A Model for Reuse-in-the-Large

Haikuan Li Jan van Katwijk

Delft University of Technology
Faculty of Mathematics and Computer Science
Julianalaan 132, Delft, The Netherlands
Tel: 31-15-784433, Fax: 31-15-787141
Email: li@dutiaa.tudelft.nl

Abstract

In this paper, a model for Reuse-in-the-Large (RITL) is proposed. RITL aims at providing large-grain components and giving information about the creation of these components, in order to reuse these components and the information to improve the process of software development and maintenance.

Keywords: large-scale component, design framework, design instance, domain resource.

1 Introduction

The technologies applied to software reuse may be classified into two groups, i.e. reuse-in-the-small and reuse-in-the-large. The former is centered on the reuse of small code components such as objects in object-oriented programming or packages in Ada[Bigg 89], while the latter is centered on the reuse of large-grain code components such as subsystems and the information about their designs.

Following [Li 91], we believe that, although reuse-in-the-small is a necessary part of software reuse and is moderately successful in some areas, it is fundamentally limited by our lack of information about the whole process of software development and maintenance in which the reuse is going to take place [Li 91]. Therefore, our research attention is drawn to reuse-in-the-large.

In this paper, a model for reuse-in-the-large is presented. This model aims at providing large-grain components and giving information about the creation of these components, in order to improve the process of software development and maintenance by reusing both these components and the information.

The motivation to establish the particular model is derived from observing forward engineering and reverse engineering[Chik 90]. In forward engineering a lot of information remains undelivered

[Webs 88] whereas, for the sake of reuse, in reverse engineering quite a lot of similar information has to be recovered [Chik 90]: software producers make puzzles that reverse engineers solve.

As Webster points out[Webs 88], conventional software methodologies are product-oriented; they were developed to address design artifacts, they do not adequately address representation needs about the creation of the artifacts. Accordingly, we need a means to strengthen software representation and we need to address research questions: (1) How to make a large-grain component reusable, i.e. how to make a large-grain component easy to maintain, evolve and reconstruct in order to adapt the component to similar applications. (2) How to organize, manage, and manipulate a collection of reusable (large-grain and small) components in order to improve the process of software development and maintenance.

In this paper, we distinguish *design representation* from *implementation*. A design representation refers to the output of the design phase of a software life cycle, such as a design specification, while an implementation refers to the output of the implementation phase, such as executable code. Consistent with this distinction, we use the term *design information* to refer to the information concerned with design, the information about design and the information about design process [Webs 88].

In section 2 the criteria and the strategies of reuse-in-the-large are presented. In section 3 the model for reuse-in-the-large is addressed. In section 4, 5 and 6 the concept, structure and features of large-scale components are described. In section 7 we demonstrate how the process of software development and maintenance may be improved by the reuse of large-scale components. Finally, in section 8 the conclusion is provided.

This work is part of a reuse project in Delft University of Technology, The Netherlands.

2 The criteria and strategies of reuse-in-the-large

Criteria normally refer to a group of standards by which something can be judged or decided. In order to judge the capability of reuse-in-the-large and evaluate the strategies to realize reuse-in-the-large, several criteria are addressed as follows.

- *capability of representing large-grain components and design information,*
- *capability of generating a group of large-grain components,*
- *capability of applying organizational principles for component representation including information hiding, genericity, tailoring and so on, and*
- *capability of enhancing the process of software development and maintenance in terms of reuse.*

The strategies which we address in order to realize reuse-in-the-large, are

- *design-centered reuse,*

- *domain-oriented reuse,*
- *architecture-based reuse, and*
- *life-cycle-supported reuse.*

In the following part of this section, we outline the strategies and evaluate them according to the criteria.

2.1 Design-Centered Reuse

Software reuse may refer to the reuse of code components, the reuse of design information, or the reuse of both. Design-centered reuse refers to the reuse of both code components and design information in such a way that the design information is taken as the key to reuse. Representation of design information should be reused in the phases of software design and software analysis; while furthermore it should allow to provide an abstraction to support the reuse of code components.

In order to provide an abstraction of a large-grain component, design-centered reuse suggests that the structure of a large-grain component should contain the different levels of abstraction of the component. The structure of the large-grain components describes the organization of the design specifications, the implementations and other information about the design of the components. The different levels of abstraction may be derived from the process of software design which is seen as successive elaborations of design representation[Webs 88]. The design representation may be given in terms of specification languages. The relationships between the different levels of design representation is: a higher level design representation acts as the abstraction of the lower level design, and the lowest level design representation acts as the abstraction of its implementation.

Design-centered reuse aims at providing guidelines to bind small components into the structures of large-grain ones, offering information to understand and reuse large-grain components, and giving support for the maintenance, evolution and reconstruction of the components.

2.2 Domain-Oriented Reuse

Software reuse often refers to the reapplication of the same software artifacts to meet requirements in the same or different applications (*use-as-is*)[Bass 87]. However, in most cases, rather than the same artifacts, similar artifacts are required[Bass 87]. Domain-oriented reuse emphasizes the capability of generating a collection of similar large-grain components, in order to meet similar requirements on the same domain (Note: *Domain* refers to a set of current and future applications marked by a set of common capabilities and data [Pete 91]). So, whereas some other kinds of software reuse aim at reducing the redundant efforts in the design of the same artifacts, domain-oriented reuse aims at reducing the redundant effort in the design of similar artifacts in a domain.

Domain-oriented reuse is based on the apparent commonality in the capabilities and data about a set of similar applications of the same domain. For example, when an operating system must

be designed, the designers have to identify the domain by simply answering the question '*what operating system is*'. Such a question together with its answer is domain information. Furthermore, we think that the design decision about a particular design is usually concerned with the selection of an appropriate design instance from a collection of alternatives[Ruga 90]. For example, when an application is concerned with *sorting*, a designer usually investigates the properties of a group of components for sorting, such as *Quick-Sort, Heap-Sort, Merge-Sort* and so on.

2.3 Architecture-Based Reuse

The key organizational principle used to form modules is *information hiding*. However, as indicated by Parnas [Parn 89], applying this principle as a basis for developing software components is not always easy because additional design information is required when decisions have to be made about the change of the components. Furthermore, currently used techniques for information hiding, typically leading to *black-boxes*, may not be directly applicable for large-grain components (subsystems or even systems) without loosing the ability to reuse[Li 91], due to the heavy degree of parameterization required for reusing the components. This is because the changes of large-grain components are too difficult to be represented by means of parameterization[Phil 87].

Architecture-based reuse provides an organizational principle for organizing both large-grain code components and design information into a unique software component, which supports a flexible change of the component. The term (software) *architecture* refers to the organization of functions and objects, their interfaces, and the control to implement applications in a domain [Pete 91]. The term *flexible change* refers to the capability of maintenance, evolution and reconstruction of large-grain components.

It is our opinion that the architecture of a large-grain component takes the design and the subdesigns of the component as black-boxes at different abstract levels. Such an architecture implies that each of the black-boxes may be opened once the detailed design has to be modified.

Using the different levels of abstraction is beneficial to both the understanding and the manipulation of large-grain components. First, we believe that, in developing and understanding complex phenomena, the different levels of abstraction are the most common and most efficient tools to human intellect. Second, in software engineering practice, the methods of software development are commonly based on object-oriented decomposition or algorithm decomposition[Booc 91], yielding the different levels of abstraction. Therefore, the history of decomposition can be tracked down if the architecture of a large-grain component provides the different levels of abstraction, as discussed before. Maintaining such history allows guiding reusers to browse through the architecture of the components, to locate elements to modify, and to supply necessary information and resources for any required modification.

2.4 Life-Cycle-Oriented Reuse

The term life-cycle-oriented reuse refers to reuse during the whole life-cycle of software, including analysis, design, implementation and maintenance. Such a reuse requires a means of representing the information concerned with each phase of the life-cycle and to develop tools to support the reapplication of the information.

Although many kinds of information can be reused, at present, only small pieces of information can efficiently be formalized and efficiently be reused[Webs 88]. Our strategy is, therefore, to identify the information which can be formalized and adopted in the practice of software development.

First, diagrams are perhaps the best-known means of representing software analysis and design. The basic advantage of using diagrams such as data-flow graph is their graphical nature, although their basic disadvantage is the lack of semantics [Katw 90].

Secondly, we select design specification as the information of design phase. Such specifications may be either semantic specification in terms of VDM[Jone 86], Z[Lano 89] and so on, or specifications such as MILs[Prie 86] and EDFG[Katw 90]. Such design representations provide semantics of the design artifacts or bring about abstraction of the artifacts.

Thirdly, in order to support software maintenance, we may build the connection between the different levels of decomposition, representing the history (snapshot) of software development. Such a connection may provide an architectural super structure[Bigg 89] that binds small components into a large-grain one, and may be used for the understanding, the modification and the manipulation of large-grain components.

Finally, in order to support the evolution and reconstruction of large-grain components, we further select information to represent design frameworks (generic designs). as well as the resources for the instantiation of the frameworks.

We are of the opinion that – if the information discussed above is reused efficiently – the life-cycle of software development and maintenance can significantly be improved. The questions are, however, how to organize these kinds of information and how to apply them to software life-cycle.

3 What is the model?

The model. RITL (Reuse-in-the-Large) is a model for software reuse, concerned with methods and a support environment for the creation and reapplication of *large-scale components*. Applying this model aims at improving the process of software development and software maintenance.

Large-scale components. A large-scale component is an integral representation of a software component and containing both the code component and the information about its design. A large-scale component may be as large as a module (a package of a set of interrelated objects as addressed in object-oriented design[Booc 91]), a subsystem or even a complete system. The advantage of using large-scale components over using small code components is that a large-scale component provides a basis for the construction of similar large-grain components by reusing both code and design

information.

The support environment. The envisaged support environment of RITL encompasses a component base, which provides a repository for large-scale components, and a design assistant, i.e. a set of tools to represent and manipulate large-scale components. The examples of these tools are: tools to specify the design of large-scale components; tools to browse through the structure of the whole or any part of a large-scale component; tools to generate implementations (large-grain code components) from large-scale components; tools to support the construction of design and tools to modify and reconstruct a large-scale component.

Reuse, cost and limitation. RITL aims at improving the process of software development and maintenance by the reuse of large-scale components. As indicated, this kind of reuse is very different from the reuse of small code components. RITL supports the reuse of the super structure that small components may be bound into. Moreover, the classification problem and the search problem[Neig 89] can be minimized by the reuse of design information appended to the structure, as discussed later on. This design information specifies the binding of small components into the super structure in order to generate a large-grain component. Therefore, the cost of reusing small components on particular building the super structure[Bigg 89] may be significantly reduced. Finally, there is the limitation that whereas a small code component may be reapplied to many different application domains, the reuse potentials of a large-scale component are logically limited to a single domains, although there has nothing to reject the normal reuse of small components.

4 The concept of large-scale components

A large-scale component may be formally defined as 4-tuple:

$$\text{large-scale-component} = (\text{design-framework}, \text{design-instance}, \text{domain-resources}, \text{refinement})$$

Design framework is an abstraction of a scope of similar components in a problem domain. A design framework consists of a *specification* and a *structure*, the former specifying the generic design of the similar components, the latter describing the analysis corresponding to the design. Both of them are centered on a set of abstract (or non-abstract) objects and the relationships between the objects. The semantics of a design framework acts as an algorithm which tells us how to organize lower level entities (objects and relationships) into alternative design instances.

Design instance is an instance of design framework, a representation of a specific design. A design instance consists of a *specification* and an *architecture*. The specification of a design instance specifies the design of a large-grain component and describes the dependency between this component and other components. The architecture consists of a set of objects and the relationships between them.

Domain resources consist of a *domain model* and a set of *domain entities*[Pete 91]. The domain model is the definition of the abstract objects and relationships appearing in the design framework.

The domain entities provide the instances of the abstract objects and relationships being defined. The domain entities are the values of the domain model.

Refinement is the detailed design or the implementation of the design instance. A refinement consists of a set of (lower level) large-scale components or implementations. Each of the (lower level) large-scale components or implementations is an elaboration of an object appearing in the design instance, and such an elaboration inherits the semantics of this object and relationships between this object and others. The refinement may be recursively continued until all objects in the design instance are suitable to be implemented in terms of small components, such as objects in object-oriented programming language or the packages in Ada.

Figure 1: The structure of a large-scale component

5 The structure of large-scale components

As a large-scale component is defined recursively, a hierarchical structure can be identified from it. The hierarchical structure can be identified by taking the *refinement* of a large-scale component as lower level representation of the large-scale component. Such a hierarchical structure can be

formally defined as follows.

$$\begin{array}{lll} \text{H-Structure} & ::= & \textit{record} \\ & & \text{design-framework;} \\ & & \text{design-instance;} \\ & & \text{domain-resources;} \\ & & \text{refinement;} \\ & & \textit{end;} \end{array}$$

$$\text{refinement} ::= \left\{ \begin{array}{ll} & \text{a set of small code components;} \\ \textit{and} & \\ & \text{a set of pointers to H-Structures;} \end{array} \right.$$

The nodes only containing code components are called *terminal nodes*, and *non-terminal nodes* otherwise.

From the hierarchical structure, the design information and the implementation of a large-grain component can easily be distinguished. Considering the nodes of the structure, we find that the terminal nodes of the structure form the implementation of a large-grain component, and the non-terminal nodes of the structure contain information about the analysis and design of the component.

Looking into each non-terminal node, three pieces of information can be investigated: design framework, design instance and domain resources. Let DF stand for design framework, DI for a range of design instances, and DR for domain resources. According to the definition of large-scale component, the relationships between DF, DI and DR may be described as follows:

$$DF : DR \longrightarrow DI$$

i.e. a design framework is a mapping from domain resources to a range of design instances.

Additionally, in order to provide an intuitive impression of large-scale components, the structure of large-scale components is sketched in Fig. 1.

6 The features of large-scale components

There are several features of large-scale components which may be used to improve the capability of reusing large-grain code components and its design information.

6.1 Different Levels of Abstraction

A large-scale component provides different levels of abstraction for a large-grain component. At top level, a large-scale component provides specifications of the large-grain component, allowing identification, understanding, and use of the large-grain component at the highest abstract level. Furthermore, we can look into the detailed design of the large-grain component, which is provided by the refinement of the large-scale component. The refinement of a large-scale component refines the

objects which form the design instance. According to the definition of large-scale components, the refinement may be continued recursively, level by level, until the objects to be refined are suitable to be implemented in terms of small code components. The representations derived from the different levels of refinement provide the different levels of abstraction. Obviously, according to the different levels of abstraction, each sub-large-grain component (or sub-sub-large-grain component, etc.) can be identified, understood and reused at an abstract level.

6.2 Information for Generating Similar Components

A large-scale component provides information (i.e. design framework and domain resources) to modify or reconstruct its constitution. A design framework of a large-scale component can be seen as a template which provides a pattern to generate similar components; and the domain resources provide candidates (other reusable components) to instantiate the template. The design instance of a large-scale component is to be seen as an instance of the template, and one or more similar (but different) design instances may also be generated by the instantiation of the design framework. Therefore, similar large-scale components may be generated by replacing one or more design instances with other instances of design frameworks according to the different levels of abstraction. As a result, a large-scale component is a component itself, while, it is, furthermore, a basis for the construction of similar components from this and other reusable components.

7 Outline of reuse-in-the-large

RITL may be characterized by the creation and reuse of large-scale components and the claim is that the process of software development and maintenance can be enhanced by the reuse of large-scale components. In this section we will discuss such an enhancement.

7.1 Reusability Information for Analysis

According to Booch[Booc 91], analysis is the process of modeling the world by identifying real world components such as subsystems, modules and primary objects, which form the vocabulary of a problem domain [Booc 91]. Practically, each large-scale component or its sub-component provides a concept which may be characterized by three dimensions: design framework (a general concept), design instance (an instance of the general concept), and domain resources (sub-concepts and their thesauruses), and these provide a vocabulary.

The output of analysis is information which tells us the required behavior of the system (or component) we must build[Booc 91]. We claim that such information is contained in a large-scale component. The behavior of a large-grain component is described in a large-scale component in two ways: First, the design framework of a large-scale component catches the common behaviors of a set of components of a domain, described in terms of a set of abstract (or non-abstract) objects and their relationships. Secondly, the design instance describes the behavior of specific components

in terms of a set of objects and the relationships between them. While the information provided by a design framework may be used to analyse a collection of components in an abstract form, the information provided by a design instance includes the analysis of an element of this collection.

7.2 Reusability Information for Design

Design is a process of inventing the abstractions and mechanisms that provide the behavior a system requires [Booc 91]. Our claim here is also that the abstraction and mechanisms may be provided by the specification of design framework, the specification of design instance and the domain resources.

The specification of a design framework provides a generic design by representing the commonalities of a set of design instances. Such a specification can be fully reapplied to build a collection of similar design instances in a domain and is, therefore, reusability information for design.

The specification of a design instance provides abstraction for further refinement or accommodates blueprint for implementation. The same specification of a design instance can be shared by either different refinements or implementations and is ,therefore, reusability information for design.

The domain resources are created somewhere else and listed in a large-scale component as candidates for the re-design or re-implementation of a large-grain component, in order to reuse this component in different applications. Therefore, they are reusability information for design.

Moreover, the structure of a large-scale component partially records the history (snapshot) of design decomposition and, therefore, partially supports the reuse of design process. Using this structure, similar large-scale components can be designed by browsing through the structure and modifying the old decisions of design decomposition.

7.3 Implementation Enhanced with Reusability

Traditionally, software reuse is supported by library components. In terms of RITL, large-grain components can be reused, based on the reuse of large-scale components. Firstly, according to the structure of large-scale components, a set of large-grain or small-grain components may be identified from the hierarchical structure of a large-scale component and each of them can be used at specification level. Secondly, since a single large-scale component may easily be modified, evolved, or reconstructed, a range of similar code components can be generated, based on a single large-scale component, as discussed before. Finally, reuse-in-the-large is not a way to deny reuse-in-the-small. A large-scale component may be viewed as a mechanism in which small components are organized in such a way that reusability may be emphasized. For example, instead of retrieving components from library, domain resources provide the components when needed; by identifying the relationships between a component and its related domain resources and design frameworks, the user may not only obtain the thesaurus of the component, but also find a group of application environments where the component may be applied to. The thesaurus of a component refers to the components which are the values of the same domain model of the domain resources. The application environments of a component refer to the design frameworks which can be instantiated with this

component. Therefore, in terms of large-scale components, the process of software implementation may be enhanced.

7.4 Maintenance Enhanced with Reusability

It is generally accepted that the lack of design information and the lack of automated support in component manipulation complicate software maintenance and enhancement.

RITL supports software maintenance by allowing the modification and reconstruction of a subsystem or system (large-grain component) at different design levels. First, when a system is represented as a large-scale component, the design information of the system is distributed on a hierarchical structure. Such a structure can be browsed with a tool of design assistant. Because of the browsing of this structure each particular part of the system can be located when it needs to be modified, enhanced or reconstructed. Associated with such a located part, there is a template (design framework) which may be used to produce a similar design of the part, while furthermore there are domain resources which provide candidates to instantiate the template. Since the elements of domain resources may be other large-scale components, the detailed design and the code of the located part may be provided automatically.

8 Conclusion

In order to support reuse-in-the-large, criteria were addressed and strategies were proposed. Following these strategies, a model for reuse-in-the-large was provided, which is centered on the representation and reuse of large-scale components. After the description about the concept, structure and features of large-scale components, we described how the process of software development and maintenance may be improved by the reuse of large-scale components.

However, as indicated by Biggerstaff[Bigg 89], reuse-in-the-large (or very-large-scale reuse alternatively [Bigg 89]) introduces a whole new set of research problems. The final target of this kind of reuse is to make component representation sufficiently general and to allow reuse over a broad range of software systems. Our model is one step forward towards this final target. Although we are satisfied with our model as it shows already the power of reuse-in-the-large, there is still a lot of work to be done. Our future research will be centered on developing tools concerned with an evolutionary process for the creation of large-scale components and the accumulation of reusability information.

Acknowledgement. We would like to thank our colleagues of the reuse project at Delft University of Technology, the Netherlands, in particular E. M. Dusink and F. Ververs. The many discussions with them have been essential for this work. We would also like to thank Trudie Stoute, who has carefully read through the earlier versions of this paper and provided helpful comments on it.

References

[Aran 89] Arango, Guillermo, *DOMAIN ANALYSIS – from Art To Engineering Dicpline–*, Communication of ACM, 1989 ACM 0-89791-305-1/89/05000/0152.

[Bass 87] Bassett, P. G., "Frame-based Software Engineering" IEEE Software, July, 1987, pp. 9-19.

[Basi 90] Victor R. Basili, *Viewing Maintenance as Reuse-Oriented Software Development*, IEEE Software, Jan. 1990.

[Bigg 89] Ted J. Biggerstaff, Alan J. Perlis, Software Reusability, Vol. I, Concepts and Models, ACM press, Addison-Wesley publishing company, 1989.

[Booc 91] Grady Booch, *Object-Oriented Design with applications*, The Benjamin/Cummings Publishing Company, Inc., 1991.

[Chik 90] Elliot J. Chikofsky, James H. Cross II, *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, Jan, 1990.

[Jone 86] Jones, Cliff B., *System Software Development Using VDM*, Prentice/Hall international, Series in Computer Sciences, 1986.

[Lano 89] Lano, K. and Breuer P. T., *From Programs to Z Specification*, Z User's Meeting, Dec. 1989.

[Li 91] Li, Haikuan, *An Introduction to Software Reuse* , Technical Report 91-50, ISSN 0922-5641, Faculty of Mathematics and Computer Science. Technical University of Delft, The Netherlands, 1991.

[Katw 90] Katwijk, van Jan, A. M. Levy etc. "Software Design for Distributed (Real-Time) systems using EDFG approach", Technical report, Software Engineering, TWI, Technical University of Delft, The Netherlands, October, 1990.

[Levy 90] Levy, A., Katwijk, van Jan, Pavlides, G., and Tolsma. *SEPDS: A Support Environment for prototyping distributed systems*, In Proceedings of the first International Conference on System Integration, New Jersay, USA, Apr, 1990.

[Parn 89] Parnas, D. L., Clements, P. C., and Weiss, D. M., *Enhancing Reusability with Information Hiding*, Software Reusability, Volume I, Concepts and Model, edited by Biggerstaff, Ted J., and Perlis, Alan J., ACM press,

[Pete 91] Peterson, A. Spencer, *Coming to Terms with Software Reuse Terminology: a Model-Based Approach* ACM SIGSOFT, Software Engineering Notes, Vol 16, No. 2, Apr 1991, pp. 45-51.

[Phil 87] Levy, Philip, and Ripken, Knut "Experience in Constructing Ada Programs from Non-Trivial Reuse Models", The Ada companion Series: Ada components and tools, Cambridge University Press, May 1987.

[Prie 86] Prieto-Diaz, Ruben, Neighbors, J. M., *Model Interconnection Languages*, The Journal of Systems and Software 6, 307-337, 1986.

[Prie 91] Prieto-Diaz, Ruben, *Software Engineering*, Communications of ACM, May 1991, Vol.34, No.5, 89-97.

[Neig 89] Neighbors, James M., Draco: A Method for Engineering Reusable Software Systems, Programming, Software reusability, Volume I, Concepts and Models, edited by Ted J. Biggerstaff, Alan J. Perlis, ACM Press, Frontier Series, 1989.

[Ruga 90] Rugaber, S., Ornburn, S. B., and LeBlanc, Jr., *Recognizing Design Decision in Programs*, IEEE Software, January, 1990, pp. 46-54.

[Verv 88] Ververs Frans, Katwijk, Jan van, Dusink, Liesbeth *Directions in reusing software*, technical report 88-58 Faculty of mathematics and Informatics, Delft University of Technology 1988.

[Webs 88] Webster, Dallas E., *Mapping the Design information representation Terrain*, Computer, December, 1988.

9 About the Authors

Haikuan Li has graduated from Beijing University, Beijing, China. He held lectureship in the Graduate School of Academia Sinica, Beijing, China. Since 1988 he has been working and studying for his PhD degree at the Faculty of Technical Mathematics and Informatics, Delft University of Technology, the Netherlands. He is working on the reuse project at this university, and his PhD thesis will be concerned with software reuse.

Jan van Katwijk is a professor of Software Engineering at the Faculty of Technical Mathematics and Informatics. Delft University of Technology, the Netherlands. He received his MSc and PhD degree at this University.

By his work at this university and by cooperating with industry, he has obtained a wide experience in computer science in general and especially in software engineering. In the area of compiler building and programming languages he has given many courses and developed several compilers, including a very large subset of the Ada language. In the area of software engineering he is working on combining the best of the formal and informal development methods for the development of software, both in the real-time and in the data processing domain. His research interests include various aspects of software reuse concerned with software design, formal specification, software environment and programming language issues.

Prof. Dr. Ir. Jan van Katwijk acted as a consultant both for Ada and for general software engineering principles for several companies in the Netherlands and, apart from being a member of the International ISO-JTC1/SC22-WG9 on Ada, a member of the CEC VDM-Europe Working Group and a member of IFIP-TC2, WG 4 (WG2.4 on systems implementation languages), he is also the Dutch representative within IFIP TC2.