ADAM: A Language-Independent, Object-Oriented, Design Environment for Modeling Inheritance and Relationship Variants in Ada 95, C++, and Eiffel*

D. Needham, S. Demurjian, K. El Guemhioui,
T. Peters, P. Zamani, M. McMahon
Computer Science & Engrg. Dept.
The University of Connecticut
Storrs, Connecticut 06269-3155
Telephone: (860)486-4818, Fax: (860)486-4817
{needham,steve,karim,tpeters,mac}@eng2.uconn.edu

H. Ellis
The Hartford Graduate Center
School of Engrg. and Sci.
275 Windsor St.
Hartford, Connecticut 06120
Telephone: (860)548-2497
heidic@hgc.edu

Abstract

The ADAM environment provides a language-independent, object-oriented platform from which Ada 95, C++, and/or Eiffel code can be automatically generated from a single design. ADAM provides design capabilities that support variants of inheritance and relationships, from different perspectives. A software engineer designing a database application has different needs than one concerned with a graphical user interface, and the variants are intended to allow each need to be met. The main purpose is to offer software engineers alternatives during design, thereby allowing behavior to more precisely match an application's requirements. By providing software engineers the ability to specifically choose and customize behavior for an application, from which the environment then automatically generates code, a more disciplined approach to design and development is promoted. Moreover, all of the inheritance and relationship variants generate code in a controlled and consistent fashion, resulting in predictable application behavior and improved readability of generated code, and hopefully leading toward a more understandable and maintainable software product. This paper examines ADAM and its inheritance/relationship variants, in a common design platform from which code in multiple object-oriented languages can be generated. We have chosen to also present C++ and Eiffel since they offer an interesting contrast to the capabilities supported by Ada 95, and they demonstrate the language-independence of the ADAM environment.

*T. Peters and D. Needham acknowledge, with appreciation, partial funding for this work received under National Science Foundation Grant Number MII-9308346. The views expressed herein are of the authors, not of the National Science Foundation.
1 Introduction

As part of our work in the exploration of issues related to information consistency and the attainment of engineering rigor, we have developed the prototype object-oriented environment ADAM (short for Active Design and Analyses Modeling) [18]. ADAM supports an object-oriented design model that is tightly integrated into the environment, with the semantic scope, content, and context of each modeling construct clearly defined. Consequently, there is no “programming language” for ADAM; design choices are made via menus, browsers, etc., and text is directly entered by the software engineer using forms. The environment stresses language independence by focusing on design and allowing source code to be generated in a variety of target languages, including: Ada 83 [13], Ada 95 [1, 14], GNU C++ [17, 40], Ontos C++ [32] and Eiffel [27, 28, 35]. It is important to note that source code for any (or all) of these languages can be generated from the identical ADAM design representation. ADAM supports both incremental and iterative design by allowing design data to be stored persistently into the Ontos object-oriented database system [32]. ADAM can generate baseline design documentation in either ASCII or LATEX, and has recently incorporated an ICAD code generator to support object-oriented design for mechanical parts [31]. In addition, ADAM considers security as an integral part of the design and development process via extensive user-role based security capabilities [9, 11, 12, 23, 24, 25].

When attempting to structure and create an environment for designing object-oriented applications that is language independent, one of the key concerns is to recognize and quantify design capabilities. Object-oriented design means different things to different individuals; software engineers, database engineers, programming language experts, and so on, all have not only complementary but often conflicting perspectives. These perspectives are further complicated when one attempts to separate design from programming. Intersecting across multiple object-oriented programming languages is likely to yield a core set of common capabilities, while their union results in a wide-range of varied constructs and features. Trying to step back from diverse programming-language issues while still integrating perspectives from different individuals is by no means an easy task. However, in the ADAM environment and its object-oriented design model, our goal has been just that - to unify multiple perspectives from different individuals into a design platform that can meet their application modeling needs independent of the target object-oriented implementation language. Our overriding intent has been to add more discipline and engineering to the software design and development process through a controlled environment that offers modeling choices with specific and defined behavior. These modeling choices are then carried forward into code that is generated in a consistent, predictable, readable, and hopefully
maintainable fashion.

To meet our goal, this paper introduces and examines inheritance and relationship variants for object-oriented design. These variants quantify design from alternate perspectives in order to meet the diverse needs of different individuals when modeling their applications. Inheritance variants distinguish between behavior at a type level and at an instance level. At a type level, software engineers are concerned with the information that is available in the subtype from the supertype. At an instance level, the variants support different criteria for instantiation, i.e., under what conditions can an object be instantiated. When type-level and instance-level considerations are combined in an inheritance hierarchy, there is an impact on the code that is automatically generated by the ADAM environment. It is important to note even at this stage, that not all object-oriented languages can support all of the inheritance variants available at the design level in ADAM. Like inheritance variants, relationship variants are intended to offer software engineers differing capabilities based on their needs. Specific relationship semantics are realized using generics during automatic code generation, promoting reuse, reducing potential inconsistencies, and improving the overall quality of the software product.

This paper focuses the capabilities of the ADAM environment for supporting object-oriented design, inheritance and relationship variants, and automatic code generation in a variety of object-oriented languages. In previous work we have reported on the educational uses of ADAM and Ada 83 [15]. In another effort [16], the object-oriented design modeling capabilities of ADAM were examined, illustrating both Ada 83 and Ada 9X code generation. While inheritance and relationship variants were covered in this effort, their support in Ada 9X was not fully understood; a number of the inheritance variants were still under research at that time. The release of Ada 95 and our subsequent revision of the code generator for ADAM has resulted in a more comprehensive understanding of the inheritance variants and their support in Ada 95. Further, this paper, unlike our prior two efforts [15, 16], also presents the support of variants in C++ and Eiffel. We believe a discussion of all three languages allows the reader to understand the strengths and weaknesses of each, contrasting their capabilities in supporting different variants of inheritance. Such a discussion is critical as we attempt to understand the different object-oriented programming languages in order to assess their ability to support the key software engineering principles and processes.

The remainder of this paper contains six sections. Section 2 reviews the ADAM environment and its design model by examining the profiling concept. A sample software development application, the Automated Automobile Factory of the Future (A²F²) is introduced for the purpose of demonstrating key features of the ADAM environment. Section 3 examines the ADAM inher-
itance variants which offers a software engineer the means to explicitly and finely control access to object behavior. Section 4 presents relationship modeling in ADAM, with a special emphasis on the Group construct's use of generics/templates. Both Sections 3 and 4 provide illustrative examples of automatically generated Ada 95, C++, and Eiffel source code. Section 5 reviews related work that is relevant with regards to the ADAM environment and its object-oriented design model. Finally, Section 6 offers concluding remarks on our experiences in developing a language-independent, object-oriented design environment, and offers a brief review of our other on-going research efforts.

2 Object-Oriented Design Model And Profiles

The ADAM environment provides the software engineer with multiple design phases for the construction of the different aspects of an application, segmenting the design process into discrete steps. These steps guide the software engineer by suppressing details not directly related to a particular design phase, thereby allowing the software engineer to focus more closely on a particular design component. For instance, the first phase allows only object type creation and inheritance hierarchy construction; relationships and group components may not be specified during this phase. Separate design phases also support modularity in the design process, since each design phase provides a localized view of one aspect of an application. Collectively, this facilitates the construction of designs in an organized fashion.

Concurrent with each design phase and with the construction of each design component is the profiling activity. A profile contains detailed requirements on the semantic content and context for all of the design constructs of the application. Profiles are used to require software engineers to supply detailed information as the application is designed. In addition, profiles track data that provides on-demand and automatic feedback to software engineers whenever an action in the environment results in a conflict or possible inconsistency. Therefore, profiles constitute an important means to instill engineering rigor in the design and development process. The remainder of this section first provides background information on the concepts and assumptions of the object-oriented design model. Then, profiles and their realization in the ADAM environment are discussed in detail as a basis for the material in Sections 3 and 4.

2.1 Concepts and Assumptions

In the object-oriented design model supported by ADAM, the main modeling component is an object type (OT) for defining similar state and behavior. By its definition, an OT encapsulates
both information via a set of private data, denoted as $D$, and behavior via a set of public and private methods, denoted as $M$. In the object-oriented design model supported by ADAM, only private data is allowed, with methods classified as both public and private. The existence of private data for an OT and the differentiation between private and public methods serves as the basis for supporting information hiding concepts. In information hiding, the public methods are intended to provide a uniform interface for maintaining consistency and reusing software, while the private methods protect an implementation that can support design changes and enhancements that do not affect the public interface. From a conceptual perspective, private in ADAM is similar to C++’s protected, while public is similar to C++’s public. By not permitting the presence of public data within a type, we are taking a very strict definition of the characteristics of an OT from a design perspective. Private data that needs to be accessible can be achieved by defining public methods that read, write, and modify the data in question.

Finally, note that in our efforts we have used the term potential public interface. This is since, in ADAM, all of the public methods are, by default, inaccessible, retaining the potential to be made public via the security aspects of the object-oriented design model. In practice, the public interface contains the union of all of the methods that are needed by all of the potential users of a OT. Hence, there are some methods that are public which the software engineer intends for use by only a limited number of individuals. But, once a method has been made public, regardless of the intent of limited use, the method is available to all. Security must be available to customize the public interface, making it appear differently to different users at different times based on their needs within the application. Thus, the term potential indicates that these methods may be made public to different users, based on some discretionary access protocols. This means that some of these methods may never be made public, while others may be given to only a limited number of users. For the purposes of this paper, we will only deal with public and private methods, and private data. The interested reader is referred to a large body of our efforts in user-role based security for object-oriented models that has appeared elsewhere in the literature [8, 9, 10, 11, 12, 23, 24, 25].

2.2 Profiling in the ADAM Environment

For the purposes of illustrating our concepts in ADAM, we will use the Automated Automobile Factory of the Future ($A^2F^2$) application to define object types that contain attributes and methods, to model inheritance hierarchies, and to detail the relationships between object types. For a more complete description of $A^2F^2$ and its use in a software engineering course, please see [30]. Figure 1 shows the inheritance hierarchy for Part as it would be displayed in the OT-specification
design phase of ADAM. In this figure, there are seven OTs: Part is the root of a hierarchy, with BodyPart and MotorPart as children of Part, and ElectricalPart and MechanicalPart as children of MotorPart, and Accessory as the child of BodyPart. The Supplier OT has no descendants, and primarily exists in this representation as a vehicle for the demonstration of relationships and groups.

Figure 1: Sample Object Types for the $A^2P^2$ Application.

The distinction between BodyPart and MotorPart is important for identifying suppliers and reordering parts from various vendors, while the ElectricalPart and MechanicalPart distinctions are necessary for servicing/repair actions and warranty tracking. The Accessory OT exists to allow the modeling of pricing information that differs depending on whether a body part is directly produced by the automobile company, or the body part is purchased at retail by the automobile company as an outsource.

To define the different components in the object-oriented design model, profiles are utilized to represent and capture relevant information. In compliance with the object-oriented philosophy, the ADAM profiles define an inheritance hierarchy. At the root of the hierarchy is the base profile, BP, which contains a name for the feature or construct and a prose description for its purpose within the application. A BP represents the minimal information that is provided for a design component (feature or construct) in an application. Using BP, profiles are definable for attributes,
methods, object types, relationships, and groups. Each of this will be examined in turn.

At the lowest level of granularity, an attribute profile, as shown in Figure 2, is defined as:

**Definition 1:** Each attribute $A_i$ in an object type (OT) is defined using an attribute profile, $AP$, which is a specialized BP that contains:

1. a prose attribute description (from BP) for the purpose of the attribute in the OT,
2. the name (from BP) and type of the attribute, and
3. a list of the methods that access (read and/or write) the attribute.

The first two items must be supplied by the designer, while the last one is generated from other profiles. Note also that for each attribute, the software engineer can specify that Set, Get and/or Print methods be automatically created, in either the public or private portions of an OT, as shown in Figure 2.

A method profile, as given in Figure 2, has information on the use and effects of each method defined on an OT:

**Definition 2:** Each method $M_i$ in an OT is defined using a method profile, $MP$, which is a specialized BP that contains:

1. a prose method description (from BP) for the method's actions within the OT,
2. the method's name (from BP), its return type, and parameters (i.e., signature),
3. the read/write set for each private attribute of the OT used by the method $M_i$, and
4. the other methods that are called by $M_i$ to accomplish its task.

All of the MP information is supplied by the designer when defining methods.

The object-type profile, presented in Figure 2, provides a higher-level perspective on an application design:

**Definition 3:** Each OT in an application is defined using an object-type profile, $OTP$, which is a specialized BP that contains:

1. a prose OT description for the purpose of the OT in the application (from BP),
2. the OT's name (from BP),
3. the persistency status of the OT,
4. the APs for all private attributes,
5. the MPs for all methods,
6. the supertypes/inheritance variants of the OT, and
7. the subtypes/inheritance variants of the OT.

Note that the inheritance variants as indicated by Definition 3.6 and 3.7, are examined in Section 3.
Figure 2: Sample Object, Attribute, and Method Profiles for the $A^2F^2$ Application.

One of the most prominent features of modern, complex applications is the high degree of interdependencies that exist among its different components. During the ADAM RT-specification phase, the software engineer defines the structure of all non-inheritance interdependencies between different components of an application via a relationship-type ($RT$) construct. Profiles for $RT$s provide information about the purpose and actions of a relationship:

**Definition 4:** Each $RT$, in an application has a relationship-type profile, $RTP$, which is a specialized OTP that contains:

1. all information from an OTP,
2. the relationship variant, and
3. the involved source and destination OTs.

Like OTPs, RTPs are utilized to provide the software engineer with feedback on the application design. Note also that ADAM supports profiles that track the definition of groups (e.g., set, collection, queue, etc.). Figure 3 contains examples of RT and group profiles for the $A^2F^2$ application in Figure 4.

Figure 3: Sample RT and Group Profiles for the $A^2F^2$ Application.
Profiles have an important role within ADAM, since they promote the understandability of a design by forcing the software engineer to supply detailed information for each and every application component. They add rigor and formality to the design process, thereby providing a vehicle for the software engineer to verify the application requirements. The profiles are utilized by a battery of analysis techniques to assist in the design process and alert the software engineer of conflicts or possible inconsistencies resulting from design decisions, modifications, and/or extensions. Some examples of analyses that are supported by ADAM include:

- Aggregating MP descriptions for an OT: This would aggregate the MP descriptions for an OT and all of the methods that are inherited from any ancestor(s). A software engineer would utilize this analysis to insure that the combined public interface for an OT represents his/her intent.

- Tracing upward and downward inheritance hierarchies with respect to attribute names, method names, etc. This would assist the software designer in preventing denomination clashes.

- Identifying RT Cycles: Since a design might occur over a significant time period and span multiple screens, searching for cycles among RTs can be utilized by a software engineer to identify behavior that could be potentially dangerous.

These and other analyses are overviewed in [18], with a detailed discussion elsewhere [19, 23].
Figure 5 shows the result of an upward inheritance analysis run on the ElectricalPart object type.

![Diagram of ADAM Application Analysis: OT-based]

**Analysis Specifications:** Subject: ElectricalPart
- ISA Upward
- ISA Downward
- ISA Direct
- ISA Superclass
- ISA Subclass
- ISA Neighbor
- ISA Cycle

**Analysis:** OT Names, Descriptions
- Attribute Names, Descriptions
- Method Names, Descriptions, Parameters
- OT Names, Descriptions

Figure 5: Sample Upward Inheritance Analysis On The ElectricalPart Object Type.

3 Inheritance in ADAM

One of the differentiating features between our object-oriented design model and others is our characterization of inheritance. At the type level, this involves an understanding of which attributes and methods of the supertype are available for use by the subtype. Different programming languages have different interpretations of inheritance: C++ [40] allows the public and protected information to be acquired by the subtype from the supertype, while making the private parts unavailable; Eiffel [28] only allows public methods to be inherited. The inheritance concept has also played different roles in programming languages and database systems [6]. In an object-oriented programming language, inheritance is incorporated to allow instances of different OTs...
to be treated in a uniform fashion, when it is semantically meaningful to do so, i.e., substitutability [46]. This contrasts with the database interpretation when work on generalization was first conducted, where the underlying motivation for inheritance was to allow "... relevant details to be introduced in a controlled manner ..." [39, page 105], which emphasizes abstraction.

The object-oriented design model supports several variants of inheritance which are defined and explored in this section. This includes a range of variants to represent different scenarios under which a subtype acquires behavior from its ancestors. The main purpose is to offer the software engineer alternatives when designing an application, thereby allowing the inheritance behavior to be carefully specified based on needs and requirements. This behavior runs the spectrum from totally accessible to totally private, depending on the situation that is being modeled. Once an inheritance behavior or variant has been defined, the semantics of the generated code are predetermined. ADAM’s support for inheritance is demonstrated, with a detailed presentation of the inheritance variants and their realization in Ada 95, C++, and Eiffel.

3.1 Inheritance Variants

ADAM provides different variants of inheritance, both at the type-level and the instance-level. A type-level (T-LEVEL) variant focuses on the acquisition of attributes and/or methods by the subtype from the supertype. Thus, T-LEVEL variants have a direct impact on the definition of OTs. An instance-level (I-LEVEL) variant complements T-LEVEL considerations by allowing the run-time behavior of each subtype to be defined. T-LEVEL variants focus on what information of the supertype is available to the subtype, and I-LEVEL variants focus on if and how instantiation can occur.

There are four T-LEVEL variants, which correspond to different combinations of the supertype attributes and methods that are available for use by the subtype. The need for the different T-LEVEL variants was motivated, in part, from our assumptions in Section 2.1. Recall that we have assumed that only private data is permissible (at a design level), with the methods partitionable into public and private subsets. Given two object types, their potential interactions and sharing, via inheritance, at a type level can be quantified using the following definition:

Definition 5: \( OT_2 \) ISA \( OT_1 = (OTName_1, \{D_1\}, \{M_1\}) \Rightarrow \)

Full: \( OT_2 = (OTName_2, \{D_1, D_2\}, \{M_1, M_2\}) \) inherits both private attributes and public and private methods from both the type and its supertype.

Methods: \( OT_2 = (OTName_2, \{D_2\}, \{M_1, M_2\}) \) same as Full, except private attributes from the supertype are not inherited.

Public Methods: \( OT_2 = (OTName_2, \{D_2\}, \{P\mathcal{M}_1, \mathcal{M}_2\}) \) same as Methods, except only public methods from the supertype are inherited.
Restr. \( OT_2 = (OT Name_2, \{D_2\}, \{M_2\}) \) methods of the supertype only available by the permission of the software engineer responsible for the design of \( OT_2 \).

In the definition, we allow \( OT_2 \) to be considered as a subtype of \( OT_1 \), which lets \( OT_2 \) be used when an \( OT_1 \) instance is expected (substitutability). The four different T-LEVEL variants of inheritance correspond to different conceptualization alternatives for a software engineer constructing a design. In some situations, a software engineer may desire that all information is passed from supertype to subtype, accomplished via Full inheritance. In other situations, all of the methods (Methods) or just the public ones (Public Methods) may be passed. When absolute protection is needed, Restr is chosen to fully hide the information in the supertype from the subtype.\(^1\) The intent of the four different T-LEVEL variants is to offer the software engineer design-level choices to more accurately model the inheritance among OTs. These different choices are then realized in the implementation via automatic code generation in respective languages. Note that not all object-oriented languages can support all inheritance variants, as we will shortly discuss.

The I-LEVEL variants for inheritance represent the allowable techniques for instantiating OTs within an inheritance hierarchy. Abstract inheritance supports situations when the instantiation of an OT is neither desirable nor possible. Leaf, a database version of inheritance found in semantic data models such as Daplex [37], is a restricted version of instantiation that only supports the creation (deletion) of instances at the leaves of the inheritance hierarchy. Regular is representative of a programming language version of inheritance, and allows instantiation to occur at any node in the type hierarchy.

The ADAM environment supports inheritance definition during the OT-specification design phase of ADAM (see Section 2.2 again). The I-LEVEL behavior is supported through the addition of three different instantiation options for object types, i.e., options for Abstract, Regular, and Leaf. This allows the software engineer to select the instantiation behavior of an object type as it is being specified. The T-LEVEL behavior is defined during the actual specification of inheritance between two OTs. When two object types are related via the inheritance relationship, the software engineer must decide what supertype information is available to the subtype by selecting from one of the four T-LEVEL options (Full, Methods, Public Methods, and Restr). I-LEVEL and T-LEVEL variant information is part of the OT profile as discussed in Section 2.2 and illustrated in Figure 2.

\(^1\)This case is related to our security research as detailed in Section 2.1.
<table>
<thead>
<tr>
<th>Inherit Variant</th>
<th>Implementation Language</th>
<th>Design-Level Private Attribute</th>
<th>Design-Level Public Method</th>
<th>Design-Level Private Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full</strong></td>
<td>Ada 95 (Child-Pkg)</td>
<td>Private Section</td>
<td>Public Section</td>
<td>Private Section</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>Protected</td>
<td>Public</td>
<td>Protected</td>
</tr>
<tr>
<td></td>
<td>Eiffel</td>
<td>Feature-Any</td>
<td>Feature-Any</td>
<td>Feature-Any</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td>Ada 95 (Non-Child)</td>
<td>Private Section</td>
<td>Public Section</td>
<td>Under Research</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>Private</td>
<td>Public</td>
<td>Protected</td>
</tr>
<tr>
<td></td>
<td>Eiffel</td>
<td>Under Research</td>
<td>Feature-Any</td>
<td>Feature-None</td>
</tr>
<tr>
<td><strong>Public Methods</strong></td>
<td>Ada 95 (Non-Child)</td>
<td>Private Section</td>
<td>Public Section</td>
<td>Private Section</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>Private</td>
<td>Public</td>
<td>Private</td>
</tr>
<tr>
<td></td>
<td>Eiffel</td>
<td>Under Research</td>
<td>Feature-Any</td>
<td>Feature-None</td>
</tr>
<tr>
<td><strong>Restr</strong></td>
<td>Ada 95</td>
<td>Package Body</td>
<td>Package Body</td>
<td>Package Body</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>Private</td>
<td>Private</td>
<td>Private</td>
</tr>
<tr>
<td></td>
<td>Eiffel</td>
<td>Under Research</td>
<td>Under Research</td>
<td>Under Research</td>
</tr>
</tbody>
</table>

Figure 6: Language Specific Inheritance Variants.

3.2 Inheritance Variants and Automatic Code Generation

The instantiation and inheritance variants selected during a design by a software engineer dictate the automatic code that is supplied by ADAM. Specifically, given the fact that there are four T-LEVEL variants (*Full, Methods, Public Methods, and Restr*) and three I-LEVEL variants (*Abstract, Regular, and Leaf*), there are a total of 12 possible combinations. Each combination has a realization in the automatically generated code that is programming-language specific. In the following, we examine the degree to which we attain the instantiation and inheritance variants supported by the object-oriented design model in the Ada 95, C++, and Eiffel that is automatically generated by ADAM.

One consequence of ADAM's multiple programming language code generation from a common design is that not all the languages that we model completely support all inheritance variants of the object-oriented design model. From the perspective of the design model, we seek to offer a software engineer reasonable design alternatives which are useful in conceptualizing an application without regard to the implementation language. Unfortunately, this means that not all variants may be supported in all of the target programming languages.

Figure 6 contains a summary of the effects of the four T-LEVEL options on source code generation which we describe below. Ada 95 uses three visibility zones to control what portions of a parent type are accessible to child types. These zones provide access in a manner somewhat similar to public/protected/private in C++. In Ada 95, the type of inheritance is determined by
where/how the derived type is physically declared. In the parent type's Ada 95 package, primitive operations (methods) and components (attributes) declared:

- in the visible part of the parent package are accessible to any object in the same scope as an object declared to be of the parent type (similar to public in C++),
- in the private part of the parent package are accessible only to subtypes declared in a child-package (similar to protected in C++), and
- in the package body of the parent are accessible only internally to the parent package (similar to private in C++).

In Ada 95, the Full variant is attained by placing attributes and private methods in the private region, and public methods in the public region of the child-package enclosing the tagged (extendible) type. The Methods variant is partially achieved by placing attributes in the private region, and public methods in the public region of a non-child-package. This placement, which is an area of ongoing research, correctly results in attributes not being inherited by the subtype, but a conflict arises over where to place private methods so that only subtypes can access them. The Public Methods variant is accomplished by placing attributes and private methods in the private region, and public methods in the public region of the child-package enclosing the tagged (extendible) type. The Restr variant is implemented by placing all attributes and methods in the Ada 95 package body enclosing the object type definition.

As shown in Figure 6, the C++ protected category allows the protected members of an object to be treated as public by any subtypes of the object type while being treated as private to non-descendant users of the object type. In C++, the Full variant is attained by placing attributes and private methods in the protected region, and public methods in the public region. The Methods variant is achieved by placing attributes in the private region, public methods in the public region, and private methods in the protected region. The Public Methods variant is accomplished by placing attributes and private methods in the private region, and public methods in the public region. The Restr variant is implemented by placing all attributes and methods in the private region of the C++ class construct.

Eiffel supports a more explicit and flexible approach to granting access to objects than does Ada 95 or C++. Where C++ is limited to public, private, and protected, Eiffel uses a feature construct to allow the designer (programmer) to specify instances of which classes can access any attribute or method (routine). Typical usage of the Eiffel feature construct is:

- feature { NUNE }: No instances of any class can access the attribute or method. However, the access rights are inherited by subclasses, approximating the C++ protected designation.
- feature { ANY }: Similar to C++ public, this allows instances of any other class to access the attribute or method.
• feature { NAMED_CLASS }: Restricts access to the attribute or method to instances of the particular NAMED_CLASS class. There is no C++ nor Ada 95 construct that corresponds to this capability of Eiffel.

In Eiffel, the Full variant is attained by placing all attributes, public, and private methods under feature {ANY} access. The Methods and Public Methods variants are partially achieved by placing public methods under feature {ANY} access, and private methods under feature {NONE} access. However, we only partially attain our goals with the Methods and Public Methods variants in Eiffel since we cannot hide a parent’s attributes and/or methods from a child class. An implementation mechanism for the Eiffel Restr variant is a subject of ongoing research.

In addition to T-LEVEL issues, instantiation of OTs must also be handled by the different object-oriented programming languages for successful code generation. In Ada 95, the Regular option is attained by making the object type a tagged (extensible) type, while under the Leaf option, the type is not tagged. The selection of the Abstract instantiation option results in the inclusion of a single, abstract method in the type’s enclosing package specification. In C++, the selection of the Regular or Leaf I-LEVEL instantiation options guides the designation of the constructor and destructor. Under the Regular option, the constructor and destructor are designated as protected, while under the Leaf option, they are designated as private, for all non-leaf nodes, so that no instance of an internal node may be created. The selection of the Abstract instantiation option results in the inclusion of a single, pure abstract method in the header file of the class corresponding to the object type. In Eiffel, the Regular and Abstract options are decided by whether or not the deferred class construct is used. Research is ongoing with regard to implementing the Leaf variant in Eiffel.

Figure 7 shows the effect of inheritance on source code generation by contrasting the code generated for the Accessory OT. Only a subset of the 12 different T-LEVEL, I-LEVEL combinations has been shown, due to space limitations. Figure 7a shows the Eiffel (Regular, Full) variant. In Eiffel, the child class (Accessory) names its parent in the Inherit clause, thereby getting full access to functionality declared in its parent (BodyPart). Figure 7b shows an example of (Abstract, Full) inheritance in C++. In C++, this variant is modeled by having a pure virtual function (AbsBase()) in Accessory. Figure 7c depicts the (Leaf, Methods) inheritance variant in C++. Here, the leaf requirement is met by moving the constructor of Accessory to the private section of the child class. The Methods variant is implemented by placing the retail_cost attribute to the private section as well. Figure 7d shows Ada 95 code for the (Leaf, Public Methods) variant. Here the Public Methods variant is implemented by moving the SetR_Cost primitive operation (method) to the private portion of the package specification.
-- Accessory.e (Regular, Full) // Accessory.h (Abstract, Full)
-- subtype of BodyPart. // subtype of BodyPart.

class Accessory
inherit
BodyPart
creation
make
feature {NONE}
-- ATTRIBUTES
retail_cost: REAL -- cost of part
-- methods hidden from classes.
Setretail_cost(tempretail_cost:REAL) is
do
    retail_cost := tempretail_cost
end
-- methods available to classes.
Printretail_cost is
do io.putreal (retail_cost)
io.new_line
end
--class Accessory

a. (Regular, Full) in Eiffel.

// Accessory.h (Leaf, Methods) // Accessory.ads (Leaf, Public Methods)
// subtype of BodyPart
-- subtype of BodyPart

#include "BodyPart.h"
class Accessory:public BodyPart{
private:
    float retail_cost; // cost of part

// System defined Constructor
    Accessory();
    virtual ~Accessory();

// System defined protected methods.
protected:
    void Setretail_cost(float toSet);
// System defined public methods.
public:
    virtual void AbsBase() {};
    void Printretail_cost();
};

c. (Leaf, Methods) in C++.

b. (Abstract, Full) in C++.

d. (Leaf, Public Methods) in Ada 95.

Figure 7: Sample Code for OTs Including Inheritance.
3.3 Contrasting Language Capabilities

The three different object-oriented programming languages offer different capabilities in support of the inheritance variants provided by ADAM. Overall, out of the twelve inheritance variants, C++ supports all, Ada 95 supports all but the Methods variant, with Eiffel unable to fully support the Methods, Public Methods, and Restr variants (based on our current understanding of Eiffel and its capabilities). While on the surface it appears that C++ has an edge, it would be misleading to reach that conclusion. From a software engineering perspective, Ada 95's separation of a types' public interface (in its package specification) from the types' private implementation (in its corresponding package body) gives the strongest mapping of the languages considered to the principles of abstraction and object-orientation. A similar degree of separation is not possible in C++, where the header file contains the entire public and private interfaces for each class. Thus, from a software-engineering perspective, the approach that Ada 95 takes is far superior at attaining the different inheritance variants in the generated code, possessing the ability to hide attributes and methods when needed. The tradeoff then becomes the ability to support all variants in the generated code (as C++ does) against the ability to support them at not only code levels, but also from the important perspectives of software engineering principles and practice (as Ada 95 does).

4 Modeling Inter-Object Type Dependencies in ADAM

One of the most prominent features of modern, complex applications is the high degree of interdependency that exists among its different components. Unfortunately, one of the shortcomings of the object-oriented paradigm, and many object-oriented models, is the lack of support for modeling interdependencies between different portions of an application. While the object-oriented paradigm stresses low coupling and high cohesion, it is also imperative that some coupling is supported to properly model application requirements. Therefore, in order to approach a more complete specification of advanced application requirements, a relationship design construct has been included in the ADAM environment to represent the non-inheritance interdependencies between different portions of an application's information.

4.1 One-To-One and One-To-Many Relationships

Figure 4 showed a one-to-one relationship between OT Part and OT Supplier, indicating which supplier shipped a particular automobile part. The Part OT appears twice on the ADAM screen
in an effort to facilitate clarity in describing OTs that may be involved in many relationships. The reader should note that only a single, unambiguous, Part OT actually exists in this ADAM design (see Figure 2 again). Figure 8a shows the one-to-one relationship captured by the ADAM environment via the Eiffel SuppliedBy class. Figure 4 also indicated a one-to-many relationship between Supplier and Parts that the supplier is capable of delivering. Figure 3 showed the ADAM profile for the corresponding Supplies relationship. The one-to-many relationship needs a container object to keep track of the suppliers parts. Since Ada 95, C++ and Eiffel all provide a mechanism for generic types, ADAM uses appropriate container constructs to allow modeling of these relationships.

Figure 8b shows that ADAM models the one-to-many relationship in Ada 95 by instantiating from the generic container package, Gen_List, currently implemented as a linked list. Note the systematic definition, for each RT, of an Ada 95 general access type, in this case Supplies_Ptr. This gives the software engineer the freedom to allocate RT objects statically or dynamically, while still being able to reference them in a consistent fashion. Although not shown, we achieve the same effect in the other languages by using C++ templates and Eiffel generic classes. Regardless of which language is selected for code generation of one-to-many relationships, the implementor is automatically provided with all of the routines needed for the manipulation of the container object that holds the Part OT instances.

4.2 Group Relationships

The complexity of modern advanced applications requires object-oriented design support for collections of object types that demonstrate a wide range of structures and semantics including lists, sets, queues, and stacks. In order to adequately support the modeling of these collections, ADAM includes a GROUP construct that is further specialized into modeling constructs. The following is a representative set of aggregate constructs that have been identified [19] for use in ADAM:

**SET:** An unordered collection of OTs or RTs that disallows duplicate entries.

**LIST:** An ordered, possibly indexed sequence of OTs or RTs.

**TREE:** A collection of OTs and RTs organized in a tree structure.

**QUEUE:** A sequence of OTs or RTs that displays prioritized behavior.

These groups represent a first approximation of the different aggregations that will be useful for advanced applications. Currently, only the SET category of group has been implemented and supported in ADAM. As shown in Figure 4, the software engineer has elected to explicitly
-- SuppliedBy.e:
-- user-defined Relationship Type
class SuppliedBy

creation make
feature {NUME}
  source:Part
  dest:Supplier
feature {ANY}

-- System defined make feature
make is
do     !!source.make
      !!dest.make
end --make

-- System methods GET/SET source:
  GetSource:Part is
do Result := source
end --GetSource

SetSource (toSet:Part) is
do source := toSet
end --SetSource

-- Methods to access dest obj(s):
  GetDest:Supplier is
do Result := dest
end --GetDest
end -- class SuppliedBy

-- Supplies.ads
-- Interface for Supplies Group

with Gen_List;
with Supplier_OT; use Supplier_OT;
with Part_OT; use Part_OT;

package Supplies_RT is
  package PartList is new
    GenList(Part,Part_Ptr); use PartList;
  type Supplies is private;
  type Supplies_Ptr is access all Supplies;

-- System methods to GET/SET destination:
function GetDest(ASupplies_Ptr: Supplies_Ptr) return List_Ptr;
procedure SetDest(ASupplies_Ptr: Supplies_Ptr; The_List_Ptr: in List_Ptr);

private
  type Supplies is
    record
      Source : Supplier_Ptr;
      Dest   : List_Ptr;
    end record;
  end Supplies_RT;

a. One-To-One Relationship (Eiffel).

b. One-To-Many Relationship (Ada 95).

Figure 8: ADAM Relationships in Eiffel and Ada 95.
aggregate a group of Electrical Parts objects into a specific type of container. In the $A^2F^2$ example, there was a need to specify, at design time, that all Electrical Parts objects with more than two years of manufacture warranty are to be grouped together into a SET. This SET is shown in Figure 4 as: Set: ExtendedWarranty(ElectricalPart), with the corresponding ADAM Group profile shown in Figure 3. Figure 9 shows the ADAM generated C++ class to capture this SET relationship.

As shown for the ExtendedWarranty SET in Figure 9a, include statements are written in the header file for the participating object type, in this case ElectricalPart, and for the Set template, followed by the class header. Next, the Set template is used to declare the data structure to hold the participating ElectricalPart object type. ADAM automatically generates a constructor, and the four methods Add, Remove, Member, and NullSet that are used to manipulate the set. The method body of the ExtendedWarranty constructor, shown in Figure 9b, actually instantiates the Set template with the ElectricalPart type. The method bodies for the four
system-generated methods consist of a single line that calls the corresponding method of the Set template. The use of the Set template allows ADAM to generate simple, uniform, and reliable code, and avoids the production of large amounts of duplicate or similar code.

4.3 Contrasting Language Capabilities

Ada 95, C++, and Eiffel all offer similar capabilities for supporting one-to-one, one-to-many, and set relationships variants in generated code. The use of generic packages in Ada 95, templates in C++, and generic classes in Eiffel, all achieve the same result in generating consistent and uniform code for each relationship variant.

5 Related Work

The ADAM environment supports varying abstract views of design information similar to the views advanced by the work in the Project Support Environment (SPE) [5]. SPE's service-based reference model proposes that a CASE tool match a users' application system development process, thereby allowing for representation of system functionality from different user views. ADAM's ability to recall an object-oriented design from the Ontos database and generate design documentation formatted for $\LaTeX$, which has previously been presented in [16], supports the user views of system developer and maintainer. In contrast to MUSDE (multi-user software development environment) [3], ADAM has not yet addressed the issue of concurrency in the multi-user software development environment. ADAM has been used in the educational tool user view to support design methodology education [15, 30], but does not incorporate scenarios of object interaction like the Object Design Exploration (ODE) educational tool [36].

Object Based Specification Execution and Rapid Validation System (OBSERV) [43] supports building rapid prototypes of embedded systems with concurrency and real-time constraints, and is based on the object-oriented model, finite state machines, and logic programming. However, the OBSERV object-oriented model uses subsystem groupings, as opposed to the ADAM inheritance hierarchies. The INCASE CASE tool, Consolidated Data Modeling (CDM) [42], uses a graphical methodology similar to ADAM, for modeling entities, relationships, and logical tables. The INCASE abstraction is data-driven leading to object-based designs and code, but, unlike ADAM, fails to provide support for the object-oriented paradigm, notably inheritance and polymorphism.

The profiling activity at the heart of ADAM's object-oriented software architecture provides many parallels in current generation software development environments. Like ADAM, AUTO-MATED PROGRAMMER [21] forces the software engineer to formulate the solution "what" and
“how” before using the tool. Whereas AUTOMATED PROGRAMMER uses executable specifications, 2-D mathematical expressions, and technical English, ADAM uses the profiling activity to model descriptive aspects of the specification. ADAM’s profiles support consistency during software system construction through the ADAM design phases, which are analogous to the descriptions of structure and relations of the consistency model [22]. 3-D virtual images have been proposed [44] as a means through which to portray large numbers of objects, convey complex relationships, and concurrent dynamic execution. ADAM’s object-type and relationship type design phases provide an analogous abstraction of application object information. The Arcadia project’s [26] goal of developing an effective software development environment that was extensible and allowed for incremental improvement, are met, in part, in ADAM through the profiling concept.

6 Concluding Remarks and Ongoing Work

This paper has presented an in-depth examination of the ADAM environment, focusing on its capabilities for language-independent, object-oriented design. The emphasis of the paper has been on the object-oriented design model and its support for inheritance and relationship variants. These different variants offer software engineers alternatives that match the intents and requirements within their applications, supporting different perspectives and crossing object-oriented programming language boundaries. The intent of the variants is to offer disciplined and varied design choices to software engineers that when selected, results in automatically generated code that is consistent, predictable, readable, and understandable. Inheritance variants, as discussed in Section 3, differentiated on a type-level and instance-level basis, allow a software engineer to finely tune application behavior and have that behavior automatically realized in generated code. Relationship variants, as presented in Section 4, offer software engineers choices for interdependencies between object types and aggregations of individual object types, with generics utilized to simplify code generation while promoting reuse and minimizing potential inconsistencies. Collectively, inheritance and relationship variants, when coupled with automatic code generation in Ada 95, C++, and Eiffel, yield a versatile and flexible design environment.

In addition to the work presented herein, a number of other projects have been ongoing using the ADAM environment and its object-oriented design capabilities:

- The object-oriented design model and ADAM have been extensively researched to support discretionary access control via user-role based security concepts [8, 9, 10, 23, 24, 25]. Our current efforts in this area have been exploring the generation of secure object-oriented code via techniques that utilize inheritance, generics, and exceptions [11, 12].
• An ongoing effort seeks to expand the modeling capabilities of ADAM with the design-level equivalents of triggers and alterers, which we have termed propagations [31, 33]. This work has been underway with a major aircraft engine manufacturer that is interested in applying ADAM’s propagation and object-oriented techniques to mechanical design, from which ICAD code can be automatically generated.

• A new effort on reverse engineering and re-engineering is underway that seeks to take existing applications created using an object-based approach with Ada 83 code and reverse engineer the code to an ADAM design. Once the ADAM intermediate design has been defined, the application will then be re-engineered into an object-oriented design, with code re-generated in Ada 95.

• Another new effort is focusing on incorporating design for reuse concepts into ADAM, to create designs that stress reuse and to then re-use those designs in future applications. This work is considering the role of metrics for evaluating the degree that reuse is attained in a design.

ADAM has also been used in our undergraduate software engineering curricula as a medium to illustrate core object-oriented design concepts [41, 45, 46] to our students [15, 30].

References


