MORALE METHODOLOGY GUIDEBOOK
Methodology Guidebook for Synchronized Refinement

Spencer Rugaber
College of Computing
Georgia Institute of Technology
spencer@cc.gatech.edu
(404) 894-8450

Abstract

Synchronized Refinement is a methodology for reverse engineering software for the purpose of documentation, reengineering, or reuse. Reverse engineering produces a high-level representation of a software system from a low-level one, typically the program itself. Synchronized Refinement constructs such a representation for a system from its source code. It proceeds by detecting design decisions in source code and relating the detected decisions to the intended functionality of the system. Examples are given demonstrating the application of the methodology to the reverse engineering of production software systems.

1. Introduction

1.1 Purpose

To effectively evolve a system, you must understand it. And that understanding must be expressed in a form with which high level (architectural) decisions about the evolution of the system can be made. *Synchronized Refinement* is a method for obtaining such an understanding and expressing it in a form that aids evolution. The process of gaining understanding and expressing it is called *reverse engineering*. Chikofsky and Cross [Chikofsky] define reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." Reverse engineering can be used for a variety of purposes: to reconstruct or otherwise improve documentation; to facilitate software maintenance or conversion activities; or to redesign and evolve an existing system.

Reverse engineering is a difficult and costly activity. The gap between the low-level source code expressing *how* computations are performed and a high-level description of *why* a particular construct is employed comprises both differences in levels of abstraction and domains of discourse. These two differences are confounded by the fact that understanding process is not uniform, moving as it does between the abstraction levels and the domains.

The Concrete and the Abstract: Computer programs are incredibly detailed. In essence they control the values of million of bits of memory inside of a computer. One of the jobs of the reverse engineer is to decide, from all this detail, which are the important concepts. This process is called abstraction; the reverse engineer must create an abstract representation of the program from the mass of concrete details. Compounding the problem is the fact that the abstraction process is not linear. That is, a given section of a program may be a part of several abstractions. The occurrence of such abstractions can be called *interleaved* or *delocalized* in the section [Pinto Letovsky Documentation Delocalized]. Typically there is no explicit indication in the source code of the interleaving. And detecting the abstractions in the first place involves complex pattern recognition. in which, unfortunately, some of the abstractions reside in the application domain where they are less accessible to the reverse engineer.

The Application Domain and the Program Domain: Programs are models or representations of problem situations from some application domain. There may or may not be hints in the program about the domain. Hints can take the form of mnemonic variable names and in-line comments. The hints are inherently informal and tend to be out-of-date with respect to the program. Because of this, totally automatic reverse engineering tools are restricted to working with the formal program text. As Brooks points out, it is the job of the reverse engineer, to reconstruct the mappings from the application domain to the program domain [Ruven Brooks Comprehension]. This of course requires knowledge, not only of programming, but also of the application domain. It is no surprise, therefore, to find that most automatic tools are restricted to analyzing the program text and do not address the application domain.

Top-Down and Bottom-Up: When a reverse engineer looks at a program, he or she is detecting patterns that indicate the intent of some section of code. Low level patterns are part of higher level constructs intended to accomplish larger purposes. In this way, the process of analyzing a program proceeds bottom-up. At the same time, the programmer has some idea of the overall purpose of the program and how it might be accomplished. As the program is perused, the overall concept is refined into lower level details. This synthesis process proceeds top-down. The difficulty is that both of these activities need to proceed at the same time and in a synchronized fashion.

This document describes *Synchronized Refinement* (SR) a reverse engineering methodology that simultaneously analyzes source code and synthesizes an application description. The analysis is based on detecting examples of characteristic design decisions in the code. The synthesis is driven by a formal description of the application domain. The resulting represention expresses the relationship between the two by instantiating the domain description with the detected code. The representation can then be used to support further software evolution via reengineering.

1.2 MORALE

SR is part of the MORALE software evolution methodology. MORALE is a method and tool suite that supports software evolution. It exploits the symmetry between requirements analysis and software understanding. When analyzing the requested changes for a new version of a system, the old version must be considered and every effort made to reuse existing infrastructure. Likewise understanding the old version can be greatly aided by an application-oriented description of the goals that the system is trying to accomplish, as obtained from its requirements specification. MORALE uses the system architecture as a common ground to synthesize the two sources of knowledge and behavioral traces (called *scenarios* at the requirements stage and *event traces* at the code understanding stage) as the common medium of comparison. The MORALE architecture can naturally be broken up into three activities, one each for requirements analysis, architectural evaluation, and reverse engineering. The relationships between these activities and the artifacts relied upon and produced are depicted in the data flow diagram shown in Figure 14.

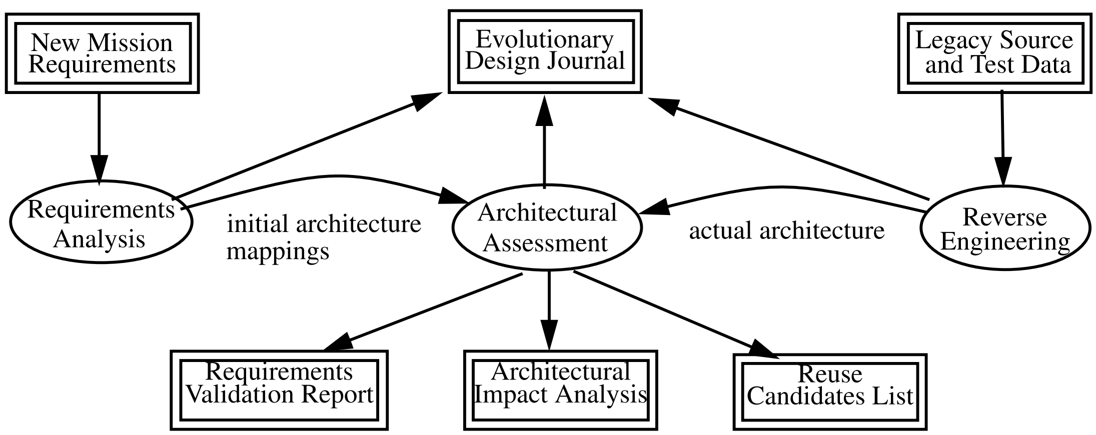

Figure 14 MORALE inputs, activities and outputs

The initial inputs to the MORALE process are the existing system in the form of legacy source code and test data, a statement of the new requirements that the system must meet in the form of mission-oriented goals, and (possibly) an architectural representation of the existing system. Requirements analysis uses the mission-oriented statement to suggest concepts of importance in the overall structuring of the system; that is, it provides a suggested taxonomy for an initial architectural description of the system in the case where no such architecture exists. Requirements analysis also provides a collection of scenarios that represent a complete description of the new system requirements. These serve as the basis for the architectural impact analysis. The architectural assessment predicts the impact a set of scenarios have on an architecture. Reverse engineering provides techniques for assuring that the architectural representation, the basis for the impact analysis, is an accurate reflection of the actual system under scrutiny. The three pieces work together to predict the extent to which the architecture of the old system can be adapted to meet the new requirements.

Besides the architectural impact analysis, MORALE produces several other artifacts useful in planning system evolution: a list of candidate code components from the old version that may be reused in the new version and a report on the extent to which the behavior of the existing system version is reflected in the requirements for the new version. Additionally, the MORALE suite provides a historical evolutionary design journal that incorporates items such as system goals and problems, traceability information between requirements and code, design alternatives and rationale. The result of the

MORALE process is the development of instantiated architectural components together with a record of design decisions and the mission-oriented goals that were met or traded off to develop the new system.

1.3 Overview of Synchronized Refinement

The overall SR process is presented in Figure 2.

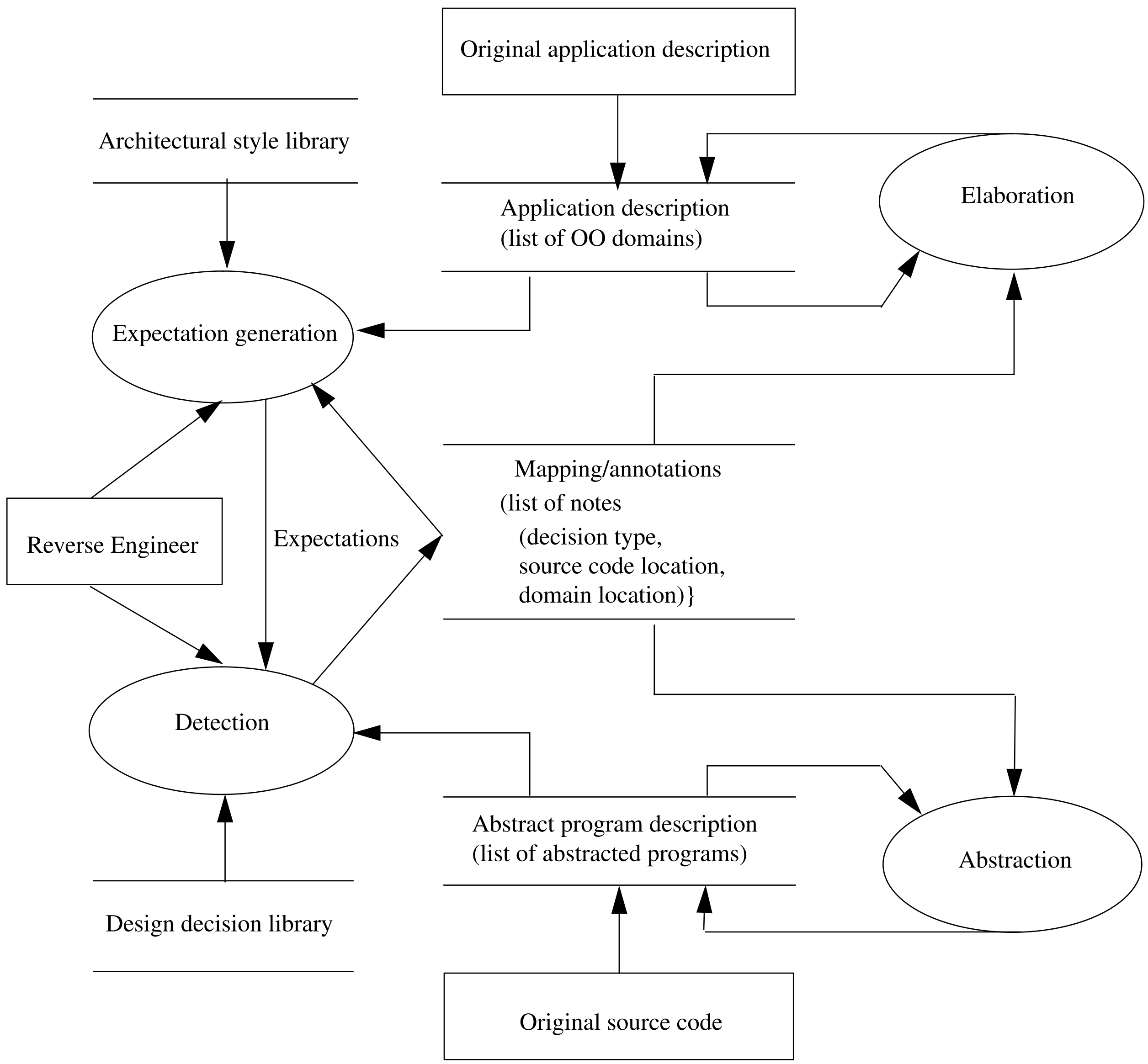

Figure 15 Data Flow Diagram for Synchronized Refinement

Synchronized Refinement takes as input the source code of a software system (labelled **Original source code** in Figure 2) and a description of the application domain that the system supports (**Original application description**). In addition, various sources of programming knowledge may be available to the analyst to aid the understanding process. These are labelled **Architectural style library** and **Design decision library** in Figure 2. The output consists of three parts: an elaborated and instantiated domain description (**Application description**), an abstracted program description (**Abstract program description**), and a description of how the code realizes the application (**Mapping/annotations**). The SR process consists of two parallel activities, roughly corresponding to the top and bottom halves of the diagram: analysis of the program (on the bottom) and synthesis of the application description (on the top). The analysis process consists of detecting design decisions and replacing the related code by an abbreviated version. In this way, the program description continues to become smaller and more abstract. The synthesis process begins with the domain description that is elaborated as more is learned from the source code. The two parts of the diagram communicate in terms of *expectations*. That is, domain concepts must be ultimately realized in terms of code constructs, and by navigating the domain description, expectations are established for what should be found in the source code. Likewise, the process of code analysis will confirm or deny that certain expectations are satisfied.

As the code analysis process proceeds, decisions are detected that relate to an expectation. The annotation for the detected decision is recorded together with the relevant expectation, specifying the type of the decision and the corresponding sections of the program (in the repository labelled *Mapping/annotations*). During the process, certain expectations will be confirmed and others may be refuted. A confirmed expectation may lead to others. Gradually, a hierarchical description of the structure of the program emerges. As the program source code shrinks, the domain description expands.

Synchronized Refinement is a labor-intensive process. As such, its value is dependent not only on its accuracy but also on the degree of precision and thoroughness required of an analysis. Ultimately, of course, the value of SR also depends on the extent to which its output representation proves useful within the larger context of system evolution.

1.4 Example Usage

Section 2 of this guidebook describes the use of Synchronized Refinement in support of the evolution of Mosaic internet web browser. In this scenario, it is assumed that an evolution requirement has arisen to add *mailto* URLs to those types understood by the browser. A MailTo URL, instead of immediately switching the browser to another HTML page, presents a window into which the user can type a mail message to be delivered to a party specified in the URL. The question to be addressed is “What parts of the Mosaic source code should be altered to accommodate this change?”

2. Tutorial Example

3. Detailed Methodology Description

3.1 Input formats

The Synchronized Refinement process takes its input from four sources: the source code, an application domain description, and programming knowledge in the form of an architectural style library and a design decisions library.

Program source code: The program to be analyzed is written in some programming language. It is assumed by SR that the analyst has a working knowledge of the language and that the program successfully compiles. Actually correct parsing and static semantics are assumed. In most cases, SR is not concerned with the vagaries of compiling to specific platforms.

Domain description: TAn SR application domain description takes the form of an object-oriented (OO) framework [johnson]. This comprises a set of cooperating abstract classes as well as the protocols by which they communicate. A class corresponds to a domain concept and its associated *features*. A feature is either a public method that objects of the class can provide or a component of the object’s state (an instance variable). The specific OO notation is not of concern to SR because the details of method implementation are not present. Instead, a specification of functionality is required expressed in a model-based specification notation such as Z [] or VDM [].

The classes in an application domains collaborate to accomplish the application’s goals. The collaboration can be thought of in terms of associations among the classes in the framework or as protocols for interobject messaging by which the classes cooperate to accomplish their ends. Associations are notated using both standard constraints (i.e. cardinality constraints) and invariants specified in the same notation used to indicate method functionality. In the case of protocols, path notation is used []. This is a variant of regular expressions denoting allowable sequences of method invocations among the constituent classes. The protocol together with the component classes can be thought of as providing an architecture for solutions to problems residing in the application domain.

Architecture style library: The SR style library is a mapping between a style name and a domain description. The library is populated with commonly occurring styles such as *client-server* and *pipe-and-filter.* The library serves as a repository for generating expectations and quickly elaborating a domain description once an expectation is ratified.

Design decision library: Programmer elaborate a design into a program by using programming language constructs to implement various kinds of *design decisions*. For example, the programmer may decide that all parsing of URLs should be done in a central place in the code rather than in components specific to a particular URL types. Reverse engineers using SR try to detect instances of such decisions by applying both their knowledge of the application domain and their experience in programming. The design decision library is a repository of such knowledge. Specifically, the repository is organized as a hierarchical classification of design decision prototypes. Prototypes are described more completely in Section 3.3 below.

3.2 Output formats

Output from SR consists of three artifacts: an **application description**, an **abstract program description**, and a mapping between the two (**mapping/annotations**). In addition, it may be the case that the **design decision library** or the **architectural style library** are updated if examples of missing generic programming knowledge are detected in the code.

Abstract program description: As SR proceeds, examples of particular categories of design decisions are detected in the source code. Once such an example is recognized and confirmed, it can be replaced in the code by an abstract description of its role. In the parsing example mentioned above, the *parse mailto* function can be abstracted to the generic *parse* method, possibly parameterized by any mail-specific information. This results in two versions of the program: the original and the (shorter) abstracted version. Two things should be noted about this representation. First, the detected example may not occur contiguously in the program text. This implies that what is actually extracted from the original version may not correspond to a single syntactic construct. SR refers to these constructs as *plans*, and, for the purposes of this example, can be thought of merely as a collection (set) of related syntactic constructs. The second thing to note is that SR does not require that both the original and abstracted version of the code be retained for all detected decision as long as such versions can be reconstructed if required.

Application description: As understanding of the code grows, it needs to be represented so that it can be used for its intended purpose such as reengineering, reuse, or documentation. The form that SR uses for this is to instantiate the OO classes used for the **domain description**. For example, if the net browser application domain contains a class for URL parsing, and if the code analysis process actually detects the code that accomplishes this, then the abstract parsing class can be instantiated with the identified code.

Mapping/annotations: The artifacts described in the previous two sections must be synchronized. That is, it must both be possible for an analyst to locate the specific code that implements a domain concept and to determine for any code construct what role it fills in the application domain. This two-way mapping is provided in the form of a set of annotations. An annotation consists of three pieces of information. The first is the type of design decision that the reverse engineer detected. This corresponds to an actual entry in the design decision library. If no such site exists, then the library needs to be updated appropriately. The second piece of information indicates the specific place in the source code at which the design was implemented. This corresponds to the set of syntactic constructs described above. The third piece of information is the place(s) in the OO framework where the code segment has been instantiated. Together the three pieces of information allow the reconstruction of the understanding process and provide the raw material for a permanent repository (the Evolutionary Design Journal).

3.3 Steps

The two key concepts underlying Synchronized Refinement are abstraction-based design decision detection in source code and domain-driven application model synthesis. The two ideas are manifested simultaneously (*synchronously*) and an annotated representation is generated linking the results. The source code analysis process proceeds by detecting program constructs and abstracting them from the code, thereby compressing it. The application description synthesis process explores the input domain model in order to generate target concepts for detection in the code and to build up the application description. In this way, the source code continually grows shorter while the application description grows more complete.

Design Decisions: Software design is the process of expressing structure, functionality, and behavior using the constructs of a particular programming language. If we want to effectively evolve software then it makes sense to reuse as much of the original version as possible. This means reusing not only the code, but also the architecture and other design choices. To do so, it is important to take advantage of as much of the reasoning that went into designing the existing version in deciding how to proceed with the new version. This reasoning is collectively called *design rationale*, and Synchronized Refinement structures the rationale that it detects based on the abstraction mechanisms used in the program's source code. The abstraction mechanisms are implementations of *design decisions* made by the original programmer or designer. For example, choosing a dynamic data structure versus one with fixed bounds reflects the decision that an original requirement to allow for an "unlimited" number of items rather than arbitrarily constraining the implementation to a fixed length is an example of an explicit decision implemented by using the programming language's dynamic memory allocation features rather than statically allocated a fixed-length structure.

Design decisions can be divided into several classes based on the type of abstraction they provide. For example, many languages have a feature for bundling together or packaging a group of named constructs, to better control naming conflicts and enforce information hiding. This is an example of the *encapsulation* abstraction. Among the classes that are useful to detect during the process of reverse engineering are the following.

- **Composition/decomposition:** Programs are built up from parts, and problems are broken down into smaller, more easily solvable sub-problems. This type of decision is manifested in the programming language code by such constructs as modules and data structures.
- **Encapsulation/interleaving:** Subcomponents interact with each other. If the interactions are limited and occur through explicit interfaces, the component is said to be encapsulated. If, usually for reasons of efficiency, two or more

plans are realized in the same section of code or by the same data structure, then the components corresponding to those plans are said to be *interleaved*.

- **Generalization/specialization:** Often one component is similar to another. It may then be possible to construct a more general and higher level, parameterized component capable of realizing both as special cases. In object-oriented programming, the process is sometimes reversed, with the more general component constructed first, and the special cases added later.
- **Representation**[1]: In translating from the problem domain to the solution domain, decisions are made that result in a program component serving as a model for some application domain entity. If efficiency is a concern, high level programming constructs can be further represented by other constructs closer to the machine, such as using an array to represent a stack. Languages such as Ada are emerging that support explicit representation, but the reverse engineering of programs from older languages requires the detection of these decisions.
- **Data/Procedure:** Programs are sequences of computations organized by control structures. Variables are ways of saving intermediate results for later use, either to avoid recomputation or to simplify the expression of the computation. The introduction of a variable is an important design decision that is, unfortunately, too easy to make without appropriate thought and annotation.
- **Modality:** In some situations, a designer has a choice of how to express the relationship between input and output parameters. This is particularly true in logic programming languages, such as Prolog. For example, imagine the relationship between two arrays, both of which contain the same elements, one of which is ordered. If the ordered version is offered as input and unordered versions are output on subsequent calls, then the program is a permutation generator. In the situation where the unordered array is offered as input and the ordered version is output, then the program is a sorter. A single relationship, expressing a high level specification, can lead to alternative functions depending on the modes (input/output designations) of the parameters. This decision is usually made at a very early stage of design, if it is made explicitly at all.

There is no strict order for detecting design decisions. Complex programs are typically designed as layers of abstraction, and, as one layer is detected during the reverse engineering process, it opens the door for the detection of other decisions in higher layers. Nevertheless, it is possible to give the following guidelines for the detection process.

- Particularly where older programming languages are used, begin by looking for the occurrence of modern control structures implemented by more primitive constructs. An example of this is the use of nested **IF** statements and **GOTO**s to implement a **SWITCH** statement.
- Likewise, look for the use of primitive data structures to represent unavailable ones, such as the use of specific integer or character constants to encode one of a set of values.
- It may sometimes be necessary to restructure the control flow of the program. This is essentially what a restructuring tool provides. The resulting code should be functionally equivalent to the original.
- Look for special cases; that is, look for similar sections of code that differ only in a small number of ways. Replace these by the parameterized use of a more abstract construct.
- If the language does not support modularization, look for code that should be grouped together and separated from the rest of the program by an abstract interface.
- In contrast, sometimes a programmer intentionally interleaves the accomplishment of two objectives within the same section of code. This is often done for reasons of efficiency. To understand the code, it becomes necessary to segregate the two functions and annotate them separately.
- If a collection of data items are used together to implement an unavailable construct, annotate that fact.
- Make sure that variables are only used for one purpose. Replace dual-use variables by two separate variables.

Design Decision Repository:

Domain analysis: TBS

3.4 Evaluation

4. Glossary

1. Design decision: An explicit refinement of a program design. Each decision reflects the elaboration of a design or pro-

1. This use of the term *representation* should not be confused with its use as a notation for capturing a high-level understanding of a program.

gramming plan into a more concrete realization.

2. Domain: A problem area. Typically, many application programs exist to solve the problems in a single domain. The following prerequisites indicate the presence of a domain: the existence of comprehensive relationships among objects in the domain, a community interested in solutions to the problems in the domain, a recognition that software solutions are appropriate to the problems in the domain, and a store of knowledge or collected wisdom to address the problems in the domain. Once recognized, a domain can be characterized by its vocabulary, common assumptions, architectural approach, and literature. [Arango Pri*'eto-Diaz 1991]
3. Domain model: "The domain model should serve as a unified, definitive source of reference when ambiguities arise in the analysis of problems or later during the implementation of reusable components, a repository of the shared knowledge for teaching and communications, and a specification to the implementer of reusable components." "A model of a domain should include information on at least three aspects of a problem domain: concepts to enable the specification of systems in the domain; plans describing how to map specifications into code; and rationales for the specification concepts, their relations, and their relation to the implementation plans." [Arango]
4. Delocalization: The phenomenon where "programming plans [are] realized by lines scattered in different parts of the program." [Letovsky86]
5. Object oriented Framework: [Johnson]
6. Interleaving: "The merging of two or more distinct plans within some contiguous textual area of a program." [icsm95]
7. Program plan: A description or representation of a computational structure that the designers have proposed as a way of achieving some purpose or goal in a program. [Letovsky86,PA]
8. Rationale: A record of the reasoning used to produce a software artifact from a design specification.
9. Software evolution: A continuous change from a lesser, simpler, or worse state to a higher or better state [for a software system]. [Arthur]
10. Software understanding: Program comprehension - "the process of acquiring knowledge about a computer program." [encyc]
11. Reverse engineering: "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." [Chikofsky and Cross]
12. Synchronized Refinement: a method for reverse engineering software system comprising both bottom-up program analysis and top-down application domain synthesis.

5. Annotated References to the Literature

How this method relates to other methods.

Discriminators

Links to further information

[24] Arango, Guillermo and Prieto-Diaz, Ruben. Domain Analysis Concepts and Research Directions, in Domain Analysis and Software Systems Modeling, ed. Ruben Prieto-Diaz and Guillermo Arango, IEEE Computer Society Press, 1991.

[25] Arango, Guillermo. Domain Analysis Methods. In Software Reusability. (Eds.) W. Schaeffer, R. PrietoDiaz, and M. Matsumoto. Ellis Horwood, New York, 1993, pp. 17-49.

[26] Automated Data Systems Manual, Installation Material Condition Status Reporting System (IMCSRS), Functional Users Manual, Commander FORSCOM, AFLG-RO, Ft. McPherson, Georgia, April 1, 1984.

[27] Campbell R. H. The Specification of Process Synchronization by Path-Expressions. In Lecture Notes in Computer Science, pages 89-102, 1974.

[28] Campbell R. H. and Islam, N. A Technique for Documenting the Framework of an Object-Oriented System. Technical report UIUCDCS, University of Illinois at Urbana-Champain.

[29] DeBaud, Jean-Marc. From Domain Analysis to Object-Oriented Frameworks, A Reuse Oriented Software Engineering Methodology. Thesis Proposal, CIMR TR# 94-04, Georgia Institute of Technology, January 1994.

[30] DeBaud, Jean-Marc, Moopen, Bijith, and Rugaber, Spencer. Domain Analysis and Reverse Engineering, Proceedings of the Conference on Software Maintenance, pp. 326-335, Victoria, British Columbia, September 1994.

[31] Devambu, Prem. GENOA/GENII A customizable, language and front-end independent code analyzer, Fourteenth International Conference on Software Engineering, Melbourne, Australia, 1992.

[32] Devambu, Prem and Frakes, Bill, Extracting Formal Domain Models from Existing Code for Generative Reuse. Unpublished Technical Report.

[33] Frakes, W.B., C.J. Fox, B.A. Nejmeh. Software Engineering in the UNIX/C Environment, Prentice-Hall, 1991.

[34] Hildreth, Holly. Reverse Engineering Requirements for Process-Control Software, Proceedings of the Conference on Software Maintenance, pp. 316-325, Victoria, British Columbia, September 1994.

[35] Johnson, Ralph E. and Foote, Brian. Designing Reusable Classes. Journal of Object-Oriented Programming, June/July 1988, Volume 1, Number 2, pp 2235.
[36] Katz, S., Richter, C.A., The, K. PARIS: A system for reusing partially interpreted schemas. In 9th International Conference on Software Engineering (Monterey, Calif. Mar. 1987). IEEE Computer Society Press, Los Alamitos, Calif., pp. 377-385.
[37] Lubars, M. Domain analysis and domain engineering in IDeA. In Domain Analysis and Software System Modeling. IEEE Computer Society Press, Los Alamitos, Calif. 1991. pp 163-178.
[38] Neighbors, James. Software Construction from Components, PhD thesis, TR-160, ICS Department, University of California at Irvine, 1980.
[39] Neighbors, James. DRACO: A Method for Engineering Reusable Software Systems. 1989 ACM, Inc. Addison-Wesley Publishing Co., Reading MA.
[40] Peterson, S., Kang, K., Cohen, S., and Hess, J. Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA 15213, Nov. 1990
[41] Reasoning Systems, Inc., Palo Alto, CA. REFINE Users Guide, 1990. For REFINE (TM) version 3.0
[42] Simos, M. The growing of an Organon: A hybrid knowledge-based technology and methodology for software reuse. In Domain Analysis and Software System Modeling. IEEE Computer Society Press, Los Alamitos, Calif. 1991. pp 204-221.