MMAIM: A Software Development Method for Ada
Part I--Description

Jeffrey R. Carter
Senior Engineer, Software
Martin Marietta Astronautics Group
P. O. Box 179
Denver, CO 80201

This work was funded by Martin Marietta Astronautics Group.

ABSTRACT
Traditional methods for software development fail to make use of Ada's unique features to support software engineering principles. Several methods have been proposed to overcome this problem, but shortcomings have been identified with all of them. A method for Ada software development is presented which avoids the shortcomings of existing methods. Further work required on the method is also discussed.

INTRODUCTION
Ada is the first programming language to provide features to support software engineering principles such as abstraction, encapsulation, software reuse, and information hiding. Ada promises, and to a certain extent has delivered [1], increased productivity and reduced costs through such features as compile-time interface checking, software reuse, and decreased maintenance. For these promises to be fulfilled, software developed in Ada must make full and good use of Ada's unique features: Reusable modules must be identified and designed for reuse, and the design must make use of abstraction, encapsulation, and information hiding to facilitate modification and maintenance. The software development method used for such software should assist in this process, as well as providing a medium for communication within the development project and with non-technical people ("users") outside the project.

Traditional methods, created before the development of Ada and intended for languages such as FORTRAN and COBOL, do not provide the support Ada needs. In some cases, these methods actively work against the development of quality Ada. For example, consider the use of functional decomposition on Ada's intended application area, large embedded systems. The decomposition of such a system may descend ten or more levels, and something as basic as access to a real-time clock may only appear at the lowest level, and there in a number of widely separate circumstances. A number of developers will probably have each performed these decompositions. It is almost impossible to find all these references and to identify that they need to be pulled together into a reusable package called CALENDAR.

Recognizing this, a number of researchers have created methods for Ada software development. Nielsen and Shumate [2] describe several such methods and discuss their shortcomings. Unfortunately, the method they propose has serious shortcomings in the encapsulation decisions it produces and in identifying reusable components [3]. I have not been able to find any method which meets all of the requirements mentioned above.

Because of this vacuum, efforts are underway at Martin Marietta to create a satisfactory software development method for Ada. This article describes the current state of the Martin Marietta Ada Implementation Method (MMAIM). Although MMAIM is still being developed, its basic concepts are sufficiently well-defined that it can be described. Hopefully its good points will influence others working in the field. Of course, the open discussion this description will generate will benefit MMAIM's further enhancement.

MMAIM

MMAIM is intended to satisfy a number of goals:

1. be useful on large projects with a number of developers
2. be useful on real-time or embedded systems
3. use a graphic notation which is easy to draw by hand
4. use a graphic notation which is sufficiently intuitive to be meaningful to non-technical people who have not been trained in the notation
5. use a graphic notation which can be mechanically translated into Ada code

Point 3. is important to facilitate communication within the development project. While discussing alternatives, it is common for developers to draw sketches for each other. In my experience with a number of methods I have found that a notation which cannot be easily drawn by hand will not be used for actual development. It may be used after the fact if such use is mandated, but the actual development which it records will be done with another method, or, too often, with no method. This is a failing of PAMELA [5]. Point 4. is important to facilitate communication with users. A cryptic notation will reinforce the conception that software is unreliable and to be avoided if possible. MMAIM's development was influenced by Buhr's edges-in approach [4], PAMELA [5], and the object-oriented concepts of Booch [6] and Seidewitz and Stark [7].

Entities. MMAIM is an entity-oriented development method. An entity models a physical or logical "thing" in the problem domain. Entities encapsulate all of the information and behavior of the thing, and hide information about the thing which is not needed for other entities to communicate with or make use of the thing. Entities are frequently abstract data types or abstract state machines.

The entity concept was developed in an attempt to combine the good points of "objects," as in object-oriented design, and "processes," as in process abstraction methods. MMAIM differs from object-oriented methods in recognizing that not all aspects of object-oriented languages can or should be emulated in Ada. It differs from process abstraction methods by considering control flow to be as important as data flow, and in the way it makes and represents encapsulation decisions. Entities are not strictly objects or processes, but they can be when it is useful, if the developer so desires. In MMAIM, entities are named, and their names are legal Ada identifiers. This simplifies the mechanical translation into Ada. Since entities are representations of things, their names should be nouns.

Entities can be differentiated in a number of ways. The first distinction is made between external and internal entities. An external entity is a physical thing in the environment with which the system interacts. An internal entity is a model of a physical or logical thing and is part of the system. External entities are modeled in MMAIM to facilitate an edges-in approach to identifying reusable interface entities.

A second distinction is made between active and passive entities. Passive entities do not appear to be able to take action unless they are called by another entity, while active entities can take independent action. The "do not appear" aspect is important. A passive interface entity to a transient, non-interruptive external event may be able to answer the question, "Has this event occurred since you were last instructed to clear your memory of the event?" Such an entity will appear passive to the outside, but to be able to answer this question it must be constantly scanning for an occurance of the event, which is certainly active behavior.

While part of the MMAIM method involves making active/passive decisions, it is important to realize that this distinction can be eliminated from MMAIM without affecting its usefulness. An active entity can always do the work of a passive entity, which gives rise to the general rule, "When in doubt, make it active." If desired, it can be changed to passive later.

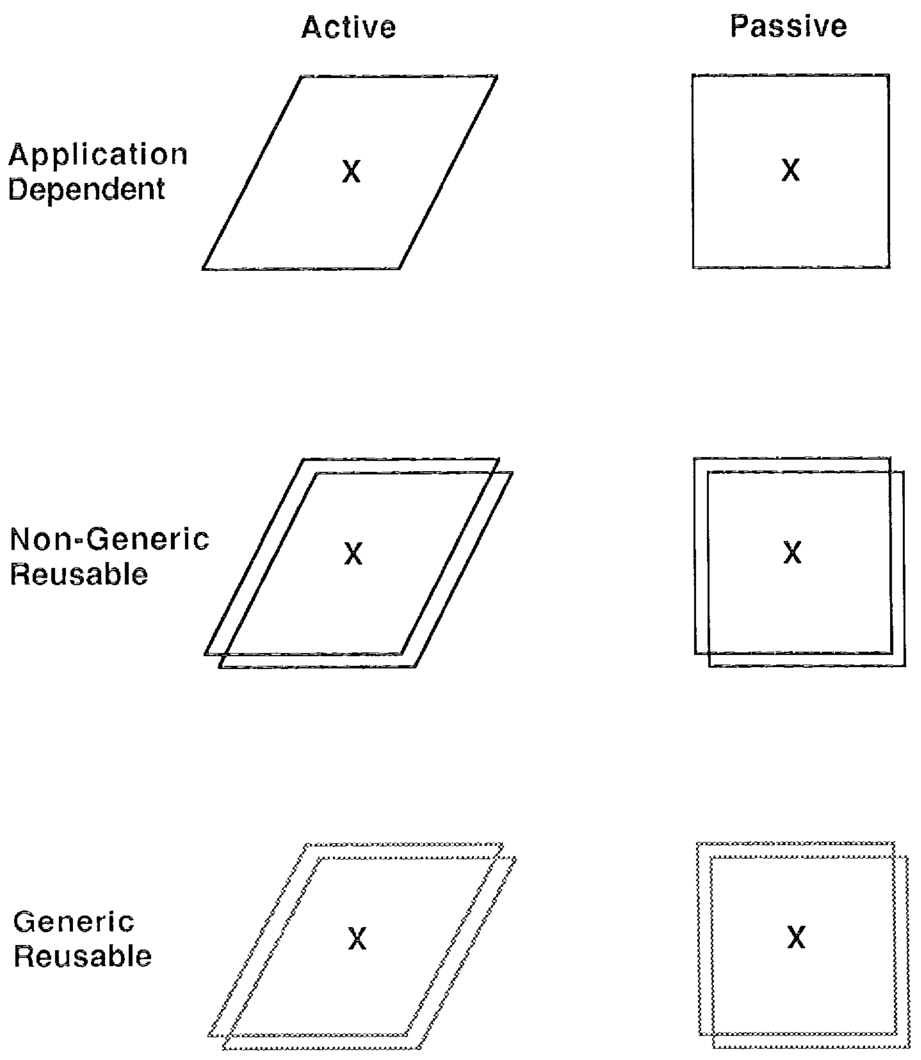
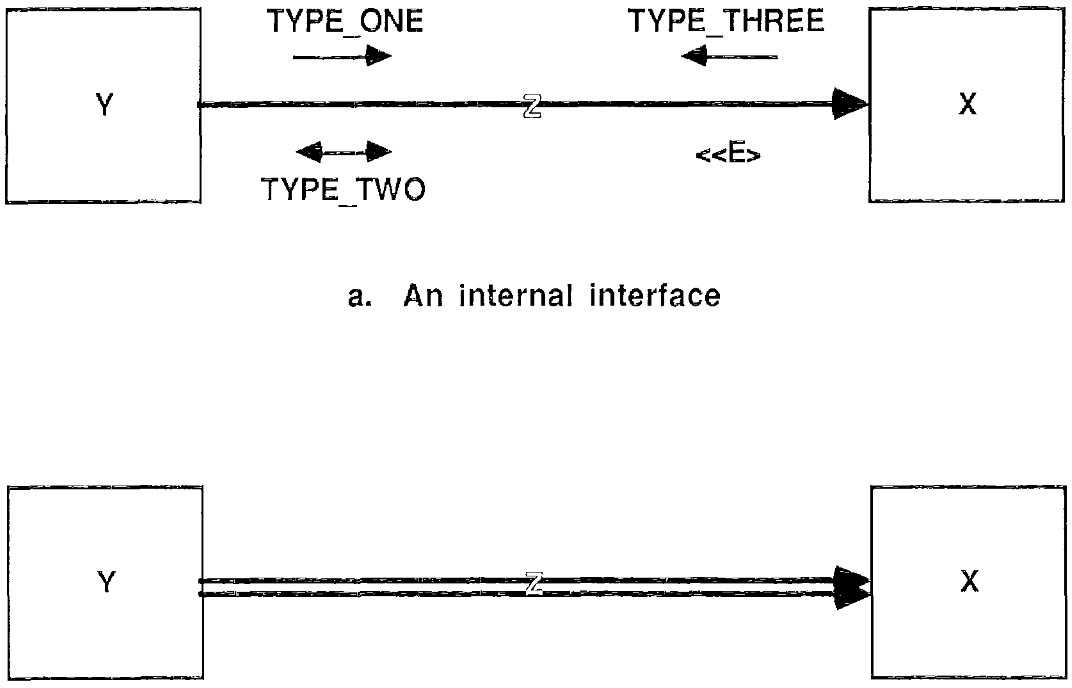


Figure 1.
MMAIM Notation for Entities

The third distinction made among entities is between reusable and application-dependent entities. This is further refined by distinguishing between generic and non-generic reusable entities. Generic entities are drawn with dashed lines, while non-generic entities are drawn with solid lines.

MMAIM's graphic notation for entities is given in Figure 1. The distinctions between non-reusable, non-generic reusable, and generic reusable entities are orthogonal to the distinction between active and passive entities. The distinction between external and internal entities is not made by the notation, but rather by the graph on which the entity appears. All of the entities in Figure 1. are named "X." In a valid MMAIM graph, each entity name must be unique.

Interfaces. Entities may have interfaces which allow other entities to communicate with them. An interface may be used to pass some information to an entity, to obtain some information from an entity, to request the entity to perform some action, or any combination of these. Passive entities must have at least one interface; active entities may have interfaces, but do not need to have one. Like entities, interfaces are named; unlike entities, these names need not be unique. A single entity may have several interfaces with the same name. In this case the interfaces may be distinguished by different data and exception flows through the interface. Since interfaces represent actions, their names should be verbs.



a. An internal interface

b. An external interface

Figure 2.
MMAIM Interfaces

MMAIM has two kinds of interfaces: internal and external. In effect, external interfaces show calls which pass through the system boundary and internal interfaces show calls which are internal to the system. An internal interface is an internal entity's interface which is called by another internal entity. An external interface is an internal entity's interface which is called by an external entity, or an external entity's interface which is called by an internal entity.

MMAIM's graphic notation for an internal interface is given in Figure 2a. The large arrow connecting two entities is the interface. The entity at the point of the arrow is the entity which has the interface. Here the entity "X" has an interface "Z," which is called by the entity "Y." The small arrows indicate data flows. All three possible kinds of data flows are shown. The names associated with the data flows are type names. Figure 2a. shows Y supplying a value of type "TYPE_ONE" when it makes the call, and X returns a value of type "TYPE_THREE." Y also supplies a value of type "TYPE_TWO," which is modified by X. These three kinds of data flows correspond to the Ada parameter modes **in**, **out**, and **in out**, respectively. An identifier enclosed in angle brackets ("<" and ">") near the interface is an exception flow. The direction of the flow is indicated by a double angle bracket. Figure 2a. shows an exception, "E," which may be raised by X during the call and propogated back to Y. The double angle bracket shows the exception flowing from X to Y.

If a type name is not one of Ada's predefined types, MMAIM requires that its definition be given and the entity which declares it be identified. In the interests of encapsulation and information hiding, the priority ranking is: Private types are first, predefined types in the middle, and other types last. This is not to say that non-private, non-predefined types are bad, but only that their use may reduce the modifiability of the system.

Sometimes several entities call the same interface of another entity. In this case MMAIM shows multiple occurences of the interface, one for each calling entity. These can be seen to be the same interface because they all have the same name and data and exception flows.

MMAIM's notation for an external interface is identical to an internal interface except that the arrow is drawn with a double line, as shown in Figure 2b. External interfaces may be named, but need not be.

Interface attributes. Sometimes an entity is not always willing or able to respond to a call to an interface immediately, sometimes it only waits a certain amount of time when it is willing to respond to a call, and sometimes a calling entity is only willing to wait a certain amount of time for a response to a call. When such decisions are made, MMAIM records them as interface attributes. There are two kinds of interface attributes: blocks and clocks.

Figure 3. shows an interface with interface attributes. The black circle at the point of the arrow is a block. It shows that the entity may not immediately respond to a call to this interface. For example, if the entity has two mutually-exclusive interfaces, it cannot respond to one while it is responding to the other. The clock by the point of the arrow shows that the called entity waits for a certain amount of time to see if any entity will call the interface, and then goes on to other things if no calls are pending. The clock by the other end of the arrow shows that the calling entity waits for a certain amount of time for the called entity to respond to the call; if it does not respond, the calling entity goes on to other things. In both cases the amount of time may be zero, in which case a calling entity requires an immediate response and the called entity requires a caller to be waiting when it checks for calls.

MMAIM requires that all interface attributes be labelled, although these labels are not part of the graph. A block's label describes why the entity may not respond to a call immediately. A clock's label describes why the entity needs a time-out on the interface or call, and what it does if the time-out occurs.

METHOD

MMAIM's system development methodology employs these notations through four major steps:

1. Create the External Entity Graph (EEG)
2. Create the top-level Entity Interaction Graph (EIG) and its Ada code
3. Create EIG's for the non-primitive entities and their Ada code
4. Create Behavior Transition Graphs (BTG's) for the primitive entities and their Ada code

These steps are not necessarily done sequentially. Each step may influence the preceding steps and cause a return to a preceding step. In addition, BTG's and lower-level EIG's may be worked on at the same time. Steps 3. and 4. are iterated until there are no non-primitive entities and all primitive entities have been decomposed into

BTG's.

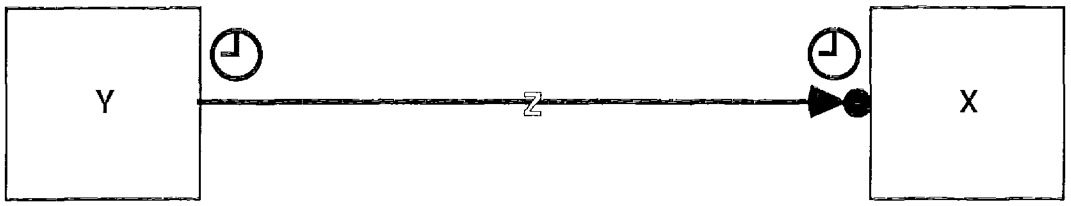


Figure 3.
MMAIM Interface Attributes

Traditionally, software development is divided into two distinct main phases. The first, sometimes called "analysis," "specification," or "system design," is concerned with determining what the system must do. The second, usually called "design," is concerned with deciding how the system will do what it must do. In an ideal world these two phases would be completely separate, and it would be possible to determine what a system must do without considering how it will do it.

Experience and research [8] show that in the real world it is impossible to separate the "what" from the "how." During most of the development process it is necessary to consider both. For this reason MMAIM's notation seeks to be useful for both kinds of information, and the method does not make a distinction between phases.

External Entity Graph. An EEG describes a system's interaction with its environment. It shows the external entities with which the system interacts, and what the flows of control and information across the system boundary are. It is useful for determining, recording, and recalling the overall function of the system. The EEG consists of the system to be developed, represented as a single reusable entity, the external enitities with which it interacts, represented as non-reusable entities, and the interfaces between the system and the external entities. Only external interfaces are shown on an EEG. Interfaces in which an external entity calls the system usually represent interrupts.

The EEG is a tool which developers use to express their understanding of the requirements to the users, for whom the system is being developed. This is done with preliminary EEG's, usually in combination with preliminary top-level EIG's. The final EEG helps record the requirements for the system to be developed. When, as is common in real life, the requirements change during development, this usually changes the EEG. Since the EEG is the starting point for the rest of the development, this ensures that the rest of the system will change in some manner to accomodate the change in requirements.

While creating the EEG, the developer examines alternatives and considers the correct representation of the environment for the system. For example, should the communications hardware be represented as one entity, or as separate input and output entities? Such decisions will affect the make-up of the top-level EIG. If the hardware is represented as two entities, it may be hard to find the existing reusable software which models it as a single entity. If this software's existence is discovered, the EEG should be changed to represent the hardware as a single entity. In this way the EEG affects the top-level EIG, and vice versa.

Entity Interaction Graph. An EIG decomposes a higher-level entity into the interaction of a set of potentially concurrent entities. The top-level, or system, EIG decomposes the system entity from the EEG, and inherits the external interfaces from the EEG. The top-level EIG is developed beginning with an edges-in approach. A single, reusable interface entity is identified for each external entity. In this way reusable components for interfacing with the environment are identified, and it can be decided whether software which exists can be reused, or if the components must be implemented for reusability. According to the MMAIM method, such components should be primitive, providing only basic functionality.

Once the interface entities have been identified, the remaining major entities of the system are identified, representing the major functionality of the system. The interfaces in these entities, which entities call other entities' interfaces, and the data and exception flows are filled in. When the EIG is complete, Ada package specifications can be mechanically produced from the graph and compiled. The main program can also be produced. It will usually be null.

It is rare for a satisfactory EIG to be produced on the first try. A number of alternatives must be considered before the best solution can be determined. For a given set of entities, the direction of calls can be varied. Interface attributes must be considered. Each entity must be considered to decide if it is active or passive. Most importantly, entities must be examined to determine their "entity-ness," how well they serve to encapsulate a thing or concept and hide the unnecessary information about the entity.

The entities on an EIG may be primitive or non-primitive. A primitive entity is sufficiently compact and well-defined that it may be implemented directly; a non-primitive entity should be decomposed into another EIG in a recursive manner. In this way some of the entities in the top-level EIG produce a hierarchy of EIG's and their corresponding Ada code. The Ada code for non-primitive entities consists primarily of declarations and body stubs.

Behavior Transition Graph. Each primitive entity is decomposed into a BTG. A BTG is similar to a state transition diagram, except the boxes represent actions the entity can take and the arrows connecting the boxes show the conditions which cause the entity to change from one behavior to another. The Ada code produced from a BTG will usually implement a body which was stubbed at a higher level.

EXAMPLE

In Part II of this paper I will demonstrate the use of MMAIM with a version of the Remote Data Acquisition System (RDAS) problem given by Cherry [5]. This problem is sufficiently small for presentation, but adequately complex and concurrent to give a good feel for all of MMAIM's features.

FUTURE WORK

All of the problems on which MMAIM has been used have been "artificial" problems, like the RDAS example which will be presented in Part II of this paper. I hope in the near future to use MMAIM and to have MMAIM used on large, real-world problems at Martin Marietta. This is essential for refining MMAIM into a final, usable form.

All of the figures in both parts of this paper have been produced using MacDraw on a Macintosh, and all of the Ada code has been produced manually. I hope to develop an automated tool which would allow the graphs to be easily produced, would enforce MMAIM's rules in the process, and would perform the mechanical production of Ada code from the graphs. This tool should be useful on large projects with many developers. There are some issues which need to be resolved to do this.

SUMMARY

Producing quality Ada systems requires a development method which makes use of Ada's unique features, but no existing method is satisfactory. MMAIM is presented as a consistent method which takes advantage of Ada's features and so fills this need. The example in Part II will demonstrate the use of MMAIM on a fairly large problem, and show that the results of using MMAIM are robust and easily accommodate a major and likely change to the system. Implementation as an automated tool should increase MMAIM's usefulness, but MMAIM should be of use as-is to developers of Ada systems.

REFERENCES

[1] Castor, V., and D. Preston, "Programmers Produce More with Ada," *Defense Electronics*, 1987 Jun.

[2] Nielsen, K., and K. Shumate, "Designing Large Real-Time Systems with Ada," *Communications of the ACM*, 1987 Aug.

[3] Carter, J., and C. McKeen, letter submitted to the "Forum" of the *Communications of the ACM*.

[4] Buhr, R., *System Design with Ada*, Prentice-Hall, 1984.

[5] Cherry, G., *PAMELA™ Designer's Handbook*, The Analytic Sciences Corp., 1986.

[6] Booch, G., *Software Engineering with Ada*, Benjamin/Cummings, 1983.

[7] Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Ada Lifecycle," in *Proceedings of Joint Ada Conference*, U. S. Army Communications--Electronics Command, 1987.

[8] Swartout, W., and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation," *Communications of the ACM*, 1982 Jul.

[9] Brooks, F., Jr., "No Silver Bullet," *Computer*, 1987 Apr.

From: *Ada Letters* Volume VIII, Number 3, May/June 1988