real Complex Applications"; Software Practice and Experience 39(3); New York: Wiley Periodicals; 2009; pags 315-330

[Nyström, 2004] NYSTRÖM, M.; **Engineering Information Integration and Application Development using Object-Oriented Mediator Databases**; Doctoral Thesis; Department of Applied Physics and Mechanical Engineering, Lulea University of Technology; 2004

[Parker, 2001] PARKER, L.; **A Fool with a Tool is still a Fool**; HP Open View; 2001; Retrieved: mai/2007; URL: http://www.parallon.com/a_fool_with_a_tool_is_still_a_fool.pdf

[Pietrobon, 1995] PIETROBON, C.A.R.; **Gerência de Configuração em Ambientes de Trabalho Cooperativo**; Tese de Doutorado; Departamento de Informática, PUC-Rio; Rio de Janeiro; 1995; in Portuguese

[Reason , 2003] REASON, J.; **Human error**. Cambridge: Cambridge University Press; 2003

[Staa and Cowan, 1995] STAA, A.v.; COWAN, D.D.; **An Overview of the Totem Software Engineering Meta-Environment**; Monografias em Ciências da Computação, Departamennto de Informática, PUC-Rio, PUC-RioInf.MCC 35/95; 1995

[Staa, 1993] STAA, A.v.; **Talisman: Ambiente de Engenharia de Software Assistido por Computador, Manual de Referência**; version 4.2; Rio de Janeiro: STAA Informática; 1993; in Portuguese

[Staa, 2000] STAA, A.v.; **Programação Modular**; Rio de Janeiro; Campus; 2000; in Portuguese

[Sterling, 2011] STERLING, C.; **Managing Software Debt: Building for Inevitable Change**; Reading, Massachusetts: Addison-Wesley; 2011

[Taylor and Seger, 1986] TAYLOR, D.J.; SEGER, C.J.H.; "Robust storage structures for crash recovery"; IEEE Transactions on Computers 35(4); 1986; pags. 288-295

[Taylor et al, 1980] TAYLOR, D.J.; MORGAN, D.E.; BLACK, J.P.; "Redundancy in data structures: Improving software fault tolerance"; IEEE Transactions on Software Engineering 6(6); 1980; pags. 585-594

[Tvedt et al, 2002] TVEDT, R.T.; COSTA, P.; LINDVALL, M.; "Does the Code Match the Design? A Process for Architecture Evaluation"; 18th IEEE International Conference on Software Maintenance (ICSM'02); Los Alamitos, CA: IEEE Computer Society; 2002; pags 393-401

[Welsh and Han, 1994] WELSH, J.; HAN, J.; "Software Documents: Concepts and Tools"; Software Concepts and Tools; Vol. 1 no. 1, 1994