CHAPTER 4 SOFTWARE DESIGN

The design of software evolves through a partitioning process that defines and specifies elements of the software solution (i.e., software modules or software components) related to parts of the practical application problems defined during requirements analysis. The design is complete when each part of the specification is solved or carried out by one or more software elements. This process represents a transition between software requirements analysis and software design.

Chapter 4 is devoted to the description of software design methodologies. The chapter starts with a description of the software design document. Section 4.2 presents a discussion on the criteria for software design where the concepts of coupling, cohesion, information hiding, complexity, adaptability, testability, and reusability are discussed. Section 4.3 describes design methodologies for real-time software. The extensions for structured design methods such as the Design Approach for Real-Time systems (DARTS), and the Objected-Oriented Design (OOD) methods among others are discussed. Design examples using Teamwork/SD and Object Team/OOD are then presented in section 4.4 and section 4.5, respectively.

4.1 Introduction

As mentioned in section 2.3 of Chapter 2 on software development standards, the first step in the design phase is to design the software architecture. Architectural design deals with the construction of what is termed as “the physical model” of the software. It defines the architecture or the structure of the software interns of a hierarchy of software components or modules and data structures. The coupling between software modules in terms of data flow and control flow information must be clearly defined.

Several alternative designs could be proposed one of which is selected for further refinement and detailed design. The construction of possible designs and the selection between them should be governed by the set of software design criteria described in the following section. Technically difficult or critical parts of the design should be identified, and prototypes could be built to analyze the validity of the design assumptions used.

The deliverables produced in this phase are the software design document and the interface design document. According to the MIL-498 standard the effort should define and document the design decisions the software product or the software configuration item (CSCI) under development (that is, decisions about the CSCI's behavioral design and other decisions affecting the selection and design of the software units comprising the CSCI). For example, a decision to use a set of reusable software components will be a decision that affects the selection and design of software units comprising the CSCI. The document should clearly define and record the architectural design of each CSCI, identifying the software units (or modules) comprising the CSCI, their interfaces, and a concept of execution among them. The detailed design specification for each unit is documented. The document should also show the traceability between the software units and the CSCI requirements specification document. Design information pertaining to interfaces between modules as well as external interfaces may be included in a separate document dedicated to interface design descriptions.

The Software Design Document (SDD) specified in the MIL-498 standard is described in the following paragraphs. The document specifies both the architectural design as well as the detailed design of a CSCI. The following description contains excerpts from the Software Design Document Data Item Description (SDD-DID). The reader is referred to this MIL-STD-498 standard document for a complete documentation specification.

The Table below gives the required sections of the SDD. The first two sections are similar to all the Mil-498documents. The Scope section consists of three subsections as follows:

- The Identification subsection contains a full identification of the system and the software to which this document applies, including, as applicable, identification number(s), title(s), abbreviation(s), version s), and release number(s);

- The System Overview contains a brief statement describing the purpose of the system and the software to which this document applies. It also must describe the general nature of the system and software; summarize the history of system development, operation, and maintenance; identify the project sponsor, acquirer, user, developer, and support agencies; identify current and planned operating sites; and list other relevant documents

- The Document overview summarizes the purpose and contents of this document and describes any security or privacy considerations associated with its use.

The Referenced documents section lists the number, title, revision, date, and source of all documents referenced in this specification.

----------Table 4.1----------

Table 4.1 Contents of the SDD

1. Scope.

1.1 Identification

1.2 System overview

1.3 Document overview

2. Referenced documents

3. CSCI-wide design decisions

4. CSCI architectural design

4.1 CSCI components

4.2 Concept of execution

4.3 Interface design

4.3.1 Interface identification and diagrams

4.3.x (Project-unique identifier of interface)

5. CSCI detailed design

5.x (Project-unique identifier of a software unit, or designator of a group of software units)

6. Requirements traceability

7. Notes

----------End of Table 4.1----------

The CSCI-wide design decisions section is needed to document CSCI-wide design decisions, that is, decisions about the CSCI's behavioral design (how it will behave, from a user's point of view, in meeting its requirements, ignoring internal implementation) and other decisions affecting the selection and design of the software units that make up the CSCI. If all such decisions are explicit in the CSCI requirements or are deferred to the design of the CSCI's software units, this section shall so state. Design decisions that respond to requirements designated critical, such as those for safety, security, or privacy, shall be placed in separate subparagraphs. If a design decision depends upon system states or modes, this dependency shall be indicated. Design conventions needed to understand the design shall be presented or referenced. Examples of CSCI-wide design decisions are the following:

a. Design decisions regarding inputs the CSCI will accept and outputs it will produce, including interfaces with other systems, HWCIs, CSCIs, and users (4.3.x of this document identifies topics to be considered in this description). If part or all of this information is given in Interface Design Descriptions (IDDs), they may be referenced.

b. Design decisions on CSCI behavior in response to each input or condition, including actions the CSCI will perform, response times and other performance characteristics, description of physical systems modeled, selected equations/algorithms/rules, and handling of unallowed inputs or conditions.

c. Design decisions on how databases/data files will appear to the user (4.3.x of this document identifies topics to be considered in this description). If part or all of this information is given in Database Design Descriptions (DBDDs), they may be referenced.

d. Selected approach to meeting safety, security, and privacy requirements.

e. Other CSCI-wide design decisions made in response to requirements, such as selected approach to providing required flexibility, availability, and maintainability.

In section 4, the CSCI architectural design is specified. The section is divided into the following subsections described below. If part or all of the design depends upon system states or modes, this dependency must be indicated. If design information falls into more than one subsection, it may

be presented once and referenced from the other subsections. Design conventions needed to understand the design must be presented or referenced.

- CSCI components (SDD section 4.1): This subsection shall:

a. Identify the software units that make up the CSCI. Each software unit shall be assigned a project-unique identifier.

Note: A software unit is an element in the design of a CSCI; for example, a major subdivision of a CSCI, a component of that subdivision, a class, object, module, function, routine, or database. Software units may occur at different levels of a hierarchy and may consist of other software units. Software units in the design may or may not have a one-to-one relationship with the code and data entities (routines, procedures, databases, data files, etc.) that implement them or with the computer files containing those entities. A database may be treated as a CSCI or as a software unit. The SDD may refer to software units by any name(s) consistent with the design methodology being used.

b. Show the static (such as “consists of”) relationship(s) of the software units. Multiple relationships may be presented, depending on the selected software design methodology (for example, in an object-oriented design, this paragraph may present the class and object structures as well as the module and process architectures of the CSCI).

c. State the purpose of each software unit and identify the CSCI requirements and CSCI-wide design decisions allocated to it. (Alternatively, the allocation of requirements may be provided in 6.a).

d. Identify each software unit’s development status/type (such as new development, existing design or software to be reused as is, existing design or software to be reengineered, software to be developed for reuse, software planned for Build N, etc.) For existing design or software, the description shall provide identifying information, such as name, version, documentation references, library, etc.

e. Describe the CSCI’s (and as applicable, each software unit’s) planned utilization of computer hardware resources (such as processor capacity, memory capacity, input/output device capacity, auxiliary storage capacity, and communications/network equipment capacity). The description shall cover all computer hardware resources included in resource utilization requirements for the CSCI, in system-level resource allocations affecting the CSCI, and in resource utilization measurement planning in the Software Development Plan. If all utilization data for a given computer hardware resource are presented in a single location, such as in one SDD, this paragraph may reference that source. Included for each computer hardware resource shall be:

1) The CSCI requirements or system-level resource allocations being satisfied

2) The assumptions and conditions on which the utilization data are based (for example, typical usage, worst-case usage, assumption of certain events)

3) Any special considerations affecting the utilization (such as use of virtual memory, overlays, or multiprocessors or the impacts of operating system overhead, library software, or other implementation overhead)

4) The units of measure used (such as percentage of processor capacity, cycles per second, bytes of memory, kilobytes per second)

5) The level(s) at which the estimates or measures will be made (such as software unit, CSCI, or executable program)

f. Identify the program library in which the software that implements each software unit is to be placed

- Concept of execution (SDD section 4.2): This subsection shall describe the concept of execution among the software units. It shall include diagrams and descriptions showing the dynamic relationship of the software units, that is, how they will interact during CSCI operation, including, as applicable, flow of execution control, data flow, dynamically controlled sequencing, state transition diagrams, timing diagrams, priorities among units, handling of interrupts, timing/ sequencing relationships, exception handling, concurrent execution, dynamic allocation/ deallocation, dynamic creation/deletion of objects, processes, tasks, and other aspects of dynamic behavior.

- Interface design (SDD section 4.3): This subsection shall be divided into the following parts to describe the interface characteristics of the software units. It shall include both interfaces among the software units and their interfaces with external entities such as systems, configuration items, and users. If part or all of this information is contained in Interface Design Descriptions (IDDs), in section 5 of the SDD, or elsewhere, these sources may be referenced.

1) Interface identification and diagrams (SDD section 4.3.1): This part shall state the project-unique identifier assigned to each interface and shall identify the interfacing entities (software units, systems, configuration items, users, etc.) by name, number, version, and documentation references, as applicable. The identification shall state which entities have fixed interface characteristics (and therefore impose interface requirements on interfacing entities) and which are being developed or modified (thus having interface requirements imposed on them). One or more interface diagrams shall be provided, as appropriate, to depict the interfaces.

2) Project-unique identifier of interface (sections 4.3.x): This part consists of several sections (starting from section 4.3.2). Each section shall identify a particular interface by project-unique identifier, shall briefly identify the interfacing entities, and shall be divided into subparagraphs as needed to describe the interface characteristics of one or both of the interfacing entities. If a given interfacing entity is not covered by this SDD (for example, an external system) but its interface characteristics need to be mentioned to describe interfacing entities that are, these characteristics shall be stated as assumptions or as "When [the entity not covered] does this, [the entity that is covered] will...." This paragraph may reference other documents (such as data dictionaries, standards for protocols, and standards for user interfaces) in place of stating the information here. The design description shall include the following, as applicable, presented in any order suited to the information to be provided, and shall note any differences in these characteristics from the point of view of the interfacing entities (such as different expectations about the size, frequency, or other characteristics of data elements):

a. Priority assigned to the interface by the interfacing entity(ies)

b. Type of interface (such as real-time data transfer, storage-and-retrieval of data, etc.) to be implemented

c. Characteristics of individual data elements that the interfacing entity(ies) will provide, store, send, access, receive, etc., such as:

1) Names/identifiers

a) Project-unique identifier

b) Non-technical (natural-language) name

c) DoD standard data element name

d) Technical name (e.g., variable or field name in

code or database)

e) Abbreviation or synonymous names

2) Data type (alphanumeric, integer, etc.)

3) Size and format (such as length and punctuation of a character string)

4) Units of measurement (such as meters, dollars, nanoseconds)

5) Range or enumeration of possible values (such as 0-99)

6) Accuracy (how correct) and precision (number of significant digits)

7) Priority, timing, frequency, volume, sequencing, and other constraints, such as whether the data element may be updated and whether business rules apply

8) Security and privacy constraints

9) Sources (setting/sending entities) and recipients (using/receiving entities)

d. Characteristics of data element assemblies (records, messages, files, arrays, displays, reports, etc.) that the interfacing entity(ies) will provide, store, send, access, receive, etc., such as:

1) Names/identifiers

a) Project-unique identifier

b) Non-technical (natural language) name

c) Technical name (e.g., record or data structure name in code or database)

d) Abbreviations or synonymous names

2) Data elements in the assembly and their structure (number, order, grouping)

3) Medium (such as disk) and structure of data elements/assemblies on the medium

4) Visual and auditory characteristics of displays and other outputs (such as colors, layouts, fonts, icons and other display elements, beeps, lights)

5) Relationships among assemblies, such as sorting/access characteristics

6) Priority, timing, frequency, volume, sequencing, and other constraints, such as whether the assembly may be updated and whether business rules apply

7) Security and privacy constraints

8) Sources (setting/sending entities) and recipients (using/receiving entities)

e. Characteristics of communication methods that the interfacing entity(ies) will use for the interface, such as:

1) Project-unique identifier(s)

2) Communication links/bands/frequencies/media and their characteristics

3) Message formatting

4) Flow control (such as sequence numbering and buffer allocation)

5) Data transfer rate, whether periodic/aperiodic, and interval between transfers

6) Routing, addressing, and naming conventions

7) Transmission services, including priority and grade

8) Safety/security/privacy considerations, such as encryption, user authentication, compartmentalization, and auditing

f. Characteristics of protocols that the interfacing entity(ies) will use for the interface, such as:

1) Project-unique identifier(s)

2) Priority/layer of the protocol

3) Packeting, including fragmentation and reassembly, routing, and addressing

4) Legality checks, error control, and recovery procedures

5) Synchronization, including connection establishment, maintenance, termination

6) Status, identification, and any other reporting features

g. Other characteristics, such as physical compatibility of the interfacing entity(ies) (dimensions, tolerances, loads, voltages, plug compatibility, etc.)

In section 5, the CSCI detailed design is specified. This section shall be divided into the following subsections to describe each software unit of the CSCI. If part of all of the design depends upon system states or modes, this dependency shall be indicated. If design information falls into more than one paragraph, it may be presented once and referenced from the other paragraphs. Design conventions needed to understand the design shall be presented or referenced. Interface characteristics of software units may be described here, in Section 4, or in Interface Design Descriptions (IDDs). Software units that are databases, or that are used to access or manipulate databases, may be described here or in Database Design Descriptions (DBDDs).

Each subsection (starting from 5.1) specifies a project-unique identifier of a software unit, or designator of a group of software units. Each subsection shall identify a software unit by project-unique identifier and shall describe the detailed design of the unit. The description shall include the following information, as applicable (Alternatively, this paragraph may designate a group of software units and identify and describe the software units in subparagraphs, Software units that contain other software units may reference the descriptions of those units rather than repeating information):

a. Unit design decisions, if any, such as algorithms to be used, if not previously selected

b. Any constraints, limitations, or unusual features in the design of the software unit

c. The programming language to be used and rationale for its use if other than the specified CSCI language

d. If the software unit consists of or contains procedural commands (such as menu selections in a database management system (DBMS) for defining forms and reports, on-line DBMS queries for database access and manipulation, input to a graphical user interface (GUI) builder for automated code generation, commands to the operating system, or shell scripts), a list of the procedural commands and reference to user manuals or other documents that explain them

e. If the software unit contains, receives, or outputs data, a description of its inputs, outputs, and other data elements and data element assemblies, as applicable. Paragraph 4.3.x of this DID provides a list of topics to be covered, as applicable. Data local to the software unit shall be described separately from data input to or output from the software unit. If the software unit is a database, a corresponding Database Design Description (DBDD) shall be referenced; interface characteristics may be provided here or by referencing section 4 or the corresponding Interface Design Description(s).

f. If the software unit contains logic, the logic to be used by the software unit, including, as applicable:

1) Conditions in effect within the software unit when its execution is initiated

2) Conditions under which control is passed to other software units

3) Response and response time to each input, including data conversion, renaming, and data transfer operations

4) Sequence of operations and dynamically controlled sequencing during the software unit's operation, including:

a) The method for sequence control

b) The logic and input conditions of that method, such as timing variations, priority assignments

c) Data transfer in and out of memory

d) The sensing of discrete input signals, and timing relationships between interrupt operations within the software unit

5) Exception and error handling

The task of Requirements traceability needed for verification and validation is documented in section 6 of the SDD. This section shall contain:

a. Traceability from each software unit identified in this SDD to the CSCI requirements allocated to it. (Alternatively, this traceability may be provided in 4.1.)

b. Traceability from each CSCI requirement to the software units to which it is allocated.

Finally section 7 of the SDD, entitled Notes, shall contain any general information that aids in understanding this document (e.g., background information, glossary, rationale). This section shall include an alphabetical listing of all acronyms, abbreviations, and their meanings as used in this document and a list of any terms and definitions needed to understand this document.

4.2 Software Design Criteria

The established criteria for software design quality help the designer in developing and assessing the quality of the software design architecture (i.e., they help in the process of partitioning the software into a set of coupled components). The software design criteria discussed in this section are as follows:

1. Modular design (coupling, and cohesion),

2. Information hiding,

3. Design complexity,

4. Testability, and

5. Reusability

The criteria of modular design, design complexity, and testability have evolved over the past several decades. These criteria have contributed to the evolution of the structured analysis and design techniques. Information hiding and reusability have evolved recently and have contributed to the evolution of the object-oriented development techniques. These design techniques are discusses later in this Chapter. The above criteria are briefly discussed in the following subsections.

4.2.1 Modular Design

Modular system design is one of the most well known powerful techniques for engineering systems design aimed at combating design complexity and enhancing the system maintainability. In software engineering, modularity is a measure of the extent to which the software is decomposed of software modules such that a change in one module has minimal effect on other modules. The question often arises on what is a software module?

A software module is a program unit consisting of procedures, functions, and data structures that is identifiable in terms of compiling (can be separately compiled), combining with other program units, and loading; an independently callable unit of code. The later characteristic of a module in terms of being an independently callable unit of code is used in structured design to show how the system functions are mapped into modules in the structure chart. The development of a modular design structure can be guided by the need for satisfying design criteria regarding the coupling of pairs of modules, and the cohesion, complexity, and reusability of individual components in each module. Coupling is a measure of interdependence between two modules. This can take the form of call relationship, parameter passing, and/or shared data area. Forms of coupling between modules will be described further below. One of the important design criteria is to reduce coupling between modules. Techniques for reducing coupling between modules will be discussed in the next section.

Reusability is now considered as an important design criteria after the emergence of software repositories which provide means of classifying, cataloging, and retrieving software components. Both domain specific and general reusable modules are sought out in the design of current software systems. In developing reusable modules, upper-level modules are likely to be domain specific, whereas lower-level service modules are more easy to generalize.

The concepts of coupling and cohesion are further discussed in the rest of this section. Complexity and reusability are discussed in the following subsections.

Coupling

As mentioned above coupling is a measure of level of interactions between two modules. Modules interact by passing parameters through a call statement, or by sharing common data and file structures. The goal of modular design is to reduce the level of coupling between modules. The following forms of module coupling are listed in the order of increasing level of coupling.

. Data coupling: communication between modules is accomplished through well-defined parameter lists (the parameter lists here include both input and output information items to/from a module). Every parameter in the list is either a simple element or a data structure all of whose elements are used in the target module. This is the most desirable form of coupling between modules.

. Stamp Coupling: similar to data coupling where a data structure is passed in an argument in the argument list between modules except in this case only a part of the data structure is used in the target module. This is a stronger form of coupling because the called module has access to data structure components defined in the calling module which are not needed and are not used. Stamp coupling can be avoided by redesigning the structure of data passed between modules.

. Control coupling: is the form in which a module controls the flow of control or the logic of another module. This is accomplished by passing control information items as arguments in the argument list. This is a stronger form of coupling because apart from using information passed from the source module for computational purposes, in this case the target module flow of control also depends on these information items.

. Common coupling: occurs when modules share common or global data or file structures. This is the strongest form of coupling both modules depend on the details of the common structure. A change in the type or form of the common structure will necessitate changes to propagate to all modules sharing this structure making the maintainability of the software more difficult to achieve. Sharing the access of common structures also requires extra code in concurrent

processing to protect the integrity of global data and to coordinate between simultaneous access requests using semaphores, monitors, or other mechanisms.

The form of data coupling is achieved when the designer always try to pass information between modules using parameter lists except when the same information is used by many modules (in which case the information become pervasive and some form of common coupling should be used). Data coupling tends to clearly define the interface between modules and make the internal structure of each module independent from other modules. Pervasive information should be put in a global module. Although this is a form of common coupling, it has been shown in practice that it is sometimes a necessary evil because of the large number of common constants, types and variables needed by many modules in a large complex system. In fact results of an empirical study of software design practices [Card 86] (also see [Lewis 1991], pp181-188]) shows that global or common coupling might result in a more reliable software (the percent of modules containing high number of errors were less in a system developed with global coupling).

Cohesion

As mentioned above, cohesion is a measure of internal relatedness of the components of a module. It is used by designers to develop the boundaries of modules as consisting of strongly cohesive components. The following forms of coherence are described in the order of decreasing levels of strength.

. Functional cohesion: is achieved when the components of a module cooperate in performing exactly one function. The name of the module should specify the nature of the function performed, for example, module names such as POLL_SENSORS, COMPUTE_SENSOR_RANGE, COMPUTE_FUEL_CAPACITY, GENERATE_ALARM, DISPLAY_SENSOR_DATA clearly specify a functionally cohesive module whose components and data structures are used only for the named function.

. Communicational cohesion: is achieved when software units or components sharing a common information structure are grouped in one module. Examples are different functions performed on the same data structure such as the statistical functions of computing the average, standard deviation, etc., of a real data read from a sensor. This form of cohesion can help in reducing the amount of information transferred between modules thus resulting in loosely coupled modules as desired.

. Procedural cohesion: is the form of cohesion obtained when software components are grouped in a module to perform a series of functions following a certain procedure specified by the application requirements. This form of cohesion also reduces coupling between modules by gathering procedurally dependent software components in the same module.

. Temporal cohesion: this form of cohesion is found when a module is composed of components or functions which are required to be activated during the same time interval. Examples are functions required to be activated during the initialization period, termination period, or at given periodical time interval. These temporal modules are common in real-time control software.

. Logical cohesion: refers to modules designed using functions who are logically related, such as input and output functions, communication type functions (such as send and receive), Numerical computation functions (e.g., matrix manipulation functions, differential equation solvers), and text editing, retrieving, and saving functions. Although matrix manipulation functions and differential equation solvers are logically related, putting them in the same module would make the module interface to other modules difficult to design. The interface must have a variable specifying which function within the module is required to be activated, and must have a means of supplying through the interface all the data structures and types, and names (or pointers) to other functions needed by the selected function. Complex module interfaces are one of the major sources of errors in software systems. This is because testing such a module will need much more effort (in terms of the number of test cases) than that required for modules with simple interfaces.

. Coincidental cohesion: is found when several unrelated or loosely related functions are grouped in one module to decrease the total number of modules and increase the module size. This type of cohesion complicates module testing and maintenance, and therefore should be totally avoided.

4.2.2 Information Hiding

Design decisions that are likely to change in the future should be identified and modules should be designed in such a way that those design decisions are hidden from other modules. This is the basic goal of the concept of information hiding is to hide the implementation details of shared information items by specifying modules called information hiding modules. In these modules the critical data structures and functions (i.e., those that likely to change and whose change will create scores of changes in the system) are hidden from other modules. Interfaces are created to allow other modules accessibility to shared data structures without having visibility to its internal implementations.

The concept of information hiding is based on the concepts of abstraction and encapsulation. Abstract data types specify a data structure type together with the set of operations or functions to be performed on an instantiations of that data structure type. Encapsulation is used to hide or encapsulate the implementation details of an abstract data type in an information hiding module.

An example of an information hiding module is one which is defined over a data structure called the sensor_data_buffer. Modules which periodically read sensor data update the relevant data in the sensor data buffer through a specific procedure called update_sensor_data. Concurrent calls to this procedure from different sensor modules will be queued and serviced. Another procedure

called get_sensor_data is called by other sensor data processing modules to read the current sensor data from the sensor_data_buffer. Other internal procedures are provided with sensor_data_buffer which implements filtering and preprocessing operations (such as scaling, noise filtering etc.). Other modules in the system cannot access the sensor_data_buffer directly, only access is allowed through a call to update_sensor_data or get_sensor_data procedures. The information hiding module, consisting of sensor_data_buffer, update_sensor_data, get_sensor_data, and the filtering and preprocessing procedures, is an example of data abstraction and encapsulation

4.2.3 Design Complexity

Complexity is another design criteria used in the process of decomposition and refinement. A module should be simple enough to be regarded as a single unit for purposes of verification and modification.

High complexity implies a complex logical structure and many relationships between the different parts of a module. Complex logical structures result, among other things, from deeply nested if-then-else, and function calling statements. Complexity measures has been one of the popular metrics used in software design. A number of measures have been proposed (see [SOM]), examples of these are:

1. Constantine and Yourdon proposed a measure of complexity for a given module proportional to the number of other modules calling this module (termed as fan-in), and the number of modules called by the given module (termed as fan-out). The fan-in number and the fan-out number can be easily obtained from a structure chart as will be shown later this chapter.

2. Henry and Kafura proposed another complexity measure based on a different form of fan-in and fan-out related to information items. The fan-in is the number of information flow items needed by the module and the fan-out equals the number items updated by the module. Informations flows here include the ones obtained through procedure calls, therefore, this measure includes Constantine/Yourdon fan-in and fan0out measure. This measure is also more suited to data-driven applications. The complexity measure is obtain as:

Complexity = length * (fan-in * fan-out) squared, Where length is equal to the estimated lines of code for the module or equal to McCabe Cyclomatic Complexity (MCC). MCC is a measure of the logical structure complexity of a module. It refers to the total number of conditions (if then) in the module (compound conditions consisting of n simple predicates are counted as n rather than 1).

4.2.4 Design for Testability

Design for testability is one of the important criteria for developing quality engineering products and engineering systems in general. In software engineering, software modules, like in other engineering products, must be designed to enhance testability. This in turn means the ease of developing a set of test cases (i.e., a set of specific input data values or events), and developing the testing software (also termed test drivers) required for the testing procedure of a given module or an integrated set of modules. Testing is performed at the module level, in which case it is termed as unit testing. It is also performed at the subsystem or system level, which is known as integration testing. Test case generation techniques are described at the end of Chapter 6. These techniques clearly show that the above criteria of modularity and information hiding enhance design testability.

The following are examples of designs which tend to be difficult to test or require a large number of test cases: These modules usually suffer from poor coupling or cohesion.

1. Modules which have a large number of inputs and outputs,

2. Modules which have no inputs and outputs (e.g., modules which only use and modify global data)

3. Modules whose interfaces pass unnecessary data,

4. A process whose functionality is spread over several unrelated modules (the functional requirements of this process traced to the requirements document will be difficult to test in the specified design),

4.2.5 Design for Reuse

Reusability is now considered as an important design criteria after the emergence of software repositories which provide means of classifying, cataloging, and retrieving software components. Both domain specific and general reusable modules are sought out in the design of current software systems. In developing reusable modules, upper-level modules are likely to be domain specific, whereas lower-level modules are more likely to be designed as generic. The specification, design, implementation, and testing of reusable components must be well documented in order to enhance the frequency of reuse, and hence achieve the proper return on investment on the effort spent in developing reusable components.

Reusable components must be designed in such a way to avoid special assumptions (e.g, using operations and data types which are satisfied only for a given application). In addition, dependencies (e.g., on specialized components, library functions, operating system functions or

primitives, or specialized hardware) which limit the component applicability and effectiveness must also be avoided.

The old subroutine libraries and the more recent toolkit class libraries (for object-oriented programming) are examples of code reuse. In fact library modules in general specify reusable components which are part of the new design as shown in section 4.4

Design Patterns

Reuse can also be utilized at the design level. For example the design patterns for object-oriented designs developed by Gamma at al ([Gamma 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns, Elements of Reusable Object-Oriented Software” Addison-Wesley, 1995) are an excellent example of reusable designs. Design patterns help reduce dependencies and provide a mechanism for loose coupling between classes. This is accomplished using techniques such as abstract coupling and layering. Design patterns also make an application more maintainable when they are used to limit platform dependencies They enhance extensibility by giving simple mechanisms used to extend class hierarchies and exploit object composition. The reduced coupling achieved by the using the design patterns also enhances extensibility. Extending a class in isolation is easier if the class does not depend on many other classes. The following list of design patterns gives examples of some of the most important design patterns described in [Gamma 95]:

1. Abstract Factory: is a design pattern intended to provide an interface for creating families of related or dependent objects without specifying their concrete (i.e., non-abstract) classes. This is achieved by having an Abstract Factory class in the application which defines the set of products needed in the application. For example, consider a user interface toolkit such as the one provided by Motif or Presentation Manager (PM). These applications create objects called “widgets” such as windows, scroll bars, and buttons.The Abstract Factory in this case is a Widget Factory which specifies the methods Create ScrollBar(), Create Window(), and Create Button(). The subtypes or subclasses Motif Widget Factory, and PM Widget Factory are used to specify the unique Create methods for each toolkit. Concrete classes of objects are then defined for Window, Button, and ScrollBar where the implementations of the Create methods are defined. In this case clients will interface with the Widget Factory class and the implementation details are buried in the concrete classes. The motivation for using this pattern is to isolate concrete classes, and making changing product families (such as Motif or PM) easy.

2. Adapter: This design pattern converts the interface of a class into another interface expected by the clients.The motivation of using this pattern is found when a class cannot be used only because its interface does not match the domain-specific interface required by another application.

3. Chain of Responsibility: This pattern avoids coupling the request of a sender to its receiver by allowing the request to be chained through several receiver objects until one handles the request

(e.g., a help request can be chained through a set of help information objects, the first object in the chain can either handle it or forward it.to the next object in the chain). This is definitely more general than coupling a request with one object.

4. Decorator: This pattern provides additional responsibilities to an object dynamically, and hence is considered to be a more flexible alternative to subclassify for extended functionality (The usual approach to provide extended functionality for a class of object is to define a subclass which specifies this added functionality)

5. Facade: This pattern provides a unified interface to a set of interfaces in a subsystem. This is achieved by defining a higher lever interface that makes the subsystem easier to use.

6. Mediator: This pattern defines an object that encapsulates how a set of objects interact. It enhances loose coupling by keeping objects from referring to each others explicitly, and hence the interactions between these objects can be varied depending on which Mediator is used

.

The above examples of design patterns are provided to give the reader a flavor of the concept of design patterns which are developed to be the elements of reusable object-oriented software. These concepts will be perhaps better understood when the reader covers the topic of object-oriented design introduced in section 4.3.2 and covered in more detail in section 4.5. For more detailed coverage of the concept of design patters, the reader is referred to [Gamma 95}.

4.3 Software Design Methodologies

The design methodologies determine how the software system modules and components are to be defined in order to accomplish the specifications produced in the previous phase. Following the structured analysis and object-oriented analysis methodologies used in the requirements phase in the pervious Chapter, the structured design methodology is introduced first in this section, then object-oriented design using the Object Team/OOD notation is briefly discussed. These methodologies are discussed with examples in the last two sections in this Chapter. These topics are introduced as language independent as possible. Language dependent design is covered in the next Chapter, Chapter 5, where the detailed design and implementation issues are addressed.

4.3.1 Structured Design

Structured design is characterized by the development of structured hierarchy of modules specified using Structure Charts (SCs). SCs are used for modeling the partitioning/grouping of tasks and functions defined in the specifications into modules, the hierarchical organization of these modules, and the data interfaces between them.

The basic units of a SC are the module, the call, the shared data area, and the couple. The module is an independently callable unit of code. The call is an activation of a module, while the shared data represents data that needs to be accessed from several modules. The couple represents an item of data or control information passed between modules. The modules in a SC are combined in a tree structure representing a true hierarchy where a module at a lower level may be called by more than one parent at a higher level. The notation for developing SCs will be discussed in section 4.4

Each module declared in the SC must be accompanied by a module specification (M-specs). The notation used in the M-specs are similar to those used in P-specs such as pre/post conditions, structure languages, and pseudo code. The amount of detail provided in the M-specs should depend on the destination programming language.

It is important to discuss and understand ways in which a SC can be developed from specifications represented by structured analysis graphs (data flow graphs and control flow graphs, and Control Specifications or C-specs). Since the higher levels of the hierarchy in the SC are responsible for controlling the decisions and the activities in the lower levels modules, control specifications can be implemented in the upper-level module structure. Data transformations specified in Data Flow Diagrams (DFDs) are allocated to modules using two techniques discussed below.

Transform-Oriented and Transaction-Oriented Designs

The first technique used in developing a top level SC from DFDs is called transform-oriented design. In this case the data transformation nodes (bubbles in the DFDs) can be divided into three groups. The first group of nodes are concerned with input and data preprocessing functions, the second group perform data processing functions, and the third group are concerned with output related functions. Three modules can then be defined in the Sc, an input module, a transform module, and an output module. These module are called from a main module as shown in Figure 4.1. The three modules can then be refined further into sub-modules.

The second technique is called transaction-oriented design as shown in Figure 4.2. In this case the data transformation is transaction driven, that is, depending on the type of the input transaction, an operation is triggered. In this case a module called get_transaction, and a module called dispatcher can be defined and then refined further into subordinate modules. The dispatcher module checks the transaction type and identifies the required modules needed to be activated. An example of this approach is the system function dealing with operator commands in a real-time system. Each command is treated as a transaction. Module get_transaction contains the operator interface which displays to the user a commands list to choose from and reads in the commands. The dispatcher module selects or invokes the proper segment of module which process the command.

Structured Chart Refinements

The above methods are used to obtain a first-cut design. The refinement process is then needed to obtain a good quality design. Structured chart refinement can be guided by the need for satisfying design criteria regarding the coupling of pairs of modules, and the cohesion, complexity, testability, and reusability of individual modules.

Coupling as mentioned above in the previous section is a measure of connection between two modules. This can take the form of call relationship, parameter passing, or shared data area. One of the important design criteria is to reduce coupling between modules. This because a good modular design lends itself to loosely coupled modules. Techniques for reducing coupling includes grouping data into data structures (using data abstraction), and eliminating control couples as much as possible passed from lower level modules to higher level modules. This is because ideally control information items should flow from higher level modules to lower level ones, and data informations flows from lower level modules to higher level ones. The main reason behind this is to make high level modules control the activities performed in lower level modules However, in many applications it is necessary sometimes to have control couples flowing upwards from lower level modules.

Cohesion is a measure of internal relatedness of the components of a module. High cohesive modules are sought out in the process of decomposition and refinement. Components of a module that or operate on shared data structures are more cohesive than modules whose components have merely a precedence relationship. The module name should also represent the functions performed by the components of the module. This is termed as external cohesion.

Complexity is another design criteria used in the process of decomposition and refinement. A module should be simple enough to be regarded as a single unit for purposes of verification and modification.

.

Examples on structured design can be found.in many text books on software engineering in the literature. Examples using an ICASE tool will be presented in section 4.4.

Design Approach for Real-Time systems (DARTS)

The DARTS approach for real time systems was developed as an extension to the structured design methodology [GOMAA 84]. It provides an approach for structuring the system into tasks and gives a mechanism for specifying the interfaces between tasks. Tasks are software modules which may run concurrently. Tasks are defined by allocating functions from the analysis diagrams. Function allocation to tasks depend on functional cohesion where a set of cohesive function are allocated to a given task, time-criticality where a time critical function is allocated to a high-priority task. A task may also be defined to control the execution of other tasks, for

example, a state transition meager task can be denied to implement the process of activating other tasks and generating control signals according to a STD specified in the analysis.

The interfaces between tasks are defined by means of task communication modules (TCMs), task synchronization modules (TSMs), and information hiding modules (IHMs). The TCMs comsist of loosely coupled modules where a message queue is setup and maintained between two communicating modules, and tightly coupled modules where a protocol of send-and-wait-for reply and receive-and-reply is implemented. The IHMs consist of a common data store and access methods used by communicating tasks to gain access to the data store. Finally TSMs are supervisory modules that manage control and coordination signals (e.g., timer interrupt) between communicationg tasks.

The DARTS design is specified using a task diagram where tasks are represented by properly labeled parallelograms. Communications between tasks are represented by links. Special symbols on the links are used to distinguish between the different types of interfaces between tasks. The DARTS method was extended recently and a tool was developed to support the design methodology. However, no commercial ICASE supporting this method tool is yet available. For this reason the detailed steps of this method will not be discussed further..

4.3.2 Object-Oriented Design

Object-Oriented Analysis specify the system under development as a set of communicating objects. An object is an instantiation of a class of objects which provides the support for data abstraction an information hiding. The class provides the template for objects by specifying the data structures and functions needed for specifying the design of the physical structure and dynamic behavior of objects. The most important part of Object-Oriented Design is to specify the design of each class of objects in terms of its data structures and member functions (the functions which operate on the data structures). It is also necessary to specify and to design the class hierarchy, and the protocols of communications between objects. Object-oriented design builds upon the software design concepts of abstraction, information hiding and modularity described in section 4.1. OOD simplifies the design process aimed at achieving the above design criteria. The resulting design specification can be easily implemented in an object-oriented programming language such as C++ or Ada.

In the following paragraphs, a design methodology based on the Shlaer-Mellor analysis and design notation is briefly introduced. A more detailed description of the method as well as other more recent methods based on the Booch notation are described in section 4.5.

To specify an Object-Oriented design, four types of diagrams are developed based on the OOA diagrams developed during the analysis phase. These diagrams are described as follows:

1. Class Diagrams: A diagram for each class to show class characteristics (e.g., data structures and member functions) and interface characteristics. The diagram specifies the names and

characteristics of all public and protected elements of the class. A function or a data structure is a public element when it can be accessed by functions of other classes or non-class functions, a protected element is one which can only be accessed by member functions of the class or its derived classes. Input/output couples for the public member functions are also specified. Figure ?? in section 4.5.1 shows an example of a class diagram.

2. Class Structure Charts (SCs): A class SC shows the architecture of class and its functions. The SC shows the internal structure of a single class by showing the structure of the modules and the data and control flow within the class. This SC is obtained from the State Transition Diagram (STD) and the corresponding Action Data Flow Diagrams specified for the class during the OOA phase. Figure ?? in section 4.5.1 shows an example of SC for a class.

3. Dependency Diagram: A dependency diagram specifies the usage and dependencies between classes and non-class functions or sub-programs. The diagram shows the client-server relationship between classes and between classes and non-class functions.

4. Inheritance Diagram: An inheritance diagram specifies the relationship between base classes and their derived classes.

The design starts with the development of the dependency diagram where the classes are defined and the dependencies in terms of how the classes will use each others functions are specified. The classes will be defined from the objects specified during analysis and additional classes can be defined to simplify the definition of complex classes. This can be done using abstraction (e.g., defining abstract base classes and other derived classes) and by using aggregation (e.g., describing a class as consisting of objects which are instantiations of other classes). The inheritance diagram is then developed to show the organization of the class hierarchy. Each class that participates in the inheritance diagram should be drawn with relevant functions and logical components.

Public and protected elements of a class are specified in the class diagram. This diagram is drawn for each class specified in the design. Private elements of the class appear only in the class structure Chart which specifies the coupling between the class functions, and the functional access to the class internal data structures.

The notation used in Object Team/OOD which is based on the Shlaer-Mellor notation for OOD was developed on top of Ada Structured Graphs to be described in the next chapter. This notation has been tailored to facilitate the generation of C++ code from design diagrams. The various symbols used in the notation are described later in this chapter in section 4.5.1.

Design Criteria for OOD-based Designs

The design criteria for OOD-based designs has been studied and published in the literature. Coud-Yourdon OOD guidelines [Coud-Yourdon 94] are based on extensive discussions and interviews with software engineers and practitioners throughout the world. These guidelines include the following:

1. Coupling guidelines: Coupling as defined in section 4.1 is the strength of interconnects between modules. Loose coupling is targeted for good design. In OOD-based designs, the concern is on coupling between independent classes of objects which are not part of inheritance or aggregation relationships. The criterion for good design in this case to minimize the number and the size of information items passed between independent classes.

2. Cohesion guidelines: In OO-based design cohesion should be observed at the following three levels:

a) Individual functions or methods within a class: The cohesiveness of a function within a class can be observed as in structured design by having the function carry out one operation or a set of operations whose purpose can be described accurately with a simple sentence containing a single verb and a single object.

b) Individual classes: The cohesiveness of individual classes is observed in the cohesiveness of the data and functions or methods encapsulated within a class. The components of a class should clearly support the purpose and intent of the class of objects, and should be easily traceable to the specifications of the object in the OOA phase.

c) Class hierarchy: The cohesiveness of an entire class hierarchy can be evaluated by examining the extent to which the derived classes override or delete attributes or functions inherited from their base classes.

3. Clarity of design: One of the important criteria of OOD-based designs is their simplicity and clarity. An OOD-based design should be easily understandable. Guidelines to achieve this include: using consistent vocabulary for naming functions and attributes; avoiding fuzzy class definitions; adhering to existing protocols or behaviors of classes.

4. Hierarchy and factoring: The number of levels in any hierarchy (inheritance or aggregation) should neither be too large nor too small.

5. Design Complexity: Keeping classes, message protocols between objects, and individual functions simple is an important guideline which help is the maintainability and reusability criteria. A bad design will have an excessive number of attributes and functions in a class. A complex message protocol is an indication of excessive coupling between classes. Complex class functions consisting of several pages of code specifications and contains many IF-THEN-ELSE

statements is an indication that procedural code is being used to make decisions that should have been made in the inhiritance hierarchy.

6. Design volatility: A bad design is observed when a small change in one class causes a ripple effect of changes throughout many classes. Design volatility may result from coupling problems or other causes, it can, however, be used as a clear unbiased assessment of the design quality.

7. Design verification: This criterion focuses on the ability of the design to be evaluated for verification purposes using specific scenarios. Reviewers should be able to play a scenario and act out the behavior of individual objects, pass messages or information items between each other and easily execute the whole scenario. The criterion also enables an analyst to quickly develop a dynamic simulation model for the design and verify the design behavior resulting from the simulation with the behavioral specification of the various objects developed in the OOA phase.

.3.2.2 REFERENCES

[BOOCH 90] Object-Oriented Design with Applications, Grady Booch, Benjamin/Cummings, 1990.

[GOMAA 84] Gomma, H., “A Software Design Method for Real-Time Systems,” Communication of the ACM, Vol. 27, No. 9, September 1984, pp 938-949.

2.2.1.1 HOMEWORK

The following is a simple assignment designed to introduce the student to the CASE tool and structured analysis process described above.

-- Use Teamwork SA and RT to analyze the requirements and obtain the specifications of a vending machine which has the following requirements. The system is to do the following:

* Accept objects from the customer in payment for their purchase.

* Check each object, using the object information (such as size, weight, thickness, and edges) to make sure it is not a slug.

* Accept only nickels, dimes, and quarters, and treat any other object as a slug.

* Initiate payment computation or product selection only after a valid coin is detected.

* Accept product selection from the customer, check to see if product is available, if not return coins and notify customer.

* Products variety can change from time to time, hence prices should be changeable.

* Return back payment if customers decides not to make a selection.

* Dispense the product to the customer if all conditions are satisfied, i.e., if product is available and amount is sufficient.

* Return the correct change to the customer.

* Make deposited coin available for change.

HOMEWORK

1. Use Teamwork/SD to develop a design of the vending machine problem. Give detailed specifications of the all the modules in your design.

2. The use of the concepts of coupling and cohesion to provide guidelines for creating good designs are shown in [NIELSEN & SHUMATE] (pp 137-149) in terms of Ada packages and tasks. Use such concepts in developing an alternate design of the robot controller example shown in Chapter 22 (pp210) of the same reference.

HOMEWORK

* Read chapter 12 in the reference (see [BOOCH 90], pp 444-470) which describes the analysis and design of a traffic management system and its implementation in Ada. Describe the differences between the analysis produced using teamwork/OOA, and the analysis used in this chapter. Comment on the difference between the design method using Object Team/OOD and the Booch design method 2

2.2.1.2 REFERENCES

[Card 86] D. N. Card, V. Church, and W. Agresti “An Empirical Study of Software Design Practices,” IEEE Trans. On Soft. Engr., Vol SE-12, No. 2, 1986, pp264-278.

[WARD&MILLER 85] Structured development for real-time systems, P. Ward and S. Mellor, Yourdon Press, 1985.

[HATLEY&PIRBHAI 88] Strategies for real-time system specification, D. J. Hatley and I. A. Pirbhai, Dorset House, 1988.

[NIELSEN&SHUMATE 88] Designing Large Real-Time Systems With Ada, K. Nielsen and K. Shumate, McGraw-Hill, 1988.

[SOM] Software Engineering, Fourth Edition, I. Sommerville, Addison Wesley, 1992.

Chapter 4

4.1 The Software Design Document

4.2 Software Design Criteria

4.3 Software Design Methodologies

*****>4.4 Structured Design Using ICASE

*****>4.5 Object-Oriented Design Using ICASE

4.4 Structured Design Using ICASE

As mentioned in section 4.3.1, structured design using ICASE is centered around the development of structure charts and M-specs to specify the architectural as well as the detailed design of software components. In this section, we first give a description of the notation used in Teamwork/SD and then examples using this notations are presented.

4.4.1 Structured Design Notation in Teamwork/SD

The notations supported by Teamwork/SD for specifying structured designs using SCs and M-specs are described in this section. The section describes the SC notation first then M-specs notations are briefly summarized.

The SC notation consists of eight components. These are modules, invocations, couples, connectors, transaction centers, iteration symbols, text blocks, and labels. These components are described in the following paragraphs using as an example the Aircraft Monitoring System (AMS) described in previous chapters. Figure 4.3 below shows the main SC of the AMS design.
Modules

Modules represent software units which are independently callable as described in the previous sections. Modules are represented by rectangles in the SC. Examples of the top-level modules shown in the AMS SC are AMS_Main, Initialize_System, and Monitor_System. Several special types of modules can also be specified as follows:

1. Library modules: These modules are represented graphically by rectangles with double vertical lines at the left and write edges as shown in Figure 4.4 They specify reusable components already defined in a reuse library. These modules do not require an M-spec to specify their internal design. Module Get_Time_of_Day in the AMS SC is an example of a library module.

2. Data-hiding modules: These modules are represented graphically by rectangles with a horizontal line separating their upper and lower parts. The lower part of the rectangle specify the data structure hidden in this module. The upper part is further divided into vertical partitions which represent submodules or functions in the data-hiding module. These functions can be invoked by other modules and are the only ones that have access to the internal data structure of the module. There are several data hiding modules in the AMS Sc example. The sensor_data module consists of two submodules Put_sensor_Data, and Get_sensor_Data, as well as a data structure called the sensor_data_buffer. Higher level modules must call one of the submodules to access the sensor_data_buffer. The fuel_data module, temperature_data module, the pressure_data module, etc., consist of one submodule and a data structure. It is assumed that external sensor drivers (not shown in the SC) will update these data structures with sensor readings.

3. Data-only modules: These are represented graphically by elongated hexagons. They represent global data that can be directly accessed by other modules. Three examples of data-only modules are shown in the AMS SC, these are range_constants, dial_buffer, and recording_buffer. These global data structures are visible or accessible to other modules in contrast with local data structures defined within modules and used in data or control couples (as discussed below), and data structures in data_hiding modules which are only accessible to submodules within their own module.

4.Macro modules: These are represented graphically by rectangles made up of dashed lines. A macro module specifies an in-line unit of code processed by a language preprocessor. Using macros is faster than using regular modules.

Invocations

Invocations are represented graphically by a solid directed line starting from the calling module and terminating at the called module. In the AMS example, an invocation is shown starting at AMS_Main, and terminating at Initialize_System. It indicates that the source module contains a call or an invocation for the destination module. The special types of invocations are as follows:

1. A dashed directed line invocation represents an asynchronous invocation in which the calling module continues processing without waiting for the called module to complete. The solid line invocation (the usual invocation) is synchronous in the sense that the calling module waits for the called module to complete and return control back to it. The asynchronous invocation, on the other hand, creates two threads of control which will be executing concurrently.

2. An invocation terminating at the called module with a hat instead of an arrow head represents an in-line invocation in which the called module is actually is a part of the calling module (textual inclusion, i.e., the code in the module being called actually exists in the calling module). This helps in reducing the complexity of the chart in cases when many invocation lines have to come out of the same module (high fan-out). In these cases using a hat type invocations the module code is split into two rectangles across which incoming/outgoing invocation lines can be divided. The invocation to the module Update_sensor_Data in the AMS SC is an example of a hat invocation.

Couples

Couples are used to represented information passes between modules connected with an invocation line. They are represented graphically by arrows with unfilled (empty), filled, or partially filled circles at their tails. These types of couples are described as follows:

1. Data couples (data information items)) represented by arrows with unfilled circles. The sensor_id data couple shown on the AMS SC is associated with invocations from Monitor_System module to its lower-level modules Determine_Fuel_Capacity, Determine_Sensor_Range.

2. Control couples (control information items) represented by arrows with filled circles. The couples temperature_data_received, pressure_data_received, fuel_data_received, and smoke_detection, are examples of control couples associated with the invocation from Monitor_system to Poll_sensor in the AMS SC.

4. Hybrid couples (both data and control information items) are represented by arrows with partially filled circles. These items are couples which are used for data processing as well as control flow constructs in receiving modules.

5. Bidirectional couples (contain two arrows) pointing to both the calling modules and the called modules represent information which are input/output to/from the called modules.

Connectors

Connectors are very important in specifying the design of complex systems. They are used to reduce the complexity of the diagrams by allowing the SC to be divided into several SC sheets.

Connectors also allow a module in a SC sheet to call another module in a different part of the same SC sheet or a module in another SC sheet. The types of connectors are described as follows:

1. On-sheet connectors are represented by circles. The connector should contain a label specifying the name of the module being connected.

2. Off-sheet connectors are represented by pentagons (in the shape of a home plate). The connector should contain a text specifying the name of the SC sheet and the name of the module in that sheet to which it is being connected.

3. Defining connectors are those used to call a module.

4. Referencing connectors are those called by a module.

Transaction centers

A transaction center is used to specify that invocation calls made by a module are conditional. This is represented by a diamond shaped symbol attached to the bottom edge of a module (since the bottom edge is where the invocations are define). In this case only a subset or non of the defined invocations can occur depending in the control flow specifications contained in the M-specs of the given module. For example, in module Monitor_System in the invocations to Determine_Fuel_Capacity and to Determine_Sensor_Range are conditioned upon the values of the couples returned from Poll_sensor.

Iteration symbols

Iteration symbols are used to specify that a set of invocations are enclosed within an iterative loop in a module. An iterative symbol is represented by a triangle based at the bottom edge of the module. A line of the triangle starts at the bottom edge of a module and the other returns back to the bottom edge encompassing the invocation calls contained in the loop. For example, module Monitor_System contains two loops represented by two triangles at its bottom edge. The inner loop encompasses the invocation calls to Determine_Sensor_Range and Generate Alarm, and the outer loop contains all the calls made by monitor sensor to its subordinate modules.

Text Blocks and Labels

Text blocks allowed in a SC sheet are two types as follows:

1. Bounded text blocks are used to provide names and labels for modules, individual couples, and connectors. These text blocks should be bound to the corresponding object (inside a module or connector and beside a couple),

2. Unbounded text blocks are used freely in the diagram to provide comments on the different parts of the SC sheet.

Each non-library module specified in each SC sheet must have a corresponding M-specs sheet to specify its internal details. The notations for M-specs is described below.

An M-spec consists of five sections as follows:

1. Title section

2. Parameters section

3. Locals section

4. Global section

5. Body section

The Title section contains the module name as specified in the SC. The Parameters section contains the names of input/output data and control couples specified on the SC on the invocation lines terminating at the module. The Locals section contains the names of local information items used in the M-spec body section and. The Global section contains the names of data-only modules invoked or accessed by this module. The body section contains the pseudo code specifying the algorithm design for the module functions. Free text may be used initially to describe the functions of the module at the architectural design step.

Although the Parameters and Global sections inherit information from the SC, the designer can still edit the information in the M-spec fields using the Parameters, Locals, or Global menu. Correct entries according to the established syntax will show up in bold, otherwise the font becomes halftone. This provides an immediate feedback to the designer on incorrect entries. The correct syntax for specifying information items in the Parameters section is specified as:

DDE_name: type_direction

where,

DDE_name = any text that is a valid Data Dictionary Entry name (every couple and every data-only module must have an entry or a DDE in the Data Dictionary established during the analysis phase and refined further by adding more entries during the design phase),

Moreover, type = [data|control|data+control]

direction = [in|out|inout]

For example the following is a valid syntax: sensor_id: data+control_inout, specifies the item sensor_id as a bi-directional hybrid couple.

4.4.2 Steps for developing structure charts

The development of structure Charts to specify the software design of a CSCI from the requirements analysis and specifications diagrams is not a straight forward task. This is due to the inherit complexity of the software design process and the lack of easily applicable mathematical models and tools for verifying the design correctness and quality.

In this section a set of steps are described to guide the designer in developing an architectural design which conforms to the design criteria described in section 4.2. The design steps described here were obtained by merging the design steps described in [Shumate-Keller 92] for real-time structured design with those described by Pressman in [Pressman 93]. These steps are described as follows:

Step 1- Review and refine the diagrams developed in the analysis phase. The analysis diagrams contained in the Software Requirements Specification document are reviewed and refined for the design phase to include greater detail. The ICASE tool help in this process tremendously since the navigation between the various levels and the refinement of specification details is made much easier. A refined specification contains a more flattened view of the logical model of the system. This results from the process of bringing lower level functions to upper level DFDs (the functional decomposition process in the analysis phase may often be inconsistent with design where important functions from the design or the physical model view point may have been buried in low-level DFDs in the specifications or in the logical model). The refinement done in this step does not change or contradicts the original functional specifications.

Step 2- Identify and label the necessary concurrent modules from the refined analysis diagrams. These modules will be invoked when the software is activated and will contain several functions

specified in the refined analysis diagrams. The phrase necessary concurrent modules here means that these modules must be running concurrently for the correct real-time operation of this system. If the identified functions in the various modules can be sequential (i.e., invoked in a sequential order) and still satisfy the timing specifications for the output events then there is no need for concurrency. Concurrent design is usually much more difficult to build and verify than sequential designs. Therefore, concurrency must be absolutely necessary in order for the system to function according to the specifications.

Step 3- Implement using asynchronous/synchronous invocations in a structure chart the invocation of concurrent and sequential modules from the main or the root module (this module usually carries the name of the software under development).

Step 4- Determine whether the refined DFD/CFD diagrams for the concurrent modules identified in step 2 have transform or transaction flow characteristics as described in section 4.3.1. Determine the first level factoring of these modules, and document their specifications using M-specs. Specify the couples using data dictionary entries. The transform center in a transform-centered flow diagram is identified by determining the incoming and outgoing flow boundaries. In a transaction-centered design, map the appropriate functions to the transaction input module and the dispatcher subordinate modules.

Step 5- Refine the first-cut design obtained above to reflect design criteria such as coupling, cohesion, and information hiding.

Step 6- The complex modules specified in the previous steps should be factored out using the steps 1 through 5 above and the process should continue until all lower level modules are simple enough to specify using simple M-specs.

The next section presents an example of applying the above procedure for software design.

4.4.3 The Aircraft monitoring System (AMS) Design

Design Overview

Figures 4.3 through 4.8 gives the SCs and M-specs for the major modules of the AMS system design. This design is developed according to the AMS system analysis described in section 3.2.3.3. Recall the top level DFD/CFD diagrams shown in the AMS analysis. Following the procedure described above, in the first step a refined DFD 0 is obtained. This refined DFD differs from the one shown in the previous Chapter by adding more details to the function Monitor sensor. The important subfunctions of Determine Range and Determine Fuel Capacity specified in DFD 1 should be moved up to DFD 0, and the original function Monitor Sensor now produces the

necessary flow to the sensor_reading store, all other functions remain unchanged. It should be emphasized that this step is made simple with the availability of an ICASE tool, where changes can be made and the refined DFDs can be automatically rechecked for consistency with all other levels.

Following the refinement step, the second step of the design procedure identifies concurrent modules. For the AMS design whose main SC is shown in Figure 4.3, two concurrent modules are identified. These modules are Monitor_System, and Process_Pilot_Request as shown in the diagram. The first module takes care of all the functions in DFD 0 except the Process_Pilot_Request function which is allocated to the second module. The AMS_Main module in the SC makes an asynchronous call to Monitor_System after calling the Initialize_System module is completed. Since the asynchronous call returns control to AMS_Main immediately after the called module is activated, the Process_Pilot_Request module (located in another SC sheet shown in Figure 4.5 as indicated by the external connector symbol) is called. Both of these modules will be concurrently active all the time.

In step 4, each concurrent module identified above is factored out further. The flow in the functions (in the DFDs) allocated to each module is determined to be either transform-oriented or transaction-oriented. In case of the Monitor_System module, the flow (as shown in the functions specified in the refined DFD 0) is transform-oriented. The incoming flow segment consists of the functions Monitor_Sensor and Receive_Smoke_Detection_Signal. The transform segment consists of the functions Determine_Range and Determine_Fuel_Capacity. And finally, the output segment consists of the functions Generate_Alarm, Record_Aircraft_Data, and Generate_Dial_Reading. The incoming flow segment is implemented by the module Poll_Sensor, the transform segment is divided into two modules Determine_Fuel Capacity and Determine_sensor_range, and the outgoing flow segment is divided into the three modules Generate_Alarm, Generate_Dial_Reading, and Record_Aircraft_Data. The Generate_Alarm module is described in another SC sheet (Figure 4.7), where it is further factored out.

The Process_Pilot_Request module described in a separate SC sheet in Figure 4.5 is factored by considering the DFD 2 describing the subfunctions specified to process pilot request. The flow type in this case is transaction-driven since a separate subfunction is activated based on the request made by the pilot.

In step 5 of the design procedure, the first-cut design is refined using the design criteria described in section 4.2. These criteria were already used to get the design shown in the AMS main SC. The reason that the modules in the transform part (i.e., Determine_Fuel Capacity and Determine_sensor_range) were separated in two modules rather than one is because of the cohesion criterion (only procedural type cohesion might exist between these two modules which is less stronger than functional cohesion). Similarly, the output modules were separated into three different modules for the same reason mentioned above. Other design criteria such as coupling and information hiding would play a role in specifying how the sensor data read by Poll_Sensor module are transferred to other modules. In this case the data hiding module is used to hide the data structure sensor_data_buffer. This module contain two submodules Put_sensor_Data used to

write sensor data into the buffer (by Poll_sensor), and Get_sensor_Data used by other modules (in the transform segment, and the outgoing flow segment) to read the sensor data.

In step 6, the complex modules identified are factored out further using the same steps outlined above. These modules are Poll_sensor, Generate_Alarm, and Process_Pilot_Request. The Poll_sensor module as shown in the main SC Figure 4.3 is factored out to four data hiding modules one for each type of sensor, and an update_sensor_data module. Notice that the later module is invoked conditioned on the value of the read_status control couple returned by the data hiding modules reading the sensor data buffers. The invocation used is a hat type invocation which indicates that this module (Update_sensor_Data) is part of the Poll_sensor module. The reason for such an invocation is to clearly show that the conditional function of updating the sensor data buffer is part of Poll_sensor (having the Update_sensor_Data module calling Get_Time_of_Day, and Put_sensor_Data made the diagram more readable and easier to follow). The M-specs for modules Monitor_System and Process_Pilot_Request are shown in Figure4.4 and Figure 4.6, respectively. The SC sheet of Generate_Alarm is also shown together with its M-specs sheet in Figure 4.7 and Figure 4.8, respectively.

4.5 Object-Oriented Design (OOD)

This section discusses the OOD methodology which dependent on the specification produced in the analysis phase. The transition between OOA and OOD is much smoother in the object-oriented approach as compared to structured analysis and design. The same objects identified in OOA together with their dynamic and functional behavior are mapped into software components during OOD. Object classes are defined or their specification are refined to provide templates for software objects. Class hierarchies are refined. The dependencies and interfaces between objects are specified. Algorithms are designed to implements operations defined on objects.

The basic procedure for OOD has been introduced in section 4.3.2. Since OOD takes us from the problem logical model domain of OOA to the solution physical model range, it is important to understand the fundamental concepts of object-oriented programming. The fundamental concepts for object-oriented design and programming are discussed below.

The concepts of encapsulation, inheritance and polymorphism are essential and fundamental in OOD. These concepts are described as follows:

1. Encapsulation: encapsulation refers to the process of hiding the structure and behavior of the elements of an abstraction (or a class of objects). It serves to separate the interface of the abstraction from the internal implementation. This is clear from examples where a class of objects have public elements, private elements, and protected elements defined as follows:

a) Public elements: the public elements of a class of objects are those attributes or functions of the class which can be invoked, accessed, or modified by any other object in the system. These elements of an object provide the outside interfaces to other objects.

b) Private elements: The private elements are those attributes and functions of a class totally hidden from the outside world. These elements can only be accessed, invoked or modified by functions defined within the same class.

c) Protected elements: These are the attributes and functions of a class that are hidden from all other classes of objects except those classes that are derived from this class (i.e., act as subclasses using the inheritance relationship).

2. Inheritance: Inheritance defines a relationship wherein one class (termed as a subclass or a derived class) shares the structure or behavior defined in one or more classes (termed as supper classes or base classes). Inheritance represents a hierarchy of abstractions in which a subclass may augment or redefine the inherited structure and behavior of its base classes. In OOA inheritance was represented by an “is a “(also called supertype/subtype) relationship. In OOD inheritance is refined and a class hierarchical structure is designed, where data structure components and functional elements of each class in the hierarchy are specified. As mentioned in the previous paragraphs, private elements of a super class can not be accessed by the elements of a subclass, whereas protected elements of super classes can be accessed by the elements of its subclasses. Also as mentioned in this paragraph, any element in the super class can be redefined (or redesigned) in the subclass.

3. Polymorphism: Polymorphism is one of the most powerful features of object-oriented programming. It exists when a single name (such as a variable declaration, or a function name) may denote objects of many different classes that are related by a common base class. This variable or function is inherited from the base class and redefined in the subclasses. In this case dynamic binding occurs where the particular binding of an access or invocation of this function or variable to a particular object instance is formed during run-time. Polymorphism is the opposite of monomorphism which exists in all programming languages that are strongly typed and allow only static binding (i.e., binding of names to types or objects at compilation time) such as Ada. The concept of polymorphism is therefore tied to the concepts of inheritance and dynamic binding.

In the next subsection, we give a description of the notation used in Object Team/OOD usingthe ATM example obtained from the Samples directory of Teamwork.. The modeling methodology of Rational Rose/C++ is then described in a following subsection. This methodology is based on the Booch notation discussed briefly in section 3.3.2. Finaly in the last subsection of the this Chapter, the analysis and design methodology of the Object Modeling Technique (OMT) is briefly described. These methods represent the most widely used and tool supported techniques in object-oriented developement.

4.5.1 The Object Team/OOD notation

The design procedure for OOD using Object Team/OOD is centered around the development of the following diagrams: a class dependency diagram; a class inheritance diagram; class diagrams; and class structure charts. These diagrams are needed to specify the architectural as well as the detailed design of software components.

The notation supported by Object Team/OOD is a variation of the notation suggested by Shlaer and Mellor called OODLE. The objectives of this notation are stated in [Shlaer-Mellor 92] as follows:

1. Representation of fundamental concepts of OOD: The notation proposed must represent the fundamental concepts of OOD such as encapsulation, inheritance, and polymorphism in a simple intuitive manner.

2. Support for data typing: Data typing should be strongly emphasized and represented by the notation, regardless of the level of support for typing in the intended implementation language (e.g., Smalltalk is an untyped object-oriented programming language).

3. Support for key design decisions and low level language-independent concepts: the notation should be rich enough to support key design decisions which must be well documented to render the design into an implementation language. At the same time the notation must be able to represent low level language independent concepts such as visibility of operations and data structures, partitioning of code, invocations and exceptions.

4. Multiple views: The notation should not seek to represent every aspect of the design in a single diagram. Notations based on a single view of the design are not likely to represent all aspects of the design equally well. For example the notation for Ada structured graphs (described in Chapter 5) is extremely effective in representing visibility of components of an object but is less effective in depicting flow of control. In a multiple views design notation, consistence rules must be specified in order to determine if the multiple views of the design are consistent.

6. The notation should be easy to learn and easy to draw, and should not be needlessly different from notations presently published or supported by CASE tools. This objective is reflected on the fact that the notation for OOA and OOD suggested by Shlaer and Mellor is simpler than other notations proposed and easier to learn in a transition from the traditional structured analysis and design notations.

7. The notation should be susceptible to language-specific interpretations. The four components of the OODLE notation are described briefly as follows:

- Dependency Diagram (DPD): A DPD depicts the client-server (i.e., invocation) and friend relationships that hold between classes.

- Inheritance Diagram (ID): An ID shows the hierarchical structure that describes the inheritance relationship between classes.

- Class Diagram (CD): A CD depicts the external view of a single class of objects. It is based on notations developed earlier by Booch and Buhr, but with considerably much more detail of the external interface specifications.

- Class Structure Chart (CSC): A CSC is used to represent the internal structure of the of the class in terms of data structures and modules or functions.

The above diagrams are described in more details in the following paragraphs using the notation supported by Object Team/OOD. This notation is language dependent in difference with the original notation of OODLE. The notation is used by teamwork to generate C++ code as shown in Chapter 5. The difference between OODLE and Object Team/OOD notations include the ways of showing and depicting classes, protected data and functions, inheritance relations, friend relations, and parameters.

Dependency Diagram

A Dependency diagram consists of three main types of components. These are classes represented by rectangles, nonclass functions represented by a rectangle with an extended upper side bar, and dependency invocations represented as directed arcs with solid or dotted lines. These arcs are used to represent dependencies or relationships between classes or between classes and nonclass functions. The solid lines represent client-server relationships where the source invokes services or methods contained in the destination. The dotted lines refer to friend relationships where the private data and member functions of one class are used by another class. Friend relationships can also be defined between classes and nonclass functions.

Nonclass functions represent software components which does not belong to any class of objects. A nonclass function with friend relationship dependency on a given class can have access to all the private data and member functions of the class.

Figure 4.9 shows a portion of the dependecy diagram for the ATM example presented in the previous Chapter. The Figure shows classes Withdrawal, Deposit, and Inquirey all have a client server dependency on classes Savings_Account, and Checking_Account. A Withdrawal type object needs to invoke an operation in either a Savings_Account type object or a Checking_Account type object which in turn changes its state. The Transfer type object is composed of a Withdrawal object and a Deposit object, and hence it invokes operations on both of

these objects. The figure shows also that a Checking_Account object can make adjustments (by invoking the proper operation) on the state of a Savings_Account object. A friend function (not shown in the Figure) Save_All_Objects acn be defined whose purpose is to save periodically the current state of active objects in the system so that a recovery procedure can be implemented in case a system shutdown or failure. This function should be represented by a rectangle with an extended upper-side bar and should have dotted-lined arec to all classes of objects whose state should be periodically saved.

Inheritance Diagram (ID)

The hierarchical structure that describes the inheritance relationship between classes is specified in an ID. This diagram is extremely important in object-oriented development. Other than implementing the subtype-supertype class relationships specified in OOA, it also establishes the design or reuse of reusable software components based on abstraction. A well structured ID results in simplifying the design of individual classes. This comes about by developing abstract classes which defines the important data and functions inherited by a number of derived classes. This facilitates the design of a class library suitable for software reuse. The set of derived classes of a certain base class specify a cluster of system components which can be traced to a corresponding set of objects specified from the requirements in OOA.

The ID gives a view of visible data and functions for each class in the inheritance hierarchy and clearly shows the type of inheritance relationship between classes. There are two types of inheritance, namely, public inheritance and private inheritance. In public inheritance, public and protected data and functions of the base class are inherited as public in the derived class. Whereas in a private inheritance, public and protected members of the base class become private in the derived class.

The notion of public, protected and private members of a class mentioned in the above paragraph is related to the C++ rules of visibility of the class data and member functions to other components (these components are either functions of other classes or non-class functions). A member of a class can be either private, protected, or public. A private member (data or function) is not visible from any other component. Only the in-class members can access private class members. Protected members of a base class are visible to members of its derived classes. Public members of a class, as the name implies, are visible to all other components.

Each class in the ID is represented by a large rectangle containing symbols for public and protected members as shown in Figure 4.10 below. Class member functions are drawn as rectangles, and data structures are drawn as hexagon icons. Directed arcs with bulky arrow heads are drawn from a base class to each of its derived classes. Deferring functions (i.e., pure virtual functions in C++) are represented by dotted rectangles with a dotted inside vertical bar. These functions refer to functions which are only defined in the base class but implemented in the derived classes of the base class. The same named functions are shown in the derived classes with a solid rectangle with a solid inside vertical bar.

Figure 4.10 shows the ID of the ATM example. Class account (specified as an abstract class, i.e., no objects will be instantiated for this class) is defined to specify a higher level of abstraction for classes Checking_Account and Savings_Account where the public virtual function get_balance is specified. The abstract class Active_Instance is defined to provide a specification for the attributes of the active instance of any object in the system. It is therefore defiend as the top class in the ID since all the other top classes in the system, such as ATM_Machine and ATM_Session, are derived from it. The detailed compoenents of this class will be discussed below when Class Diagrams are discussed. The classes representing the types of transactions in the system such as Validate_PIN and INQUIRE, are also derived from class Transaction, which should have also been designated as an abstract class.

The ID can be drawn on several diagrams. This is useful when specifying the inheritance in a sub-tree of classes which have multiple inheritance relationships with different base classes. These base classes are also part of other inheritance hierarchies. In this case a foreign module is used (the term foreign here means that this does not represent any of the known classes to be specified) in each ID containing a base class to refer to another ID containing the subfamily of derived classes. This module will carry the name Inheritance_Diagram_identifier, where identifier would appear as the name of the subfamily ID.

Class Diagrams (CDS)

For each class specified in the above diagrams, a CD is developed. A CD depicts the logical components and the external interfaces of a single class of objects. A large rectangle is used with the class name specified at the top followed by the optional key word “abstract”. The key word abstract specifies that the class should be implemented as an abstract class (in C++ an abstract class is one which does not have objects constructed for it but is used as a base class to specify logical components for a set of derived classes, e.g., the class Vehicle is an abstract base class for class car, class tank, class truck, etc.). Abstract classes often contain deferring functions implemented as pure virtual functions in C++.

The private data of the class is specified by drawing hexagon shaped icons inside the class rectangle. In OODLE terminology these data icons are called “works”. Each icon contains the name of a private data structure. Following the name of the data, a type specification is given separated from the name by a colon. The type specification could be a fundamental data type in C++ (e.g., int, float, etc.), a predefined class name, or as an “unknown” type.

Functions of the class appear as small rectangles inside the class rectangle. Only public or protected functions of a class can appear in the class diagram. Private class functions are specified in the Class Structure Chart. Protected functions are distinguished by drawing an X in the left side of the function rectangle. Following the concept of polymorphism static binding and dynamic binding functions can also be specified. Dynamic binding functions specify run-time binding (i.e., the function invocation and the function code are bound at run-time, whereas the usual static binding is done at compile-time) which supports what is termed in OO terminology as subtyping

inclusion polymorphism. Dynamic function are distinguished by drawing a inside vertical bar near the left side of the function rectangle. Virtual functions are specified by a dotted rectangle rather than a solid one as mentioned before. Eight different types of functions can be specified depending whether the function is virtual or nonvirtual, protected or public, dynamic binding or static binding.

Function parameters are specified in the diagram for each function requiring a parameter list. An invocation line pointing to the function is drawn for each function shown in the diagram (recall that only public and protected functions are shown in the class diagram). Input and output couples specifying the names and types of arguments in the parameter list are added over the invocation line for the function. An arrow head is used to specify whether the couple is an input couple, an output couple, or both input and output (in this case two arrow heads are drawn pointing to and away from the function).

The text specifying the couple follows the following format, name:(const) type (&) (=default_value). Where name is the name of the argument, the optional key word const is used if the argument value will not change, type refer to the data type, the optional ampersand signifies that the parameter is passed as a reference, and the optional default_value is used if no value is supplied by the calling function.

Functions raising or handling exceptions can be specified in the class diagram by specifying the following text in the function rectangle, function_name => (or <=) exception_name. Where the symbol => is used for a function raising an exception, <= is used for a function handling an exception, and the exception_name specifies the event name causing the exception.

Figure 4.11 shows the CD of class Active_Instance shown in the ID discussed above. The class consists of a private integer type data called current_state, and a public function called do_event. The function parameter listis specified on the invocation line poiting to the function. This function is used to uptdate the current state of the finite state machine (FSM) representing the dynamic behavior of an object. The details on the function body specification as well as the acess of data current state will be described when the class strucutre chart (CSC) for class Active_Instance is decribed below. Figure 4.12 shows the CD of class Inquire. which is derived from class Transaction as shown in the above ID. The class contains two public functions and two private data strucutres. The data strucutre account_typewill hold the information on which the inquiry transaction will be processed. The data structure my_fsm holds the information on the FSM which implements the specified behavior (specified during analysis) of the objects of this class. The FSM type is defined in another class whose CD is shown in Figure 4.13. Details on the design of these classes will be discussed below when their CSCs are discussed.. Finally, Figure 4.14 shows the CD of class Session which implements the analysis specified for this class whose state diagram is given in Figure 3.42

Class Structure Chart (CSC)

The class structure chart shows the detailed design of a class represented by a class diagram. Public and protected functions specified in the class diagram appear in the SCS. Private functions which can not be specified in any other diagram appear in the SCS. Functions that belong to other classes and non-class functions used by the depicted class appear as foreign modules in the SCS. They are represented by a structure chart off-sheet connector. If the foreign module is a class

function it should have the name of the class and the name of the function separated by a dot. If the function is invoked by a polymorphic invocation (in which case the class of the function is determined only at run-time), a question mark "?" is placed as the class name. Invocations are used to show the coupling between the class functions, foreign function, and class data structures. The notation used for the CSC follows in general the structure chart notation described in previous sections.

Figure 4.15 shos the CSC of class Active_Instance. The public function do_event, specified in the class diagram of Figure 4.11, invokes in sequence the private function get_current_state, then the function traverse of the FSM object, and finaly the private function post_new_state. The get_current_state function accesses the private data instance_data (which contains the current state of the object) and returns this to do_event. The function traverse of FSM is then used to get the new_state from the current_state and the event causing the change. The post_new_state function updates the instance data by making the new_state to become the current_state. The functions are inherited by all objects in the sysytem since the Active_instance class is the top level class in the class hierachy specified in the ID diagram in Figure 4.10. Every object also has its own FSM, or finite state machine type object, which implements the dynamics behavior of the object. The ivocation lines to functions get_current_state and post_new_state comming from the represent possible invocations from perhaps friend functions of this class.

Figure 4.16 shows another example of an CSC, the CSC of class Inquire specified in Figure 4.12. The public function take_event_iq1 is invoked whenever an inquiry_request event named iq1 occurs. The function first updates the active_instance of the Inquire object to reflect that an inquiry is activated and is in progress. Then it invokes the get_balance function of either a Savings_Account object or a Checking_Account object depending on the account_type_list. The function then updates the transaction log information of object ATM_Machine. The take_event_tr2 function of object Transaction is then invoked to process the end of an inquiery type transaction. Finally the private function take_event_iq2 is invoked which in turn invokes the ~Inuire function to terminate the current instance of the Inquire type object. (e.g., by releasing its class_data to the free store)..

4.5.2 OOD Using Rational Rose/C++

Rational Rose CASE tools provide an environment for OOA, OOD, code generation, reverse engineering, and round trip engineering capabilities. The tools include the following language dependent versions: Rose/C++, Rose/Ada, Rose/PowerBuilder, Rose/Smalltalk, and Rose/ SQLWindows. This section will summarize the OOD features supported in Rose/C++.

The Rose/C++ modeling techniques and notations are based on the Booch notation, and the OMT notation introduced in section 3.3.2. A demo version of Rose/C++ is available through the web address www.rational.com under the object technology category. The information presented in this section summarizes the tool documentation available through this web server. The Booch notation will be used in this section. The OOD model consists of Class Diagrams (CDs), Scenario Diagrams (SDs), STDs, Module Diagrams (MDs), and Process Diagrams (PDs). These diagrams are descibed as follows:

1. Class Diagrams: CDs consist of icons representing classes (drawn as dotted cloud shaped icons) and class categories (rectangular icons). Class categories are loosely coupled clusters of highly-related classes and class categories that are cohesive. This concept can be used to develop

a set of hierarchical CDs for the model. CDs also contain arcs between classes and class categories specifying relationships such as association, inheritance, and composition. The relation multiplicity is also specified on the arcs. Each class or a class category defined in the CD hierarchy may have an associated specification sheet containing its properties, attributes, and relationships.Figure 4.17 in the following pages shows an example of a class diagram. The example consists of seven classes. The Environmental Controller class, the SystemLog class, the Light class, and the Actuator class contains operations described in their icons. Arcs ending with a black circle represent composition or aggregation relationship. This translates to having the Environmental Controller type object to be composed of n Light type objects, a Heater type object, and a Cooler type object. Inheritance relationship is defined by directed arcs from the derived classes to the base class. This is shown in the Figure by having the classes Heater and Cooler to be derived from class Actuator. Finally the Actuator type object which provides the operations of startUp() and shutDown() uses objects of type Temperature and SystemLog. This relationship is shown by an arc with a white circle at one end.

2. Scenario Diagrams: A sequence of interactions between objects is specified by a scenario. Scenarios are needed to represent critical requirements, This is captured using an Object Message Diagram (OMD) or a Message Trace Diagram (MTD). Only one type of diagram is needed to be developed for each scenario, the other type can be automatically generated. The OMD consists of a set of objects instantiated from classes specified in a Class Diagram and arcs or links between them specifying messages passed. These messages are used to enable one object to invoke an operation of another object. Figure 4.18 shows an example of an OMD describing a scenario involving three objects (represented by solid-lined cloud shaped icons to distinguish them from classes). The object Temperature Controller is instantiaited from class Environmental controller specified in the Class Diagram shown in Figure 4.17. The object Air Conditioner Cooler is instantiated from class Actuator also shown in Figure 4.17. Figure4.18 shows a message sent from object Temperature Controller to invoke the startUp() operation of object Air Conditioner Cooler. A second message is sent from Temperature Controller to Object SystemLog to invoke its RecordEvent() operation. A third message is also sent from Air Conditioner Cooler to SysteLog to invoke the RecordEvent() operation after the former have successfully started up. A MTD corresponding to the above OMD can be automatically generated. This diagram is shown in Figure 4.19. The MTD can be used to change the order of messages, an operation which cannot be done on the OMD. Suppose that the designer decides that the Environmental Controller should not log its intention to start the Air Conditioner but rather its receipt of successful response to the startup message. The time order of messages can be changed by simple drag and drop operations on the MTD diagram.

3. State Transition Diagrams (STDs): An STD is developed to define the dynamic behavior of a given class. Figure4.20 shows the STD of the Environmental Controller Class defined in the Class Diagram described above. The STD consists of three states. The initial state, which is the Idle state, waits for a Define Climate input signal where a transition is made to state Daytime. At this state the Display operation of the SystemLog object is invoked and the adjustTemperature() operation of the Environmental Controller type object is also invoked when a Temperature drop or rise event occurs. A Sunset event will cause the operation off() of class Lights to be invoked and a transition to state Nighttime will take place where similar functions are invoked accordingly. The rest of the diagram is self explanatory.

4. Module Diagram (MD): A MD is developed to specify the physical design of the system architecture into modules. This contains similar information as a structure chart as described in

the structured design methodology. The MD is used to specify the allocation of classes (specified in Class Diagrams) and objects (specified in Senario Diagrams) to modules in the physical design of the system. A MD may contain icons representing entities such as a main program, specification modules, body modules, and subsystems as well as arcs representing dependencies between such entities. The main program icon represents a file that contains the root of a program (e.g., in C++ this would be a .cpp file containing function main()). The specification icons and body icons represent files containing the specification, and definition of classes (e.g., in C++ .h files are specification modules, while the .cpp files containing the class member functions definitions are body modules). A subsystem icon represents an aggregate containing modules and subsystems which is specified by a lower level MD. Therefore, subsystem icons can be used to represent a hierarchy of MDs defining the system architecture. To allocate a class to a module, the designer must first allocate the class category enclosing that class to a subsystem, then it is possible to allocate the class to any module within the subsystem. Figure 4.21 shows a MD consisting of subsystems and their dependencies. The MD of subsystem Climate_Control is shown in Figure 4.22. This MD consists of three modules where the actuator module is only a specification module, and the other two modules are shown with their specification and bodies grouped are together.

5. Process Diagram (PD): This diagram is used to specify the hardware architecture under which the software will run in terms of processors and devices (such as network devices, sensors, actuators, etc.). The PD also shows the allocation of processes to processors. Figure 4.23 shows a PD consisting of a processor called Gardner Workstation, and three devices consisting of hardware modules of sensors and actuators located in the three Greenhouses. The software consisting of the Climate_Control subsystem and other planning and user interface subsystems shown in the first MD above are allocated to the workstation. The devices consisting of three modules of sensors and actuators are distributed over the three green houses.

The above diagrams are augmented with specifications. Rose provides textual specifications of model components such as classes, relationships, or operations. Some of the textual specification of a model component can also be displayed inside the icon representing the component. For example in the Class diagram of the above example, class operations (such as startUp() and shutDown operations of class Actuator) are specified inside the class icon. In general each component in any diagram can have a specification sheet. For example in state diagrams every state and every state transition can have a specification sheet attached to it. Rose/C++ provides semantic checking facility to help build consistent models. The facility provides support for prevention of circular inheritance, detection of class access violation, detection of inconsistencies between class and objects, and detection of inconsistencies between operations and messages. The check facility Code generation in C++ code is provided by the tool. The developer controls the code generated by manipulating specifications and code generation properties of components in the model.