

# Modeling the Static Software Architecture in AADL

Peter Feiler  
[phf@sei.cmu.edu](mailto:phf@sei.cmu.edu)  
July 15, 2009

## Abstract

*Abstraction and encapsulation are two principles that have been introduced to manage the complexity of software systems through modular design. They have manifested themselves in the concept of classes and methods to support data abstraction, in the concept of package to support organization of the design space, and the concept of component to support structuring the system architecture. AADL has been developed as architecture modeling language for embedded software systems. As such it focuses on modeling the runtime architecture, i.e., the dynamics of the operational system and its non-functional quality attributes, such as performance and safety. In this note we discuss how AADL supports modular design of software in a hierarchical fashion, its organization into libraries, and constraints on the architecture structure.*

## 1. Introduction

A challenge for designing large scale software systems is to apply the principles of abstraction and encapsulation in order to reduce complexity. Data abstraction has led to abstract data types [Dahl 1967, Wexelblat 1981] and later to classes with methods, whose instances are objects [Gosling 1996]. Similarly, functional abstraction was initially addressed through library of functions, virtual machine layering [Dijkstra 1963], modular encapsulation [Parnas 1972], and the integration of these two concepts [Habermann 1976]. This has led to the package concept in many programming languages and modelling notations as a way to organize software artifacts into a hierarchy of manageable units with visibility rules and constraints on the use of their content. In the 1990s the concepts of components and connections have emerged as abstraction for modelling software architectures [Shaw 1996]. In this note we will examine each of these abstractions and discuss how they are reflected in an AADL [AADL 2009] model of your embedded software system architecture.

Packages and components have been introduced with different objectives in mind. The objective of the package concept is to manage and improve modifiability of a software system. It does so by providing a tool for organizing software design artifacts into a package hierarchy with separation of public interfaces and private implementation details, to record use dependencies between the packages, and to place restrictions on packages in the use of design artifacts from other packages. The objective of the component concept is to manage the complexity of a system architecture and its operational quality attributes. It does so by representing a system architecture as a hierarchy of components with interactions between the components limited to the interaction points specified in the interface of each component in the hierarchy. To address the operational quality attributes of an embedded software system, such an architecture must consist of the runtime components of the application software, the computer system it executes on, and the physical system/environment it interacts with.

## 2. Modular Design with Packages

In this section we first examine how programming languages support modular design through the package concept. Then we discuss how AADL allows us to reflect such an organization of software artifacts in an architecture model.

## 2.1. Modular Design in Programming Languages

A modular design is based on a package (module) that defines interfaces to a set of software design artifacts - typically data types and procedures or equivalently classes and methods, and sometimes application constants - making some publically available for use by other packages, while keeping others private to the package. In the case of procedure libraries the function signatures are made available as the library interface, while their implementation and any support functions are private. In the case of data types and classes, instances can only be manipulated through accessible fields or access methods, while the internal representation of data is private. The interface specification of such software artifacts are placed in the public part of packages, while their implementations are placed in the private part of packages. Only those artifacts exposed through the public part of a package can be referenced by artifacts in other packages, e.g., only public functions in a library can be called. For programming languages like C this is emulated by a header file (.h) for a set of functions and data types (*struct*) as the public interface, and a C source file (.c) for the implementation. Other programming & modeling languages such as Java and UML provide a package construct as part of the language syntax. Packages have public and private sections, or artifacts kept in packages are tagged as public or private. Intended use dependencies on other packages are indicated through a *with* statement.

Packages typically introduce name scopes for the artifacts contained in them. The artifacts are uniquely identified by the package name and the artifact name. Packages can be organized into hierarchies to manage the design space. Such name hierarchy may reflect a combination of organizational units and functional units of a system, e.g., *org.eclipse.jface.util.OpenStrategy*. Some languages allow users to place in a naming hierarchy (Java) without imposing restrictions on dependencies between packages due to the hierarchy. In this case, other administrative mechanisms such management oversight and access control may limit the authorized use of other packages. Ada supports nesting of package declarations, when placed in a private section limiting their visibility to that package and child packages.

Languages typically do not support instantiation of packages directly, but consider that to be part of the software build process. Ada offers *generic* packages, i.e., packages that are parameterizable. They use the term instantiation in this context to refer to the creation of a package that is derived from the generic package by supplying a parameter.

## 2.2. Modular Design in AADL

AADL has followed the lead of programming languages and provides a package concept to organize modeling artifacts, in the case of AADL component specifications. AADL packages have public and private sections, each of which can be placed in a separate package declaration - stored separately in the file system or a development repository.

AADL packages contain component interface specifications (component types) and blue prints of their implementations (component implementations). A component type can have multiple component implementations. A component implementation declaration can be separated into a public variant specification, i.e., a refinement of its interface specification in terms of variant-specific properties without implementation details, and a declaration of its full implementation details which may remain in the private package section.

AADL supports modeling of data types and classes through the *data* component. *Data* component types may include properties regarding an instance of the data type. The interface feature *provides subprogram access* of a data component type can represent methods on classes. The internals of the data representation are kept in a separate data component implementation declaration. The focus of AADL is on architecture modeling. Therefore an AADL model may reflect only those data types in packages that are relevant for modeling persistent and shared data components as well as the type of data being exchanged between architectural components without requiring their internal representation. AADL can be used as a data modeling language by representing the implementation details of every data type.

AADL supports modeling of procedures through the *subprogram* component. Subprogram types can be declared with or within interface signature, and the internals of a procedure can be reflected in a subprogram implementation declaration. The subprogram type is referenced by a subprogram call when modeling call sequences inside a thread or a subprogram implementation, and by the subprogram access feature of the data type when modeling methods of classes. In an architecture model we may only model those subprograms that are relevant in the interaction between

architecture components. Detailed design such as call sequences within thread and other behavior specifications are supported by the core language and Annex extensions for behavior modeling. The detailed software design expressed in a detailed design notation is associated with its AADL abstraction via the *Source\_Text* property.

AADL supports modeling of procedure libraries through the *subprogram group* component. The features of the subprogram group type represent the externally callable procedures in the library. The subprogram group type also indicates any dependencies of this library on other procedures and libraries. Procedures local to the library are hidden from external users.

AADL packages have *with* statements to indicate intended use of other packages. A separate *with* statement can be associated with the public and the private section. The *with* statement places a restriction on the use of other packages, i.e., for component specifications in a package to only reference component types and implementations (classifiers) in those packages.

AADL supports organization of packages into a naming hierarchy with each package representing a separate name space. This supports independent development of packages without concern for name conflicts. By default AADL does not impose restrictions on use dependencies between packages based on the naming hierarchy. AADL does not prohibit projects to impose constraints, such as the ability to only reference classifiers in packages at the same or lower levels of the package hierarchy, if desired.

Other forms of visibility restrictions on packages can be achieved by complementing the basic language support – as is done for source code in programming languages. We can utilize the access control mechanisms of a model repository, or by introducing a property that represents an access control list. We can also introduce a property of type *reference*, whose value is a list of references to other packages. This allows us to explicitly specify packages that are allowed to make use of a given package. Finally, if we want to impose a layer restriction on package *with* statements, we can introduce a property that reflects a level or tier and require that the *with* statement only refers to packages at the same or lower tiers. For more details on reflecting different dimensions of architecture layers in AADL see [Feiler 2009b].

AADL packages and the package naming hierarchy are used to organize component specifications along multiple dimensions. Packages are used to organize the software artifacts, i.e., data types and procedures. For example, set of packages that contain AADL data types for application data type can be organized under a package name *data* dictionary. Subprograms that represent the methods of a class may be grouped with the data component type representing the class. Packages are used to organize the component specifications of the architecture, grouping them into hardware, software, and physical system. In each case, the grouping may reflect libraries of lower level components separate from their composition into higher-level subsystems. Higher levels of the package hierarchy may reflect organizational units, such as different development teams, major subsystems of a system, a family of systems derived from a common reference architecture, variations of a system in terms of different software and hardware configurations, or different deployment configurations [Feiler 2007] [Feiler 2009a].

### 3. Architectural Design with Components

In this section we first examine how modeling languages support component-based design through the component concept. Then we discuss how AADL allows us to represent the architecture of an embedded software system, i.e., the task and communication architecture of the application utilizing its software artifacts, the computer system architecture, runtime infrastructure and the deployment of the software on the hardware, the elements of the physical system and its logical interface with the application and the physical interface with the computer hardware.

#### 3.1. Component-based Design in Modeling Languages

Component-based design is based on the concept of component that defines an interface through which it interacts with other components, and an implementation that specifies how a component may be composed of other components, which themselves may be compositional components.

Component-based modeling languages usually follow the type/instance paradigm, i.e., a component type specification that defines the external interface and properties, and a component implementation as blue print for how such a component is instantiated in terms of other components. The resulting hierarchical instance model represents an instance of an actual system.

Component interactions are expressed by connections between the interaction points in the interface, typically ports. Connections follow the component hierarchy, i.e., components in different parts of the component instance hierarchy only interact if permitted by the interface of their common ancestors in the hierarchy. For example, two threads in different processes may only communicate if the processes provide an interface through which the cross-process communication is supported.

Component-based modeling is used for modeling physical systems (SysML), for high-level and detailed design of control systems (Simulink), for high-level and detailed design of computer systems (VHDL), for software architectures (AADL, UML components) and for detailed design (UML/Java classes, flow charts). In the case of software, modeling languages distinguish between passive and active objects in a first step towards a dynamic architecture, i.e., a task and communication architecture that is mapped onto a computer hardware architecture and interacts with a physical system and external environment.

### 3.2. Components in AADL

AADL supports component-based modeling of embedded software system architectures. It does so through the concepts of component types with interface features and properties, and multiple component implementations for the same component type with subcomponents and connections.

AADL provides specific component categories for computer hardware architectures and physical system components (processor, memory, bus, device) – an abstraction originally used by Bell & Newell as PMS [Bell 1971], for the runtime system (virtual processor, virtual bus), for the software task architecture (process, thread group, thread), for the application software (data, subprogram, subprogram group), and for general compositional components (system, abstract).

Threads are active software components that execute on processors, while data components are passive persistent software components that are accessed by threads. Devices are active physical components with a logical interface to interact with threads. Processes represent protected address spaces. Thread groups represent generic compositional components within a process that can contain threads, thread groups, subprograms, and data. Systems play the same role for the higher levels of a system component hierarchy; they can contain hardware components as well as software components in the form of processes, data, and subprograms. The abstract component represents a generic component that can later get refined into one of the concrete categories through *extends* refinement.

Interaction between active components falls into three categories: port-based directional transfer of data via port connections, shared access to persistent data component instances via data access connections, and procedural service invocation – both local and remote – via subprogram access connections or call bindings. They are supported on the hardware side by bus access connections to bus instances that provide physical interconnections between hardware component instances.

AADL supports multiple modeling approaches to representing embedded software systems. Software artifacts in the form of data types and procedures are represented as data subprogram component types in “application software” packages. The application software task architecture is represented as threads, persistent data, protected address spaces of processes, and compositions of them. Application software is mapped into this task & communication architecture as source code through source code properties pointing to source code files and program units within them. If desired the call dependency between procedures in the source code can be reflected in subprogram component specifications. If desired code instances of subprograms and subprogram libraries (subprogram groups) can be modeled explicitly. If desired, component representing software, such as subprograms, subprogram groups, threads, can have interface specifications that include call dependencies in the form of *requires subprogram access* and *requires subprogram group access*.

On the computer system side, hardware components can be grouped into system components of different types that represent composite components such as boards with processor and memory, or cabinets with multiple boards and network connections. The physical system, which is the context of an embedded software system is represented by devices and buses of different types that composed into a physical system hierarchy and interface with the computer system via buses.

Conceptual architectures are often represented in the form of a data model (e.g., class diagrams) and a component model with domain-specific component types (e.g., UML components & stereotypes). AADL supports such architecture models through data component types with a refinement type hierarchy with services as subprogram features, and through the *abstract* component category used to define application domain specific component types.

AADL support partial component specifications, i.e., component type and implementation patterns parameterizable with data types and component classifiers. This supports specification of architecture patterns, such as redundancy patterns, and reference architectures that can be refined into application-specific instances through *extends* refinement. The *extends* refinement mechanism also allows us to refine a model of abstract components into specific software and hardware component categories.

AADL supports strong typing of its component models. Component categories have specific semantics, e.g., only threads can be bound to processors for execution. Component types are treated as types and are used to type component interface features. For example, a processor of type PowerPC supports connectivity to devices such as a camera via a bus of type USB. Similarly, ports are typed and the data types of connected ports must satisfy type matching rules.

Layered architectures can be represented in AADL in two ways: layering of the component hierarchy, and layered interaction between active components. Layering of the component hierarchy means that components are organized into different tiers and components of one tier can only contain components of a lower tier. We can reflect such component tier layering by organizing component specifications of different tiers into different sets of packages, either utilizing the package name hierarchy to reflect the tier, or by tagging each package with the appropriate tier – and then restricting the *with* statements to adhere to the tier hierarchy restriction. We can also associate a tier property with each component specification if the package organization is not appropriate and constrain the subcomponent classifier tiers with respect to the containing component implementation tier.

System architectures may have what is sometimes called horizontal layering, i.e., restrictions on the interaction between active components. For example, in e-business an application system may consist of front-end user interface tasks through a web interface, a middle layer of service execution, and a back-end set of data base services. Tasks of one layer can only directly interact with tasks of the next layer (sometimes also tasks at the same layer). Such restrictions can be addressed by tagging active components (threads and devices) with a *layer* property and enforcing interaction connections to adhere to the layer restrictions. For more discussion on modeling architecture layers in AADL see [Feiler 2009b].

## 4. Mapping Application Source Code Concepts into AADL

Several programming language concepts do not always cleanly fit into the component framework: source code libraries, class & method instantiation, static variables and component instances, application constants, data type visibility & architectural component hierarchy. We will discuss each in turn.

### 4.1. Application Source Libraries

Application source libraries are collections of procedure specifications and packages of class and method declarations representing software artifacts that are used in the specification of other software artifacts. When used in the construction of a software system, they are turned into instances and grouped into executable libraries, e.g., Windows *dll* files or Java *jar* files. Typically, programming and modeling languages support source libraries in the form of packages and leave the instantiation of such libraries into executables to tools.

AADL supports modeling of package-based source code libraries as collections of subprogram types and the representation of a collection of subprograms as a subprogram group with subprogram features making up the callable subprograms. Subprogram groups cannot not be nested in subprogram groups the same way that *dll* files and *jar* files cannot be nested inside *dlls* or *jars*.

These packages are then combined with component libraries of the software task architecture. The modeler specifies thread components and associates subprograms to be executed as part of a task as thread entry points or call sequences by referencing the subprogram type or subprogram feature of a subprogram group. The use of a subprogram group instead of just a collection of subprogram types in package is useful if underlying application source language has an explicit concept of library separate from the package concept, e.g., Ada, if it is desirable to explicitly model the use dependency at the component rather than the package level, or if it is desirable to explicitly represent application source and library instances rather than build tools determining the number of instances.

If the modeler pursues a component-based modeling approach, they may explicitly specify the fact that they require a specific subprogram by making use of *requires subprogram access* and *requires subprogram group access* declarations as part of the component interface specification. For example, a subprogram type may indicate that it requires the use of a library in its implementation.

AADL allows you to go one step further and explicitly represent source code instances in the form of subprogram subcomponents and subprogram group subcomponents. They can be declared as instances at the thread group, process, or system level and shared across threads or processes. As such they can be explicitly allocated to specific memory locations in hardware. Calls can be made to specific subprogram instances – expressed by subprogram access connections from the call to the respective instance. Such models may reflect the result of a compilation and build process by the tool environment and may be generated rather than hand-constructed by the modeler.

## 4.2. Class and Method Instantiation

In object-oriented languages instantiation of a class into an object does not imply an instance of the method as well. In the language the user explicitly expresses when a class is instantiated, either statically through an instance variable, or dynamically through a *new* constructor. The methods of a class are treated like a subprogram library and their instance creation is handled by the language tool environment.

In AADL classes are modeled as data component types and methods are declared as provided subprogram features that refer to subprogram types to identify the method signature. Data component instances are declared in terms of data component types and do not imply instantiation of associated subprograms. As discussed in section 4.1 subprogram instantiation can be modeled independently by declaring subprogram subcomponents or subprogram group subcomponents that contain the subprogram as feature.

## 4.3. Static Variables & Data Component Instances

Programming languages support the declaration of static variables, i.e., data that persists throughout the life of an application. In languages like C they are declared as part of the source file and are assumed to be instantiated once per program instance, typically a single or multi-threaded application process. They are accessible within the source file and by other source files if declared external.

In languages like Java such static variables can be declared together with instance variables (fields of a record). They are considered to be instantiated once per program rather than every time an instance object of the class is created. They represent state that is shared between all object instances and is accessible by the methods. If they are declared as public they may be accessible to methods in other classes as well. The need for synchronization is declared with the class, i.e., applies to all object instances. Sharing of such data is usually not explicitly declared part of a thread interface specification in programming languages such as Java and has to be inferred from subprogram call dependencies.

In AADL static variables are represented by data component subcomponents. They are persistent unless they are declared as a subprogram subcomponent. Access to them from outside a component is provided explicitly through *provides* and *requires data access* features. Such persistent data components can be shared across threads in the same thread group or process, across thread group or processes, and even across systems. Since AADL subprograms are stateless subprograms that access persistent state have to do so through *requires data access* features. For example, two thread instances may update a persistent state variable by sharing access through data access connections to the data component instance. This data component instance may be declared in the enclosing process and tagged with requiring a concurrency control protocol by property. Each thread then makes this shared data component accessible to the subprogram performing the update through a *requires data access* feature in the subprogram interface, i.e., as reference parameter.

In summary, in AADL shared access to persistent data is managed according to the component interface specifications and access connections in the component instance hierarchy. This reflects the potential need for concurrency control on such shared data. In contrast, in programming languages access to persistent data is limited by the visibility rules for application source artifacts and access by multiple threads has to be inferred from subprogram call dependencies.

## 4.4. Application Constants & Model Parameters

Programming languages permit the declaration of application constants. Such constant declarations can take different forms. In C we can define a symbolic constant label for a numeric or other value (*#define*). This symbolic label can then be used anywhere the value is syntactically acceptable. For example, the constant label can be used to

specify the size of an array, the initial value of a variable, the upper bound in a for loop that creates a number of threads, or a calibration parameter or other value in a computation. In the generated code the constant may take up space to hold the read-only value, may affect the amount of memory allocated to an array, or may determine the number of threads being created. The symbolic labels are also used to configure source code through conditional compilation.

In Java application constants can be specified by declaring *static final* variables in interfaces or classes and making them publically available. These variables can then be referenced anywhere a constant value of the appropriate type is acceptable. Configuration of source code by conditional compilation is achieved through code optimization by the compiler recognizing a constant value as the condition of an *if* statement.

When representing application constants in AADL model we need to consider their role and their effect on the architecture. Some application constants are read-only data values that are part of the application. They can either be simply treated to be part of the application code generated from the application source and accounted for in terms of memory requirements through the *Source\_Code\_Size* and *Source\_Data\_Size* properties. If the application constant has architectural significance, then it can be declared as a persistent data component with read-only access rights. The value itself can be recorded through the *Initial\_Value* property of the Data Modeling Annex. If a constant is used to determine the initial value of a data component, then a property constant can be referenced as the value of the *Initial\_Value* property.

Some constants are parameters to the model itself. One example is constants that represent variant selection through conditional compilation. Such constants can be represented as property constants if they apply globally and are passed on to the build process as parameters. If conditional compilation is controlled separately for each package, then the compilation flag values can be represented by properties associated with each package. Another example is the use of property constants as values for the size of component arrays such as thread arrays. In the case of data arrays, we can either parameterize the size data component arrays by property, or we can follow the Data Modeling Annex guidance for representing data structures such as arrays and parameterize their size via properties. For further discussion of representing application and model variation by properties and classifier parameterization see [Feiler 2007].

#### **4.5. Data Type Visibility & Component Hierarchy**

The application is represented as a task architecture with threads contained in thread groups, processes, and systems. Threads communicate data and events through ports and other features that are typed with data types. It is tempting to impose the component hierarchy as a structure to control visibility of data types by requiring the modeler to make the data types accessible up the component hierarchy through component interfaces. However, data types are normally not communicated between application tasks, therefore are not a public feature of the component interface, but an attribute of a feature such as a port. In some cases, data types are encoded as Meta data and sent with the data. That fact is reflected in the data type of the port in that it consists of content data and Meta data.

Let us illustrate why it is important to allow data types and subprograms to be organized in a structure separate from the component hierarchy. A system may have two tasks in different subsystems or partitions that communicate. In addition, the sender task may be implemented as a redundant task, whose output is monitored by a health monitoring task. The sender and the receiver tasks are the only tasks that operate on data of this data type. The data type and the procedures operating on it are defined in a single package. Since this data is communicated from a thread inside one process to a thread inside another process, the modeler specifies for each common ancestor component of the communicating threads ports that reference the data component type. If we were to have the data type be part of a component that is made accessible to other components it would be placed high in the component hierarchy, thus, potentially accessible to a number of tasks other than the two intended tasks. By placing the data type and its procedures in a single package we indicate that in terms of software artifacts this is the only package that is affected by changes to the data type. In terms of the software architecture port communication is typed and follows the component hierarchy to a common ancestor of two communicating active components. Only the active components operate on the data, thus, they are the only ones affected by a data type change. In our example, the active components would be the two instances of the sender task, the health monitor task, and the receiver task. The use of the data type up and down the hierarchy allows us to incrementally perform type checking of port connections for matching types. Thus, only if we change the type reference of a port will it potentially introduce a type mismatch.

## 5. Summary

In this note we have made the case that both the package concept and the component concept play an important role in the modelling of the static architecture of a system, packages focusing on structuring the design space, while components reflect the structure of the system architecture. A comparison table is provided to identify the appropriate use of packages and components.

	Package	Component
Role	Manage design space	Manage system architecture
Modular abstraction	SW spec: Data type, procedure	Component spec: Interface, implementation w. subcomponents & connections SW, HW, system categories w. semantics
Encapsulation	Public & private parts Package hierarchies	Component interface & implementation Component hierarchies
Hierarchical organization	SW specs, component specs, configuration specs	Composition of system component instances
Organizational principle	Multiple: teams, subsystems, configurations, HW, SW, System, data dictionary	System component hierarchy Management of component interactions
Public interfaces	Data types, procedure signature	Data transfer, data access, service call
Dependencies	Use dependencies for packages	Interaction dependencies between components
Dependency restriction	Public/private visibility Optional: hierarchical visibility	Interface & component hierarchy Optional: active instance layers

## 6. References

[AADL 2009] SAE International. “Architecture Analysis & Design Language (AADL)”, SAE Standards: AS5506A, Nov 2004, <http://www.sae.org/technical/standards/AS5506A>. Revised Jan 2009.

[Bell 1971] G. Bell, A. Newell, “Computer Structures: Readings and Examples”, McGraw-Hill, Computer Science Series, 1971.

[Dahl 1967] Ole-Johan Dahl and Kristen Nygaard, “Class and subclass declarations” Proceedings of the IFIP working conference on simulation programming languages, Oslo, May 1967, <http://en.wikipedia.org/wiki/Simula>.

[Dijkstra 1968] E. W. Dijkstra, *The structure of the 'THE'-multiprogramming system*, *Communications of the ACM* **11(5)**:341 – 346, 1968. [http://en.wikipedia.org/wiki/THE\\_multiprogramming\\_system](http://en.wikipedia.org/wiki/THE_multiprogramming_system).

[Feiler 2007] P. Feiler, “Modeling of System Families”, Technical Report CMU/SEI-2007-TN-047, July 2007.

[Feiler 2009a] Feiler P., Gluch D., Weiss K., Woodham K., “Model-Based Software Quality Assurance with the Architecture Analysis and Design Language”, *Proceedings of AIAA Infotech @Aerospace 2009* (April 2009).

[Feiler 2009b] P. Feiler, J. Delange, “Modeling of Layers in Embedded System Architectures”, Technical Report CMU/SEI-2009-TN-0XX, Aug 2009.

[Gosling 1996] James Gosling, Bill Joy, Guy L. Steele Jr., *The Java Language Specification*, Addison Wesley Publishing Company, 1996.

[Habermann 1976] A. Nico Habermann, Lawrence Flon, Lee W. Cooper: Modularization and Hierarchy in a Family of Operating Systems. *Commun. ACM* **19(5)**: 266-272 (1976).

[Parnas 1972] Parnas, D. L. “On the Criteria to be Used in Decomposing Systems into Modules”, *Comm. ACM* **15**, 12, Dec, 1972.

[Shaw 1996] M. Shaw, D. Garlan, “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall, 1996.

[Wexelblat 1981] Ole-Johan Dahl and Kristen Nygaard, "How Object-Oriented Programming Started", in "A History of Programming Languages", Ed. R. Wexelblat, ACM Academic Press, June 1981.