ATHABASCA UNIVERSITY

SAFETY CRITICAL SOFTWARE CERTIFICATION USING DESIGN PATTERNS

BY

VARUN MALIK

A thesis essay submitted in partial fulfillment

Of the requirements for the degree of

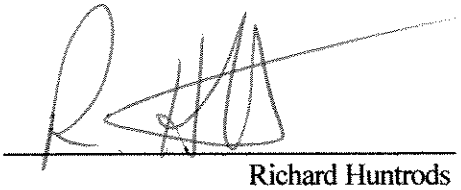
MASTER OF SCIENCE in INFORMATION SYSTEMS

Athabasca, Alberta

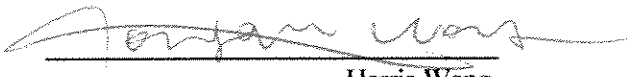
March , 2005

ATHABASCA UNIVERSITY

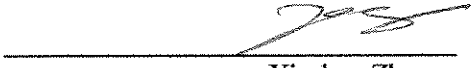
The undersigned certify that they have read and recommend for acceptance the project "SAFETY CRITICAL SOFTWARE CERTIFICATION USING DESIGN PATTERNS" submitted by "VARUN MALIK" in partial fulfillment of the requirements for the degree of MASTERS OF SCIENCE in INFORMATION SYSTEMS.



Richard Huntrods
Supervisor



Harris Wang
Examiner



Xiaokun Zhang
Chair

Date: May 25. 2005

ABSTRACT

The purpose of this research essay is to illustrate the development of safety critical, Reusable Software Component (RSC) using Design Patterns. The research essay demonstrates the process and issues of certification using DO-178B airworthiness assurance standard. The DO-178B standard is restricted to procedural programming languages, which results in a hindrance for using the Object-Oriented principles in aerospace applications. The three High-Level domains pursued in the research are Design Patterns, DO-178B assurance standard and a sample component, which is developed as a reusable sample component. The sample component illustrates the integration of the domains to give a notion for developing critical software design. All the entities in the sample component are portrayed as self-sufficient components, which interact via their exposed interfaces to meet the objective of reuse. The design of the sample component uses Object-Oriented principles and the guidance rules specified in the Object-Oriented Technology in Aviation (OOTiA) Handbook. The rules, which are relevant to the planning and the design phase, are considered throughout the development of the component. The sample component illustrates the incorporation of patterns in the design to ease the process of certification. The research essay can be used as a reference for organizations planning to utilize Design Patterns and Object-Oriented software in the aerospace projects. The maximum benefit to any project can be achieved by reviewing this research in the initial phases, as it provides information on integration of Design Patterns, certification information and development approach to create a safety critical Reusable Software Component (RSC).

COPYRIGHT

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Statement of the Purpose

The purpose of this research is to illustrate the development of safety critical, Reusable Software Component (RSC) using Design Patterns. The research essay will demonstrate the process and issues of certification using DO-178B airworthiness assurance standard. The research essay via the sample component will explore the use of Object-Oriented principles. The design of the component will adhere to the guidelines from the Object-Oriented Technology in Aviation (OOTiA) Handbook.

Research Problem

The design and deployment of safety critical software for aerospace applications has been restricted to procedural programming languages, as there is no specific standard to certify Object-Oriented (OO) software (RTCA Practitioners Course CD, 2005,OOTiA Handbook Vol-1, 2004, p.1-1). The lack of proper certification standard and methodology for the airborne software designed using OO techniques restricts the benefits of using the OO principles like abstraction, encapsulation, modularity, hierarchy, typing, concurrency and persistence (RTCA Practitioners Course CD, 2005,CAST Paper-4, 2000, p.3).

There are various limitations associated with procedural programming. For instance their tendency to turn into "spaghetti-code" makes the program complicated and hence any new programmer on the project would require a certain preparation time in order to

understand the details of the software. This time lag might lead to rescheduling increasing the overhead for the project (Sol, 1998).

The structure of programs developed using OO principles follows the paradigm of abstraction in which the "data as well as the associated data structures" are primary building blocks (Portunity, 2002). "An Object-Oriented approach to programming focuses on data and behaviour, that relates to the data, as opposed to procedural methodologies, which focus on means to manipulate data" (Phillips et al, 1999, p.347).

Solution and Significance

The notion of Design Pattern is one of the most practical solutions while developing RSC. The enormous knowledge of design in different computational areas are captured and used collectively by developers to create robust, reusable and scalable software. Design Patterns are used as tools to create reusable design for common problems that occur in various domains. The reusability of a pattern-oriented design is beneficial for developing generic components for aerospace industry, which can be optimally shared among different vendors. Every component, which becomes a part of the airborne software, has to comply with the rigorous processes of the DO-178B assurance standard. Software certification is important to increase the credibility of the software and to assure end-users the safety of the software. The end-user could be either a person warming up coffee in the microwave or a passenger in a jumbo jet. The software used in designing the plane is more critical when compared to the software for the microwave; as in the former case, the passengers' lives are at stake. As per Michael P. DeWalt in the DO-178B Practitioners Course, software certification pertains to the systems and not software itself, which is approved as a part of the system using assurance standards like DO-178B.

The focus of the research essay is to present information on the integration of three High-Level domains: Design Patterns, sample component designed for reusability, and certification of safety critical software using DO-178B standard. Sample components (MemoryManager, MemoryCaller, and OperatingSystem) are designed using a Model Development Architecture (MDA) tool, Rhapsody, to show how Design Patterns are used in software development to increase the reusability. A set of functional requirements defined for the components are satisfied using Design Patterns. Planning and Design rather than the implementation, testing and verification would be the primary focus of the component. The primary component of the sample is the MemoryManager component and other two components MemoryCaller and OperatingSystem are there to provide a scenario for interaction. MemoryCaller (initial call) requests memory by calling the OperatingSystem. OperatingSystem initializes the memory by sending request to the MemoryManager. MemoryManager initializes the block and returns the handle back to the OperatingSystem. OperatingSystem returns the handle back to the MemoryCaller (initial call). MemoryCaller uses the handle to request operations like insert, delete, and getfreesize from the MemoryManager. The sample component presented in Chapter III is not designed to any specific DO-178B assurance level. The sample component is an example to support the use of Design Patterns for the research. Unified Modelling Language (UML) notation along with the Rational Unified Process (RUP) development methodology for developing OO software would be used to design the component. Although Design Patterns ease the development of OO software, there is no current certification standard for OO software that is used in aerospace industry. The current document supporting the Object-Oriented Software in Aviation (OOTiA) is only a

guideline and not a standard followed by Federal Aviation Authority (FAA) (RTCA Practitioners Course CD, 2005, OOTiA Handbook Vol-1, 2004, p.1-i).

The use of software components designed using Object-Oriented Technology (OOT) in the aerospace industry could open a new horizon for the developers. It would allow them to incorporate the OO design principles in the development of reusable components and certifying them with little apprehension.

Limitations

One of the limitations of the research essay is that DO-178B does not provide any guidelines for applying assurance standards to software developed using OOT. DO-178B does not explicitly specify the use of OOT for the development of software. DO-178B focuses more on the structural programming. Another limitation is that the design of the sample component using the Rhapsody tool is dependent on the availability of the tool. The licence key for the Rhapsody is valid for three months; therefore limiting the time available to complete the development process for the sample component.

Delimitation

The scope of the sample component is confined to focus only on the Planning phase (High-Level Requirements, High-Level Sequence Diagrams) and partial Design Phase (Low-Level Requirements, Low-Level Sequence Diagrams, Object Model Diagrams), which shall be completed in the initial iterations of the development methodology used. The sample component is not a full functional component. The design of the sample component contains only partial sets of requirements, which are chosen to illustrate the use of Design Patterns when developing a RSC. The implementation, testing, verification

and validation phases are not discussed as part of the sample model for components. The essay discusses the issues, which are faced currently in the industry for the certification of OO development. The integral processes (Verification & Validation, Quality Assurance and Configuration Management) for DO-178B certification (RTCA DO-178B Standard, 1992, p.4) are not defined in the scope of the sample component. The report will explain the processes of DO-178B and provide template for a planning artifact that is created as a contract between the developer and the certification authority (CA). An assumption is made regarding the availability of functionality for Operating System (OS), which is not depicted in the sample model but is required for software components implementation. The native OS functionality like scheduling, timing, address translation and other critical functions for objects and variables required for the functionality of the components are not shown as part of the sample.

Glossary of Terms

Acronyms	Description
ACG	Automatic Code Generation
CA	Certification Authority
CAST	Certification Authorities Software Team
CCA	Certification of Civil Avionics
CSI	Certification Services Inc.
CD	Component Diagram
COTS	Commercial off the Shelf Software
DASC	Digital Avionics System Conference
DER	Designated Engineering Representative
EUROCAE	European Organization for Civil Aviation Equipment
FAA	Federal Aviation Authority
FAR	Federal Aviation Regulations
FIFO	First In First Out

GOF	Gang Of Four
GRASP	General Responsibility Assignment Software Patterns
HL	High-Level
ICD	Interface Control Documents
JAA	Joint Aviation Authority
LaRC	Langley Research Center
LDRA	Liverpool Data Research Associates
LL	Low-Level
LOC	Line of Code
MC	Memory Caller
MC/DC	Modified Condition Decision Coverage
MDA	Model Development Architecture
MM	Memory Manager
NASA	National Aeronautics and Space Administration
OMD	Object Model Diagram
OMG	Object Management Group
OO	Object-Oriented
OOT	Object-Oriented Technology
OOTiA	Object-Oriented Technology in Aviation
OS	Operating System
PSAC	Plan for Software Aspects of Certification
RAM	Random Access Memory
ROI	Return on Investment
RSC	Reusable Software Component
RTCA	Radio Technical Commissions of Aeronautics (former name)
RTOS	Real Time Operation Systems
RUP	Rational Unified Development Process
SAS	Software Accomplishment Summary
SCI	Software Configuration Index
SCMP	Software Configuration Management Plan
SCS	Software Coding Standards
SD	Sequence Diagram
SDD	Software Design Description
SDS	Software Design Standards

SOI	Stages of Involvement
SQAP	Software Quality Assurance Plan
SRD	Software Requirement Documents
SRS	Software Requirements Standards
STCP	Software Test Plan
SUB_SYS	Sub-System
SVP	Software Verification Plan
SVR	Software Verification Results
UC	Use Case
UML	Unified Modelling Language

Organization of the Project

The research essay is comprised of five chapters. Chapter I provides the statement of the research, the reason behind the study and the benefits that can be attained by using the proposed solution.

Chapter II gives a literature background on the domain areas of the research that are Design Patterns, DO-178B and the sample component.

Chapter III describes the detailed design of the sample component via Design Patterns. The design includes the set of High-Level requirements, which are satisfied by the Low-Level design.

Chapter IV concentrates on the results of the research. This Chapter provides detailed explanation of the Design Patterns that were used in creating the sample component. It will explain how the technical areas outlined by OOTiA were considered, when developing the sample component.

Chapter V gives the conclusion on the research and recommendations for future research, as the use of OOT in aerospace software is fairly new concept and requires more rigorous approval for certification.

The Appendixes of this essay provide a sample template illustrating the structure for a planning artifact of DO-178B project. They also present the package information providing details of the sample component.

CHAPTER II

REVIEW OF RELATED LITERATURE

The purpose of the following literature review is to provide a synopsis of the main domains of the research essay. The High-Level areas pursued in the research are Design Patterns, DO-178B assurance standard and a sample component, which is developed as a RSC. The review illustrates the methodologies, tools and processes used for developing the sample component. It gives an overview on the specific domains independently and the sample component illustrates the integration of the domains to give a notion for developing critical software design.

Design Patterns

Overview

One of the objectives of Design Patterns is to support the reuse of software architecture and design. "Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain" (Schmidt, 2004, Slide 3). Design patterns give developers access to proven techniques that solve specific classes of problems. Designers can reuse optimized solutions that have worked in the past instead of solving the problem and reinventing the wheel. "Put simply, design patterns help a designer get a design 'right' faster" (Gamma et al., 1995, p. 2).

According to Christopher Alexander,

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Gamma et al., 1995, p. 2).

The above quote about building patterns is also true for OO Design Patterns (Gamma et al., p.2). A pattern has four essential components: pattern name, problem, solution, and consequences (Gamma et al., p. 3). The pattern name defines a vocabulary of the pattern and the problem defines when the pattern is applied to solve the problem (Gamma et al., p. 3). The resulting solution for the problem explains the details of the elements and interactions in the design (Gamma et al., p. 3). The implementation of the solution results in consequences, which are an important aspect of implementation, as they are used to develop alternatives to the solution (Gamma et al., p. 3). "A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object oriented design" (Gamma et al., p. 3).

Pattern Description and Classification

The 23 Patterns defined in the book, *Design Patterns: Elements of Reusable Object Oriented Software,* authored by Gamma, Helm, Johnson and Vlissides are also known as the Gang of Four (GOF) Design Patterns. Table 1, outlines a format, which is followed when describing Design Patterns. The consistent format for the sections makes it easier to learn and to use patterns in the design decisions (Gamma et al., 1995, p. 6).

Table 1: Pattern Description Structure (Gamma et al., p. 6, 7)

Section	Description
Pattern Name and Classification	The name is just what the pattern is called. The classification is creational, structural, or behavioural.
Intent	The following questions should be answered: • What does the Design Pattern do? • What is the rationale and intent? • What particular design issue or problem does it address?
Also Known As	Other names the pattern has been called.
Motivation	A scenario which shows how the problem can be solved using the pattern.
Applicability	When to apply the pattern and how to recognize the situation
Structure	Representation of the Patterns using a graphical notation based on an object Modeling Technique
Participants	The classes' and / or objects in the Design Pattern and their responsibility.
Collaborations	The means by which the participants

	interact to comply with their behaviour.
Consequences	Did the pattern satisfy the objectives that were set out? What are the trade-offs and the results of using this pattern over another?
Implementation	Any specific style that should be followed when using a particular pattern.
Sample Code	A code fragment that shows how one might implement the pattern.
Known Uses	Where this pattern has been implemented previously.
Related Patterns	A pattern that has similar attributes to the one chosen. Look at the differences between the patterns and ensure that the right pattern was chosen.

Design Patterns are classified on the basis of its purpose, which reflects what the pattern does and scope, which tells the developer whether the pattern deals with classes or objects (Gamma et al., 1995, p. 10). The following table, Table 2, from the book *Design Pattern: Elements of Reusable Object Oriented Software* classifies the Design Patterns making it easier to find out which group of patterns is used to solve the problem. Creational patterns are important part of the design and describe how the objects are

created. Creational patterns allow flexibility in what gets created, who creates it, how it is created, and when it is created (Cooper, 2000, p.6). Structural Design Patterns deal with how classes and objects are composed to form larger structures (Cooper, p.6). Behavioural Design Patterns deal with algorithms and the assignment of responsibilities between objects (Gamma et al., p.10).

Table 2: Classification of Patterns (Gamma et al., p.10)

		Purpose		
		Creational	Structural	Behavioural
Scope	Class	Factory Method	Adapter (Class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Momento Observer State Strategy Visitor

The two most common techniques for reusing functionality in an OO system are class inheritance and object composition. Design Patterns propose to "program to an interface, not to an implementation" (Gamma et al., 1995, p. 18). This analogy prevents unnecessary inheritance of methods by subclasses. Defining methods in an interface or an abstract class provide more flexibility for classes implementing these interfaces. The second concept of Design Patterns to create reusable, robust and flexible software is to "Favour object composition over class inheritance (Gamma et al., p. 20).

DO-178B

Overview

DO-178B developed in 1992 by 257 members of Radio Technical Commission of Aeronautics (RTCA) and European Organization for Civil Aviation Equipment (EUROCAE) is the certification standard for Software Consideration in Airborne Systems and Equipment Certification (Wlad, 2003, Slide 3) (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). It is a guidance document, which focuses on the software development processes and objectives to comply for certification. The purpose of DO-178B is to provide a guideline outlining a set of assurance criteria that, along with a software development methodology, can be used for developing software to provide safe functionality in the aerospace industry. DO-178B focuses on objectives and assurance guidelines; which can be used in the organization (RTCA Practitioners Course CD, 2005, CCA-ver.2.0). DO-178B does not explain documentation and development environment set-up in an organization. The members of the committee came up with 66 objectives. If 63 of the 66 objectives were satisfied, the resulting software is acceptable. These acceptable systems can create catastrophic events (Wlad, 2003, Slide 7) (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). In addition, three objectives were identified which make the software comply with the CA liaisons (RTCA Practitioners Course CD, 2005, CCA-ver.2.0) (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). The committee created software levels, which were categorized against the failure conditions. These software levels are listed in Table 3.

Table 3: DO-178B Objectives and Levels (Wlad, 2003, Slide 6, 7)

Software Level	Failure Condition	Number of Objectives
Level A	Catastrophic	66
Level B	Hazardous	65
Level C	Major	57
Level D	Minor	28
Level E	No Effect	None

DO-178B assurance standard does not imply that by following the standards, all the safety critical errors will be detected for the airborne software (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). The rigorous process and robust testing followed during the development of a system applying DO-178B standard reduces the occurrence of errors like the Mars rover Spirit. In this incidence, the error causes the rover to become non-responsive and to terminate while attempting to allocate more files than its Random Access Memory (RAM)-based directory structure could accommodate (Hachman, 2004).

The main actors in the DO-178B process for software approval are the Approver, the Applicant and the Developer. The CA or its designee, Designated Engineering Representative (DER), is the final approval authority (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). The CA for North America is the FAA and for Europe is the Joint Aviation Authority (JAA). DER is an appointed individual by the FAA, who performs a variety of roles (software certification, physicians to provide medical certificates for pilots, examiners for licensing of the pilots) on its behalf (RTCA Practitioners Course CD, 2005, CCA-ver.2.0). The FAA enforces the regulations by Federal Aviation Regulations (FAR), which are organized by sections known as Parts as

shown in Table 4 (RTCA Practitioners Course CD, 2005, CCA-ver.2.0). The authorities examine the evidence of compliance, which is presented by the applicant. The applicant, in negotiation with the developer and with concurrence with the CA, determines the level needed for the software approval (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005).

Table 4: Federal Aviation Authorities Regulations (RTCA Practitioners Course CD, 2005, CCA-ver.2.0)

Parts	Activities
Part 1	Definition and Abbreviations
Part 21	Certification Procedures for Products and Parts
Part 23	Airworthiness Standards: Normal Utility, Acrobatic and Commuter Airplanes
Part 25	Airworthiness Standards: Transport Category Airplanes
Part 27	Airworthiness Standards: Normal Category Rotorcraft
Part 29	Airworthiness Standards: Transport Category Rotorcraft
Part 33	Airworthiness Standards: Aircraft Engines
Part 34	Fuel Venting and Exhaust Emission Requirements for Turbine Engine Powered Airplanes
Part 39	Airworthiness Directive

Part 91	General Operating and Flight Rules
Part 121	Operating Requirements Domestic, Flag and Supplemental Operations
Part 183	Representative of the Administrator

The assurance level for the airborne software can be guaranteed by practicing safety critical standards, such as DO-178B, which provides evidence that development processes and products have a desired integrity (Carpenter, 1999, p. 23). The level of integrity is established by the severity of the hazard in question (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). For example, systems, which control the Landing Gear of the plane, are designed with more scrutiny than the passenger entertainment systems. This different level of scrutiny is formalized as software assurance levels (RTCA Practitioners Course CD, 2005, CCA-ver.2.0).

DO-178B Objective Criterion

The objectives of DO-178B are based on evaluating the reliability of the artifacts produced in each of the development phases and the relationship between the artifacts, which are independent of the phases of the life-cycle (RTCA Practitioners Course Lecture Notes, 2005, CSI-002). Adhering to the following criterion satisfies the DO-178B objectives (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005).

Consistence and Compatibility

The Consistency criterion in DO-178B is defined as, "the requirement does not conflict with each other" (RTCA DO-178B Standard, 1992, p.35). The artifacts, including the High-Level requirements, Low-Level requirements, and source code define a specific behaviour, which is synchronous and does not create a conflict. Compatibility criterion is satisfied when the software developed from the Low-Level requirements is compatible with the end target (RTCA Practitioners Course Lecture Notes, 2005, CSI-005).

Traceability and Compliance

"Traceability is primarily an allocating function. It ensures that all the artifacts created at one phase of the development life-cycle have been addressed at the next lower level of the life-cycle and vice versa" (RTCA Practitioners Course Lecture Notes, 2005, CSI-005). The High and Low-Level requirements and the High and Low-Level derived requirements are downward traceable to their Low-Level counterparts. The High and Low-Level derived requirements do not have upward traceability (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). The compliance determines whether the requirement is correct (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). A traceability matrix tracks the traceability requirements between the artifacts of different phases and the matrix is verified for its correctness and completeness to meet the criteria of traceability and compliance (Carpenter, 1999, p. 25). As the size of the project increases, the manual traceability for the requirements is not feasible and tools like DOORS by Telelogic (Telelogic, 2005) and VeroTrace by VeroCel (VeroCel, 2005) are used to create artifacts and to verify the traceability matrix.

Verifiability and Accuracy

The Verifiability criterion checks that "each High-Level requirement can be verified" (RTCA DO-178B Standard, 1992, p. 36). This could be accomplished by reviews conducted on the life-cycle data. The accuracy criterion for components (e.g. requirements, source code) is primarily concerned with whether the requirement is stated properly and performs the intended behaviour inclusive of all of the derived requirements (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). Checking them against the standards and manually reviewing them using checklists and producing more refined requirements can evaluate the accuracy of the components. The accuracy criterion for the algorithms used in the design, ensures the accuracy and behaviour of the proposed algorithms, by rigorous testing and evaluating the outputs for precision over the range of inputs (RTCA Practitioners Course Lecture Notes, 2005, CSI-005). The accuracy for the algorithm can be determined by analysis, simulation and calculations that also become the verification artifacts (RTCA Practitioners Course Lecture Notes, 2005, CSI-005).

Testing and Completeness

The criterion of testing is satisfied by requirement based testing in DO-178B in which test cases are created for each of the High-Level and Low-Level requirements (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). The approach for creating tests for High-Level requirements also indirectly tests part of Low-Level requirements reducing the number of explicit Low-Level test cases. The High-Level test cases cover the Low-Level normal and robust test cases (RTCA DO-178B Standard, 1992, p. 41). The three types of testing, defined by DO-178B, are compliance or normal range test, robustness tests, and compatibility test (Carpenter, 1999, p. 28). Compliance testing confirms that

the software works according to its specification over the relevant ranges. Robustness testing covers the behaviour of the software in abnormal conditions by exercising the unexpected or out-of-bound values (RTCA Practitioners Course Lecture Notes, 2005, CSI-008). Compatibility testing is to ensure that the software is executed correctly within the operational target hardware environment (RTCA Practitioners Course Lecture Notes, 2005, CSI-008).

The completeness criterion of DO-178B covers those tests, which are not exercised by the requirements-based testing (RTCA Practitioners Course Lecture Notes, 2005, CSI-008). The Low-Level test cases are created to verify the structural coverage of the source code. The main objective of this coverage is to provide evidence that all the software was explored (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). To explore this, every function of the source code has at least one test case that proves that the executable code properly implements the behaviour as desired (RTCA Practitioners Course Lecture Notes, 2005, CSI-008). The test also verifies that the internal structure of the function is thoroughly explored and all the paths and decisions have been explored. This type of testing, also known as a Modified Condition Decision Coverage (MC/DC), uses logic to guarantee that the number of test cases is, at most, one more than the total conditions in the statement (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). MC/DC testing includes the criterion for statement coverage and decision coverage (RTCA Practitioners Course CD, 2005, CAST Paper-6, 2000) (RTCA Practitioners Course, 2005 Lecture Notes, CSI-008). Completeness criterion is also satisfied by testing all of the behaviour, which is introduced without the knowledge of the developer (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). The most

common instance of this hidden behaviour is the compiler-generated code (RTCA Practitioners Course Lecture Notes, 2005, CSI-008) (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). The size of the software increases the complexity of the source making it difficult to achieve the DO-178B objectives using manual tests. Tools like Liverpool Data Research Associates (LDRA) can be used which are capable of full coverage analysis and generate reports that outline the coverage metrics for all the functions (LDRA Website, 2005).

Development Phases and Artifacts

DO-178B certification standard creates artifacts, which provide a means of compliance for the projects. The artifacts are created at the end of the phases of the development methodology followed by the project. The main phases for software development are Planning, Development (Requirements and Design), Implementation and Testing. All phases of the project follow integral processes and standards which are developed in the planning phase for a new DO-178B project (Carpenter, 1999, p. 23). Verification and Validation, Software Quality Assurance and Configuration Management processes are integral to all the phases of the development. Every phase has inputs, outputs and transition criteria that create their artifacts. The following three standards are required for all projects:

- Software Coding Standards (SCS): The following document provides the guidelines to produce source code according to the guidelines in RTCA DO-178B Section 11.8. In the words of DO-178B, it defines "the programming languages, methods, rules and tools to be used to code the software" (RTCA DO-178B Standard, 1992, p. 69).

- Software Requirements Standards (SRS): The following document is prepared in accordance with the Technical Commission for Aeronautics, *Software Consideration in Airborne Systems and Equipment Certification* Document No. RTCA DO-178B, Section 11.8. In the words of DO-178B, it defines "the methods, rules and tools to be used to develop High-Level requirements" (RTCA DO-178B Standard, 1992, p. 69).

- Software Design Standards (SDS): The following document defines the standards for the Software Design Document. This document is prepared in accordance with the Technical Commission for Aeronautics, 'Software Considerations in Airborne Systems and Equipment Certification' Document No. RTCA/DO-178B (RTCA DO-178B Standard, 1992, p.69).

Table 5, shows all the inputs, outputs and transition criteria, which are produced at different phases for DO-178B certification (RTCA DO-178B Standard, 1992). The phases are also referred as Stages of Involvement (SOI), which pertain to the involvement of DER at the end of the phases (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005) (RTCA Practitioners Course Lecture Notes, 2005, CSI-014). The three main artifacts, which are submitted to the certification authorities for compliance, are as follows:

- Plan for Software Aspects of Certification (PSAC): This certification plan is the first communication between the certification authorities and the applicant. This is the contract between both the parties outlining how things will be done throughout the software development (RTCA DO-178B Standard, 1992, p. 64).

- Software Configuration Index (SCI): This document outlines the details of the configuration of the software (RTCA DO-178B Standard, 1992, p. 72).

- Software Accomplishment Summary (SAS). This document is the last document produced for the software project and reflects the PSAC but instead of explaining how things will be done it describes how things were done (RTCA DO-178B Standard, 1992, p.74).

Table 5: List of Artifacts

Phase	Input	Output	Transition Criteria
Planning	Software Requirement Standards (SRS) Software Coding Standards (SCS) Software Design Standards (SDS)	Plan For Software Aspects of Certification (PSAC) Software Development Plan (SDP) Software Configuration Management Plan (SCMP) Software Quality Assurance Plan (SQAP) Software Verification Plan (SVP)	*Criteria for transition into the next phase:* All the documents are complete. *Criteria for re-entry into this phase*: Process or Standard change.

Development	Requirements	PSAC SDP SCMP SQAP SVP	Software Requirement Documents (SRD)	*Criteria for transition into the next phase:* All the documents are complete. Completion of the requirements review. *Criteria for re-entry into this phase:* Requirement change found in the design.
	Design	SRD SDP SCMP SQAP SVP	Software Design Description (SDD)	*Criteria for transition into the next phase:* All the documents are completed. Completion of the Design review *Criteria for re-entry into this phase:* Design change.
Implementation		SDD SCMP SQAP	Software Configuration Index (SCI) Source Code	*Criteria for transition into the next phase:* All the documents are completed. Completion of the Implementation review

			Criteria for re-entry into this phase: Implementation change found in the source. Correction of misconceptions.
Testing	SVP SRD SCMP SQAP SDD Source Code	STCP	*Criteria for transition into the next phase:* All the documents are completed. Testing review indicates all expected goals of the project are met. *Criteria for re-entry into this phase* Change found in the Test Cases and the source. Correction of misconceptions.
Final Verification and Validation	SVP STCP Executable	Software Verification Results (SVR)	Integral Process
Integration *(optional)*		Interface Control Document (ICD)	
Final		SCI Software Accomplishment Summary (SAS)	

Description of the all the artifacts produced during the life-cycle is shown in Appendix A. In Appendix C, a sample template for PSAC is included. A PSAC is an artifact created during the Planning phase of the DO-178B project.

Object-Oriented Design and OOTiA

"*An Empirical Comparison of Modularity of procedural and Object-oriented Software*" was a study done by Lisa K. Farrett and Jeff Offutt. One of the hypotheses for the study was to compare the module size for program written in procedural and OO language. The conclusion, based upon the results of the study (Farrett & Offutt, 2002) was that OO programs are more modular than procedural programs. "The average module size of the object-oriented programs, measured as lines of code, was half that of the procedural programs" (Farrett & Offutt, 2002). The data and its graphical representation for the module size comparison can be viewed in Appendix B.

The design objective of OO programming is to create robust, adaptable and reusable software. An object is the primary fundamental concept of OO paradigm (Goodrich & Tamassia, 2001, p. 3). An object is an element in a computer software system that resides in the memory for a portion of the run-time of that system (Douglas, 2003, p.7). Every object has a state (data) and behaviour (operations on data) (Gamma et al., 1995, p. 11). A class, another important concept of the OO programming is a template that facilitates the creation of objects (Goodrich, 2001, p.3). It is an element in a computer software system that is used to generate one or more objects of a particular type. Object orientation, groups' similar objects into types or classes (Eckel, 2000, p.32).

Both objects and classes include methods that carry out operations or tasks. The analysis and design phases of OO software development identify tasks that need to be performed, and assign them to particular classes (Larman, 2005, p.7). When the design of a class is implemented, executable code is written to carry out the tasks assigned to that class, and included in the definition of that class as its methods. When an object is created from that class, copies of these methods are created in memory, which carry out tasks on behalf of the object (Douglass, 2001, p.7).

The benefits of using OOT principles in the aerospace industry lead to the increased desire among the software developers and certification authorities on finding how the objectives of DO-178B can be satisfied, when using OOT for development (Hayhurst, Holoway, Michael, 2003, p.1). In year 2000, an OOT project was initiated with National Aeronautics and Space Administration (NASA) Langley Research Center (LaRC). The purpose of this project was to provide input to the FAA for developing policy and guidance for the OOTiA systems and to support certification authorities on the use of OOTiA. In October 2004, the committee listed the considerations, issues and the best certification practices for the use of OOTiA, by releasing a document called the OOTiA Handbook. The OOTiA Handbook comprised of four volumes listed in Table 6. The OOTiA Handbook does not illustrate a policy or guideline from the FAA, but rather provides an input for the organizations considering using OOTiA in their projects (RTCA Practitioners Course CD, 2005, OOTiA Handbook, 2004, Vol-3, 2004, p. 3-i).

Table 6: OOTiA Handbook Volume Description (RTCA Practitioners Course CD, 2005, OOTiA Handbook, 2004)

Volume	Description and Audience
Volume 1	"Handbook Overview: Provides background and foundational information needed to use all other volumes Target Audience: All Handbook users" (RTCA Practitioners Course CD, 2005, OOTiA Handbook, 2004).
Volume 2	"Considerations and Issues: Poses questions to be answered before committing to OOT Presents concerns raised about OOT that are relevant to certification and safety without discussing approaches for addressing these concerns Categorizes, summarizes, and discusses key issues Provides issue rationale and ties to DO-178B life-cycle processes Target Audience: Project planners, decision makers, certification authorities" (RTCA Practitioners Course CD, 2005, OOTiA Handbook, 2004).
Volume 3	"Best Practices: Identifies best practices to safely implement OOT in aviation by providing some known ways to address the issues documented in Volume 2 Target Audience: Developers, certification authorities" (RTCA Practitioners Course CD, 2005, OOTiA Handbook, 2004).
Volume 4	"Certification Practices: Provides an approach for certification authorities and designees to ensure that OOT issues are addressed Target Audience: Certification authorities, designees" (RTCA Practitioners Course CD, 2005, OOTiA Handbook, 2004).

The committee of the document came up with technical areas that are revisited to satisfy the objectives of DO-178B when developing software, using OOT (RTCA Practitioners Course CD, 2005, OOTiA Handbook, 2004, Vol-3, 2004, p. 3-1). The organization was responsible to illustrate that the product being developed satisfies these criteria either by the proposed solution in the volume or some other generic and efficient methodology followed by the developers. The technical areas for which the issues and guidelines are discussed in the OOTiA Handbook are:

- Single Inheritance
- Multiple Inheritances
- Templates
- Inlining
- Type Conversion
- Overloading and Method Resolution
- Dead and Deactivated Code and Reuse
- OO Tools
- Traceability
- Structural Coverage

Sample Component

Overview

The purpose of the sample components for the research essay is to illustrate an abstract software model, which uses Design Patterns and DO-178B assurance standard during the development cycle. The primary actor is the MemoryManager component and

other two components are there to provide a scenario for interaction. The secondary actors are the MemoryCaller component, that requests memory and the OperatingSystem component, which is responsible for usage of resources and supporting functionality required for the allocation of memory.

The OS used in the aerospace industry is designed to provide a safety critical functionality in real-time environment. In real-time systems, correctness depends not only upon providing a correct solution to a problem, but also within a given time constraint (Zhang, 2003). A system is said to have failed if it produces the correct logical results but in wrong time (Zhang, 2003). A real-time system has to fulfill its purpose under predictable and often extreme load conditions including Timeliness, Simultaneous Processing, Predictability and Dependability (Zhang, 2003). OS providing real-time functionality are also known as Real-Time Operating Systems (RTOS). In the aerospace industry, RTOS that satisfies the objectives of DO-178B are developed as Reusable Commercial off the Shelf Software (COTS) (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). Two such RTOS which are DO-178B Level A certified are VxWorks and Integrity® 178B RTOS (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). VxWorks provides a deterministic memory management by allowing memory allocation at run-time until the function; "*noMoreAllocation*" is called (Romanski, 2002, p.2). Once the function, "*noMoreAllocation*" is called the de-allocation of memory will no longer be allowed (Romanski, 2002, p.2). Integrity 178B RTOS "kernel guarantees bounded time by eliminating dynamic allocation feature" and uses hardware mechanisms to provide memory protection (GreenHills Software Inc., 2005). Memory management is an integral part of any OS (Crowley, 1997, p.4). However for the

research essay, it is portrayed as a reusable MemoryManager component, which interacts with other components of the subsystem via its interface.

Memory management is primarily concerned with the allocation of a fixed capacity physical memory to the requesting caller. The OS creates an illusion by dividing up the physical memory into virtual parts (Crowley, 1997, p. 10). In the MemoryManager component, the MemoryCaller requests memory for its operations from the OperatingSystem component and is returned a handle for a block of memory, which it uses for processing. The MemoryManager breaks the physical size of heap passed by the OperatingSystem component into virtual fixed partition blocks of same sizes. MemoryManager partitions the memory into a set of non-overlapping equal sized blocks. The OS does the address translation between the virtual addresses to physical address. The block handle given back to the MemoryCaller is the virtual address for the initial position of the block used by MemoryCaller. Even though fixed sized partition creates internal fragmentation (Aho et al., 1983, p. 381), it is a deterministic approach and a good strategy for developing safety critical aerospace applications (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). The MemoryManager component design follows the best-fit strategy for data allocation. The entire list of block is searched and the best one found is allocated (Goodrich, 2001, p.198). All the fixed blocks created are optimum as the maximum size of the block required by the MemoryCaller along with the physical size provided by OperatingSystem component is used to create the fixed size for the blocks. The requirement of memory to perform operations by all the MemoryCallers is defined in a configuration file, which is shared between the MemoryCaller, the OperatingSystem and the MemoryManger.

Component and UML Notation

"UML is the predominant OO modeling language that is currently being considered, by the aviation community" (RTCA Practitioners Course CD, 2005, OOTiA Handbook, Vol-3, p.3-52). The sample component represents the structural, functional and behavioural aspects which are defined using UML; a third generation language owned by Object Management Group (OMG) and a defacto standard for software object modeling (Douglass, 2003, p.4). UML notation is used to define the semantic model and how the MemoryCaller, OperatingSystem and MemoryManager perceive the model when creating design time classes and run-time objects. The component also specifies the interaction between the objects via Sequence Diagrams to implement the behaviour defined in the Use Cases.

The use of UML for real-time embedded development can be justified by decreasing the development time for complex system, and by increasing the return on investment (ROI) for the company. The vocabulary and rules of UML focus on the conceptual and physical representation of a system (Douglass, 2003, p.5, 56). By adopting UML notations, development teams can communicate among themselves and others using a defined standard.

The OO analysis for the visual models created using UML notation uses the GOF patterns and the underlying General Responsibility Assignment Software Patterns (GRASP) principles (Larman, 2005, p.277, p.291). In Craig Larman's book *Applying UML and Patterns,* he explains the nine GRASP patterns (Creator, Controller, Pure Fabrication, Information Expert, High Cohesion, Indirection, Low Coupling Polymorphism Protected Variations) and provides an analogy that most Design Patterns

can be seen as specialization of a few basic GRASP principles (Larman, 2005, p.291, p.280). The GRASP and GOF patterns are design principles, which are used for object modelling and defining a means for assigning responsibilities and collaborations between objects (Larman, 2005, p.273). This type of design is known as responsibility-driven design (RDD) (Larman, 2005, p.273).

Component Development Tool

The components are designed using a UML compliant MDA tool Rhapsody from Ilogix. Rhapsody increases the efficiency of design and reduces the cost of development. It does so by creating a "software design graphically", executing it and validating the design during development, before deploying on the target system (Ilogix, 2005). The components created using the Rhapsody interface are transformed to source code using the built-in code generators. The Rhapsody product family offers development environment with implementation of C, C++, Ada and Java languages (Ilogix, 2005). Rhapsody's built in feature of model code associability maintains the integrity of the model by synchronizing the source code generated by the model. The additional tools provided by Ilogix can be integrated with Rhapsody to ease the development life-cycle and to create quality driven systems, which easily comply with certification standards (Ilogix, 2005). The design automation tool allows drawing visual diagrams and managing the semantics of the component while maintaining consistency by checking the validity using simulation (Douglass, 2003, p.4, 5). Simulation of the model is used primarily to develop verification test cases and validate the results by executing them on the model. Rhapsody is classified as a development tool according to DO-178B tool qualification standard. This is because Rhapsody is used to output a part of the component (Use Cases,

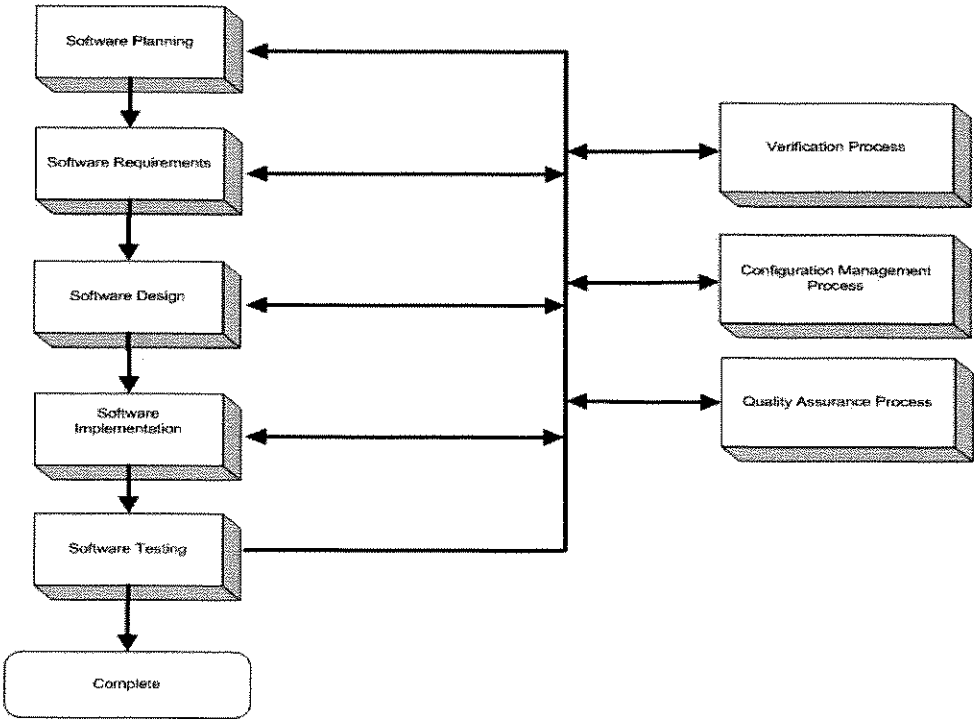
Sequence Diagrams, Object Model Diagrams) which modify "the meaning of or create new requirements, design, code or other data" (RTCA Practitioners Course Lecture Notes, 2005, Slide CSI-011). To satisfy the DO-178B objective, a tool qualification plan is required for all developmental tools which state that it complies with its operational requirements under normal conditions (RTCA DO-178B Standard, 1992, p. 82).

Component Design Methodology

The design of the sample component follows a RUP. The iterative and incremental Use Case driven methodology is an optimum choice for the development of an RSC. The increased ROI incentive for using an RSC in aerospace industry led to the advent of Advisory Circular on December 7th 2004 by FAA to provide an acceptable means of compliance for RSC developers, integrators and applicants (RTCA Practitioners Course CD, 2005, AC-20-148, 2004). The circular provides guidelines for RSC to gain FAA's "acceptance" of a software component for aerospace software (RTCA Practitioners Course CD, 2005, AC-20-148, 2004). It also explains how full or partial credit can be achieved for compliance with the RTCA DO-178B objectives. The MDA tool following the RUP creates the artifacts for the sample component, which is designed to achieve the reusable incentive. The RUP methodology is an ideal approach for developing a RSC as compared to a non-deterministic waterfall methodology. A study done on numerous software projects concluded that the percentage of change in software projects is 35%-50% for large projects (Larman, 2005, p.24). The objective of a RSC is to develop a component, which can be easily plugged-in to the user's domain. The possibility of change during the development of RSC is relative to the user's requirements. Hence following a "timeboxed" process of development can reduce the cost and can increase the

ROI for both the users and the developers (Larman, 2005, p.23). The phases of the unified process are mapped to SOI of DER followed by the DO-178B standard. Figure 1, shows the different phases of development for the sample component. The sample component for the research focuses on the planning and requirement and partial design phase. The implementation, testing of the sample component is left as part of future research.

Figure 1: Life-Cycle Phases



The Verification, Configuration Management and Quality Assurance processes are concurrent processes, which are performed in each iteration to meet the objectives of DO-178B (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). The analysis and

result of the components provide an overview of the current issues faced in the industry for the certification of reusable software component using OOT.

Figure 2: Iterative Development (Larman, 2005, p.20)

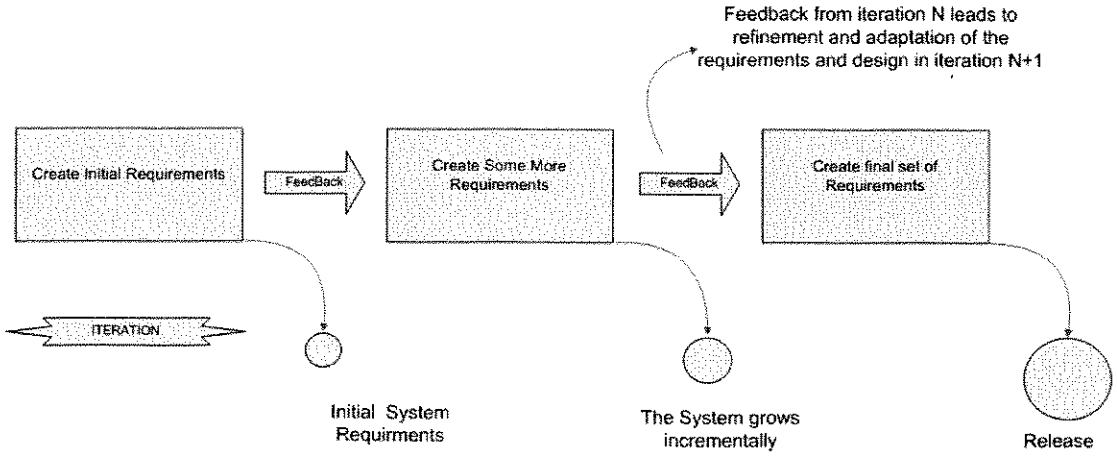


Figure 2, illustrates the iterative development approach used for the design of the sample component. It comprises of several iterations in sequence during the life-cycle phases (Larman, 2005, p.20). Each iteration release is composed of activities such as requirements analysis, design, programming, and test. The iterations for the sample component create a set of requirements that are refined as the component grows iteratively. In the first iteration of the sample component, High-Level Use Cases and High-Level Sequence Diagrams are created. In subsequent iterations, Low-Level artifacts including Low-Level requirements, Sequence Diagrams and Object Model Diagrams are created which satisfies the High-Level requirements.

In the above literature review, all three High-Level domains of the research are explored. The standard available for certification of systems developed using OO

software is a recent addition as a guideline to the aerospace industry. The review gave the background on the existing DO-178B standard and explains the technical areas, which are proposed as guidelines for OO development. The review also illustrated the description of the Design Patterns and the sample component MemoryManager, which is created via Design Patterns.

CHAPTER III

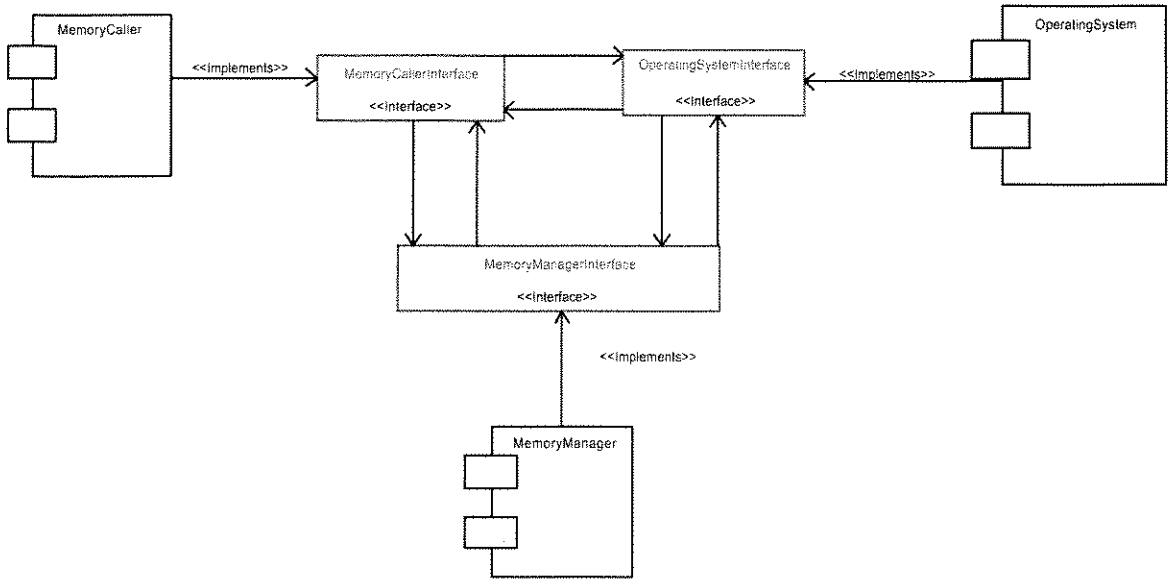
METHODOLOGY

High-Level Component Diagram

All the entities in the sample component are portrayed as self-sufficient components, which interact via their exposed interfaces. The internals of the components are hidden from each other and this leads to the reusability of the components. However, for the instance of initialization, the components are also aware of some internal details of the other components. The High-Level Component Diagram provides a domain view for the sample and as a DO-178B standard this artifact becomes the part of the Planning documents.

Component Diagram Name: SUB_SYS_HL_CD

Figure 3: High-Level Component Diagram



In Figure 3 above, the interaction between the MemoryCaller, MemoryManager and the OperatingSystem components is illustrated. All three components have their specific interfaces, which are implemented by the respective components. Components interact with each other via the interfaces. MemoryCaller component implements the MemoryCallerInterface, MemoryManager implements the MemoryManagerInterface and OperatingSystem implements the OperatingSystemInterface. The initialization of the MemoryManager component begins when the MemoryCaller (initial call) asks for memory by calling a method from the OperatingSystem component's interface. OperatingSystem initializes the MemoryManager with the size of the heap. MemoryManager returns the handle for the initialized heap back to the OperatingSystem. The OperatingSystem momentarily returns it to the MemoryCaller (initial call). The MemoryCaller encapsulates the methods from the MemoryManagerInterface as parameters and sends the request to the MemoryManager. The status of the operation on MemoryManager is returned back to the MemoryCaller. Each of the components has their respective packages, which explain the internal design of the component. The details of the packages are explained in the Appendices as follows:

Package Name	**Appendix**
MemoryCallerPackage	Appendix D
MemoryManagerPackage	Appendix E
OperatingSystemPackage	Appendix F

The following section describes the High-Level and Low-Level Use Cases and Sequence Diagrams for the sample component.

High-Level Use Cases

The sample component's High-Level requirements are illustrated as Use Cases. As per the DO-178B, artifacts' High-Level requirements are defined in the SRD. The requirements for the sample components are designed using visual diagrams created in Rhapsody via embedded UML notation. There is no specific standard when developing requirements using MDA tools; hence the requirements depicting the OO design of the components are mapped to the Use Cases (RTCA Practitioners Course CD, 2005, OOTiA Handbook, 2004, Vol-3, p. 3-57). Each Use Case requirement is a separate entity and is traceable down to Low-Level requirements. The Use Case is not traceable up to system requirements because the intent of the sample components is to develop a reusable software component, which can be plugged into many different systems. All the High-Level Use Cases are treated as derived requirements and are not traced up to any specific customer requirements (Michael P. DeWalt RTCA Practitioners Course Lecture, 2005). High-Level requirements are further detailed to demonstrate the design and functionality in Low-Level requirements. The traceability matrix for the sample component depicts how each High-Level requirement in the Use Case and Sequence Diagram is satisfied by the Low-Level design via Sequence Diagrams and Object Model Diagrams.

Use Case Diagram name: SUB_SYS_HL_UC_001

Figure 4: Sub-System High-Level Use Case 001

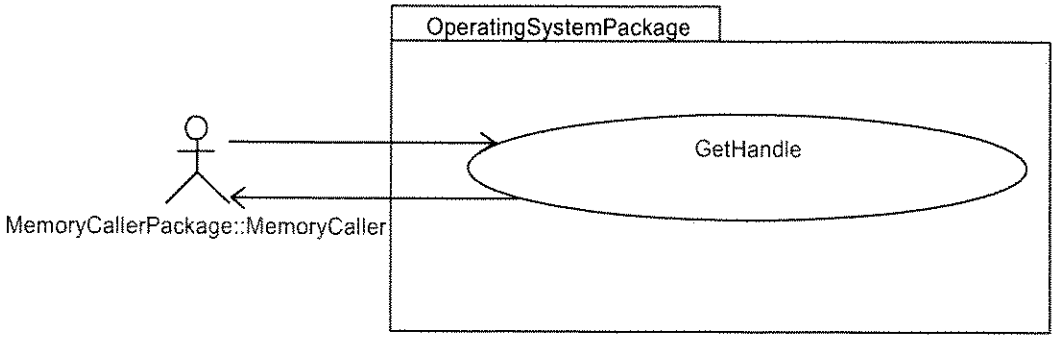


Figure 4, illustrates a High-Level Use Case for the model showing the memory handle request made by the MemoryCaller. The actor in the Use Case is the MemoryCaller (initial call), which belongs to the MemoryCallerPackage and the Use Case is GetHandle. In this Use Case, the MemoryCaller initializes the block of memory for the first time. The address of the block is returned back to this MemoryCaller and the initial caller is responsible for blocking any further initializations. Subsequent calls from the MemoryCaller are made directly to the MemoryManager. The system boundary of the Use Case is the OperatingSystemPackage, which gives the handle for the block back to the initial MemoryCaller.

Use Case Diagram name: SUB_SYS_HL_UC_002

Figure 5: Sub-System High-Level Use Case 002

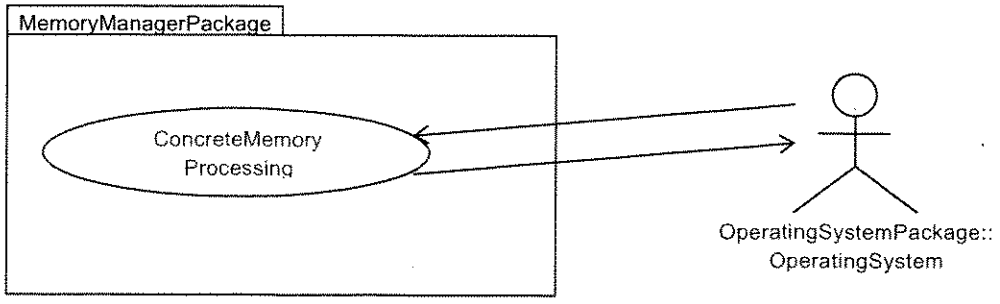


Figure 5, illustrates a High-Level Use Case for the model showing the initialization request sent by the Operating System to the MemoryManager. The actor in the Use Case is the OperatingSystemPackage and the Use Case is ConcreteMemoryProcessing. OperatingSystem is the first entity to initialize the MemoryManager with the size of the master block. The OperatingSystem shares a configuration file with the Memory Manager. This file defines the memory requirements for different MemoryCallers. The system boundary of the Use Case is the MemoryManagerPackage. The MemoryManager using the initial size of heap, provided by the OperatingSystem and the memory needs of the MemoryCallers (defined in the configuration file) creates fixed size blocks. The handle for the new block is returned back to the MemoryCaller (initial call) in response to its memory request.

Use Case Diagram name: SUB_SYS_HL_UC_003

Figure 6: Sub-System High-Level Use Case 003

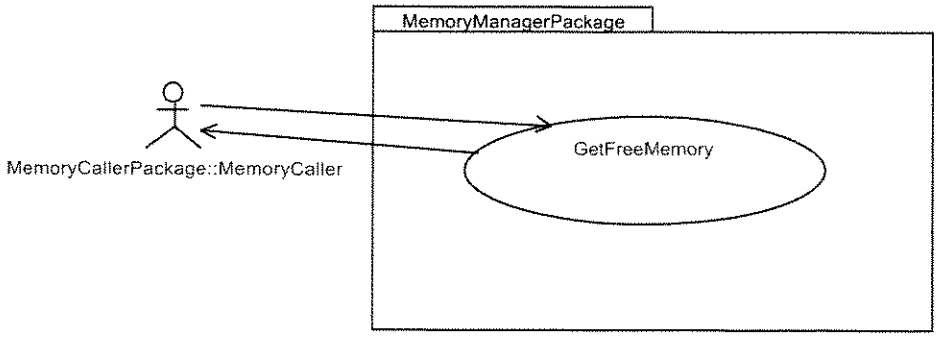


Figure 6, illustrates a High-Level Use Case for the model showing the request to query the available size of the memory. The actor in the Use Case is the MemoryCallerPackage and the Use Case is GetFreeMemory. The MemoryCaller calls the MemoryManager to get the status information on the free blocks on the heap. The system boundary of the Use Case is the MemoryManagerPackage, which returns the MemoryStatus of the request back to the MemoryCaller.

Use Case Diagram name: SUB_SYS_HL_UC_004

Figure 7: Sub-System High-Level Use Case 004

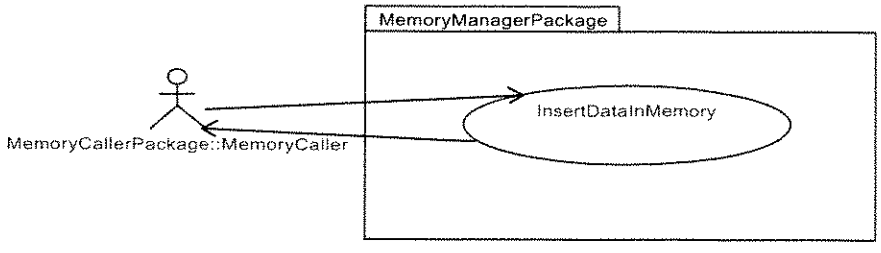


Figure 7, illustrates a High-Level Use Case for the model showing the request to insert data into the memory. The actor in the Use Case is the MemoryCallerPackage and the Use Case is InsertDataInMemory. MemoryCaller calls the MemoryManager to insert data in the allocated block. The system boundary of the Use Case is the MemoryManagerPackage, which returns the MemoryStatus of the request back to the MemoryCaller

Use Case Diagram name: SUB_SYS_HL_UC_005

Figure 8: Sub-System High-Level Use Case 005

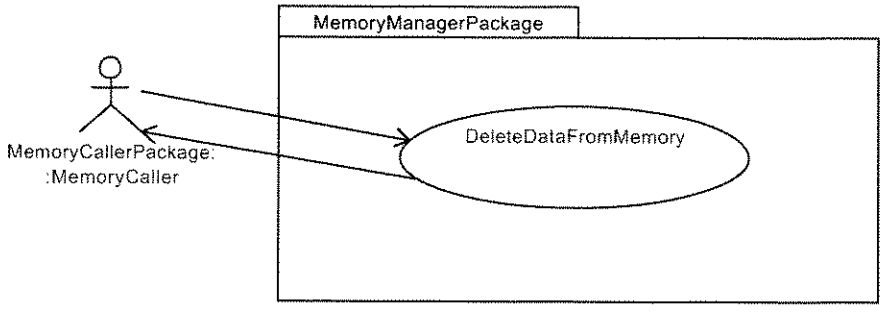


Figure 8, illustrates a High-Level Use Case for the model showing the request to delete data from the memory. The actor in the Use Case is the MemoryCallerPackage and the Use Case is DeleteDataFromMemory. The MemoryCaller calls the MemoryManager to delete data from the allocated block. The system boundary of the Use Case is the MemoryManagerPackage, which returns the MemoryStatus of the request back to the MemoryCaller.

High-Level Sequence Diagrams

Developing requirements using visual models creates Sequence Diagrams, which show the flow of messages and behaviour between the entities in a timely manner. DO-178B does not specify any specific criteria to deal with these High-Level Sequence Diagrams. In this model, these are also depicted as part of High-Level requirements.

Sequence Diagram name: SUB_SYS_HL_SD_001

Figure 9: Sub-System High-Level Sequence Diagram 001

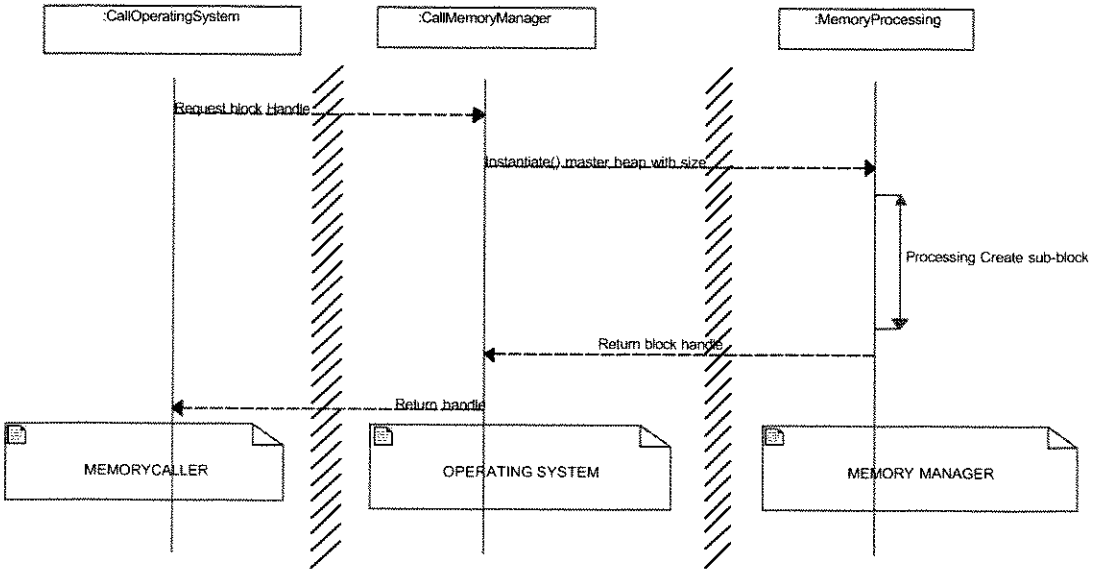


Figure 9, illustrates a High-Level Sequence Diagram showing the process of initialization of the MemoryManager. MemoryCaller requests a block from the OperatingSystem. OperatingSystem calls the MemoryManager with the initial size of the heap to initialize the Memory. MemoryManager creates the fixed sized blocks (sub-

blocks) and returns the starting address of the block of the master heap to the MemoryCaller, which uses the block of memory for it operations.

Sequence Diagram name: SUB_SYS_HL_SD_002_01

Figure 10: Sub-System High-Level Sequence Diagram 002_01

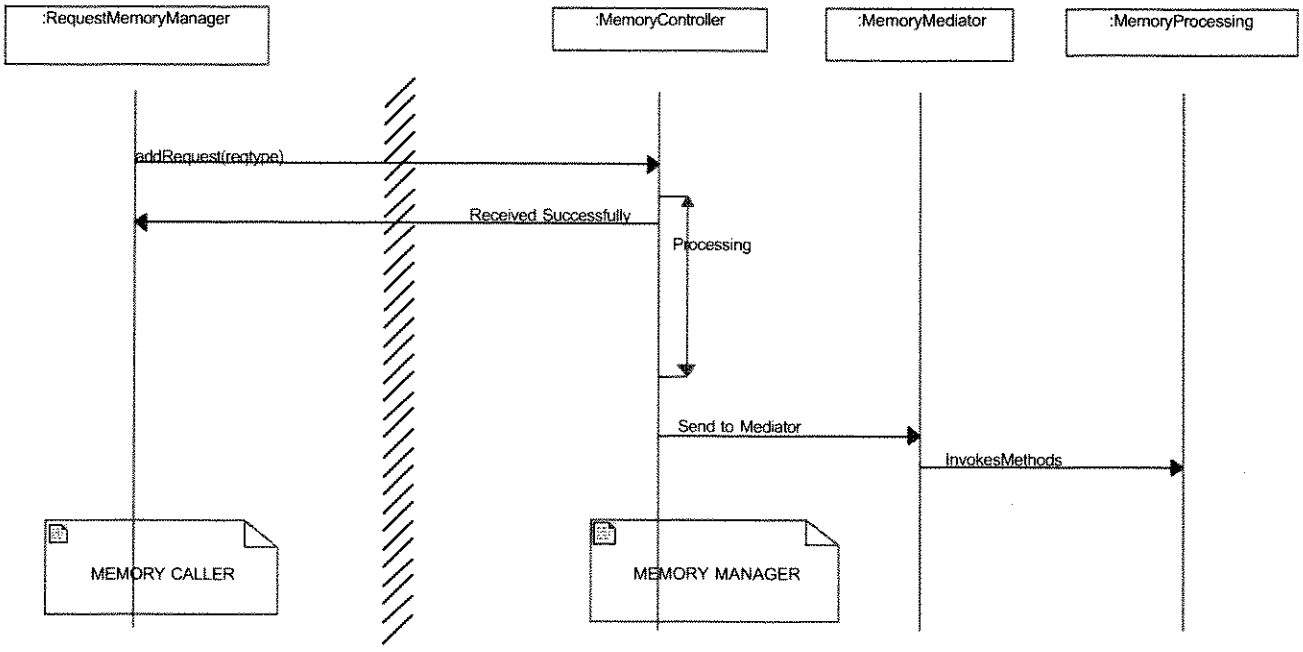


Figure 10, is a High-Level Sequence Diagram showing the invocation of services offered by MemoryManager. MemoryCaller passes its request to the MemoryController. MemoryController returns a confirmation. The request is processed and managed by the MemoryController and passed to the MemoryMediator who has knowledge on all the internals of the MemoryManager. The Mediator invokes the correct methods. This Sequence Diagram shows all method invocation by just one message flow

(invokesMethod), but in real-time this is specific for each method. MemoryController also keeps track of the MemoryCaller for easily managing the requests and responses.

Sequence Diagram name: SUB_SYS_HL_SD_002_02

Figure 11: Sub-System High-Level Sequence Diagram 002_02

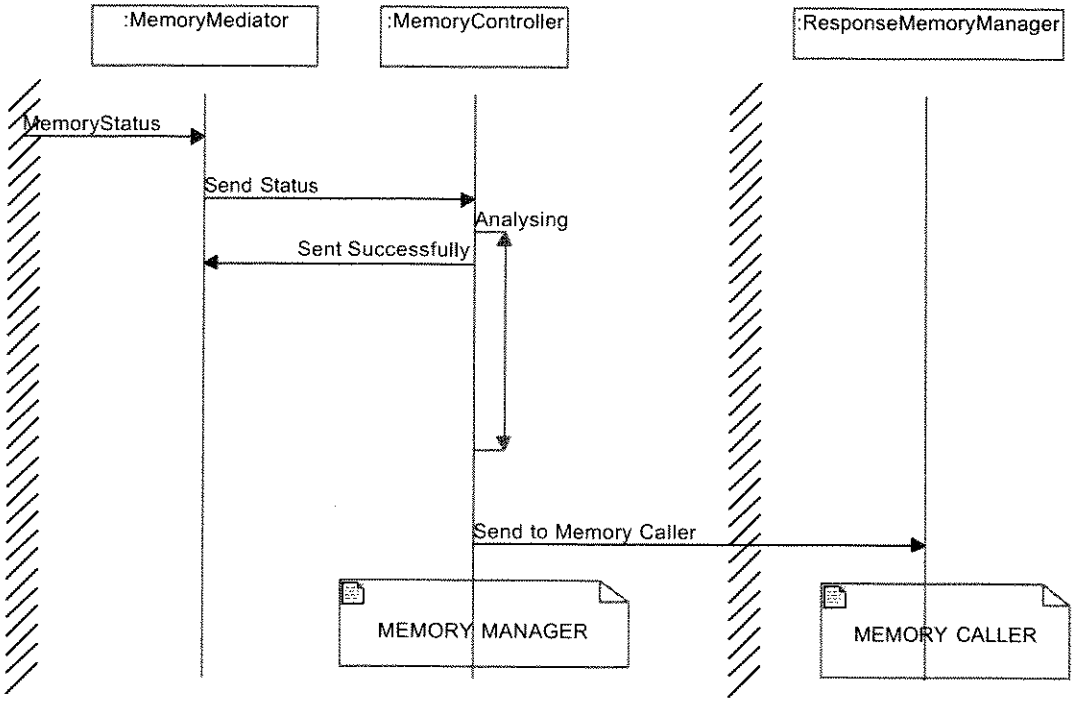


Figure 11, is a High-Level Sequence Diagram showing the response from the MemoryManager. The response is sent after processing of the methods to the MemoryMediator class. The MemoryMediator class has knowledge of the outer world, which is segregated from MemoryProcessing. The status is sent to the MemoryController, which analyzes the response and matches the right response, with the request sent by the MemoryCaller. The response is sent to the MemoryCaller as soon as the MemoryController identifies the correct MemoryCaller.

Low-Level Use Case Diagram

The Low-Level Use Cases are derived Use Cases and only provide in-depth requirements for the MemoryManager. MemoryManager is the prime component of the model and more in detail information is provided on its internal functionality. The Low-Level detailed information for the MemoryCaller and the OperatingSystem is not in the scope of this essay. The following detailed Low-Level Use Cases are Low-Level artifacts and become the part of the SDD.

Use Case Diagram name: MM_LL_UC_001

Figure 12: MemoryManager Low-Level Use Case 001

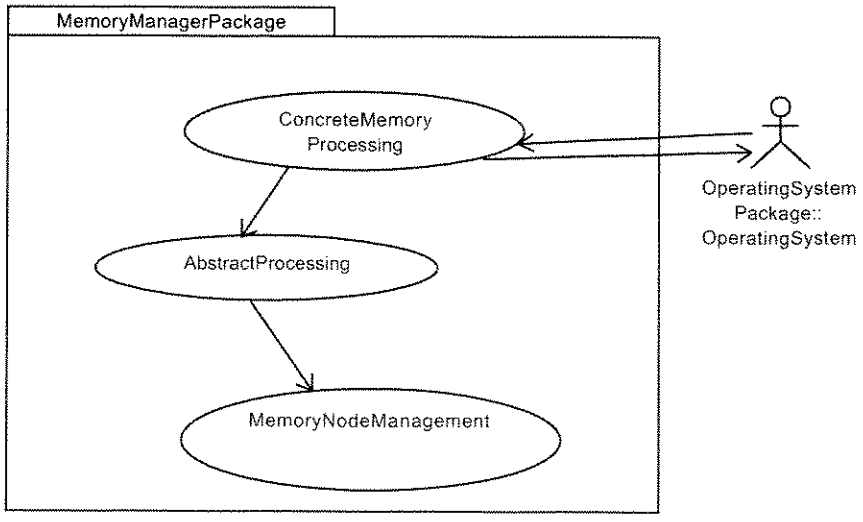


Figure 12, is a Low-Level Use Case Sequence Diagram for the MemoryManager. It illustrates the initialization of the heap by depicting the design within the MemoryManagerPackage. The three Use Cases in the diagram are ConcreteMemoryProcessing, AbstractProcessing and MemoryNodeManagement. The

actor is the OperatingSystemPackage and the MemoryManagerPackage sets the system boundary. A size of the heap is passed to the ConcreteMemoryProcessing, and then the size is further passed and managed by the AbstractProcessing. The size along with a pre-defined value shared between the OperatingSystem and the MemoryManager is used to determine the fixed size of the block. The fixed size is used to create initial nodes. MemoryNodeManagement uses a methodology to keep track of the size of the nodes and a methodology to traverse the nodes.

Use Case Diagram name: MM_LL_UC_002

Figure 13: MemoryManager Low-Level Use Case 002

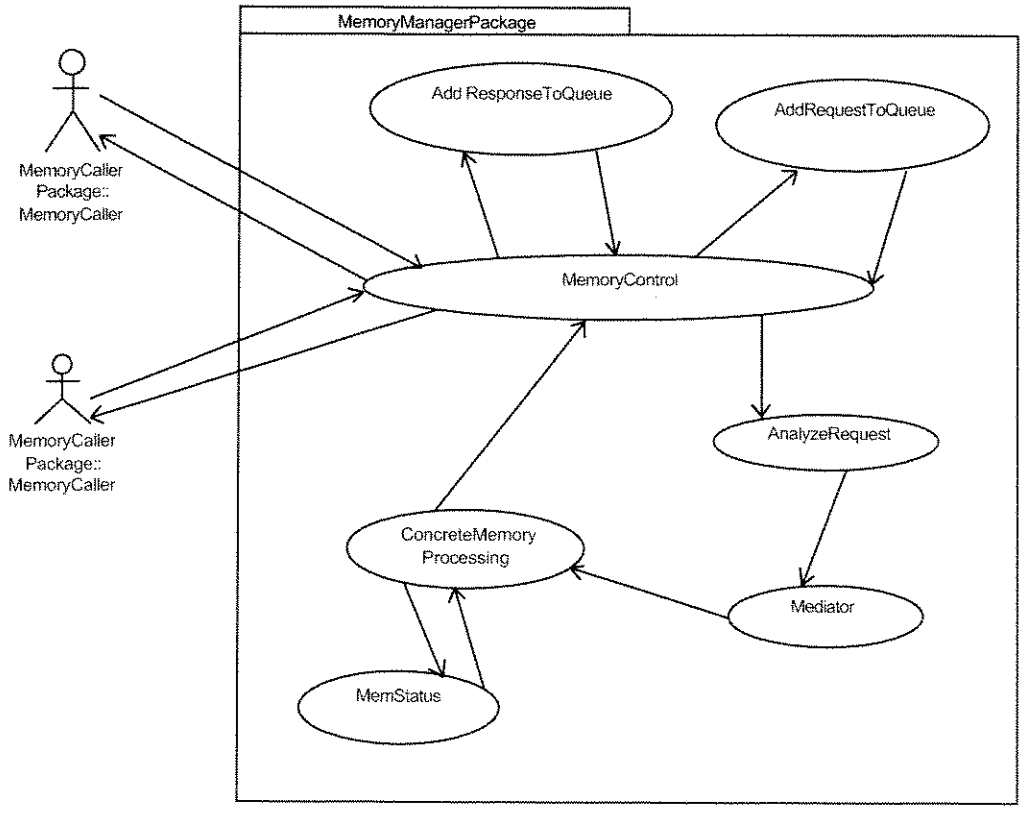


Figure 13, is a Low-Level Use Case for the MemoryManagerPackage. The actor is the MemoryCaller and MemoryManagerPackage sets the system boundary. The diagram shows two MemoryCallers illustrating different MemoryCallers. The MemoryCaller sends a request to MemoryControl, which checks the request for integrity and adds it to the request queue. The controller does the logic and after the conditions are met, it removes the request from the queue and analyses the request. The analysis is sent to the Mediator, which further invokes the appropriate methods. After processing of the method, a status is sent back to the MemoryControl, which is stored in the response queue and eventually returned back to the MemoryCaller. The MemoryControl also keeps track of the MemoryCallers by storing the respective identification number. This number is checked when the response is sent back to the respective MemoryCaller.

Low-Level Sequence Diagram

The Low-Level Sequence Diagram provide a detailed interaction between the internal classes of the MemoryManager component. The following Low-Level artifacts become the part of the SDD.

Sequence Diagram name: MM_LL_SD_001

Figure 14: MemoryManager Low-Level Sequence Diagram 001

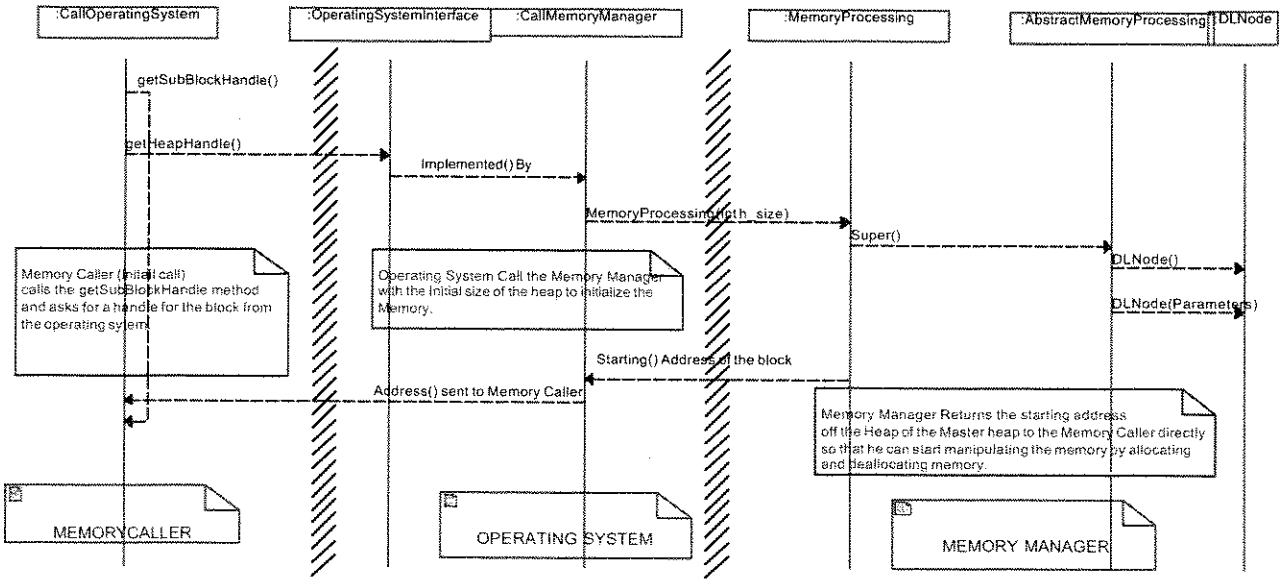


Figure 14, is a Low-Level Sequence Diagram for the MemoryManager. MemoryCaller asks for a handle for the block from the OperatingSystem. OperatingSystem calls the MemoryManager with the initial size of the heap to initialize the memory. MemoryManager returns the starting address of the block of the master heap to the MemoryCaller, which uses the block of memory for its operations. The diagram illustrates message passing and the creation of a block during the initialization of the heap. The OperatingSystem creates a new instance of MemoryProcessing with a parameter expressing the size of the master heap. The AbstractMemoryProcessing class instantiates a specialized class DLNode, that creates the node structure. The AbstractMemoryProcessing takes the maximum size of the heap given by the OperatingSystem and generates a fixed block size for that heap. The MemoryManager is made deterministic by using the pre-defined memory requirement of MemoryCallers in a configuration file. The memory requirement for a set of deterministic MemoryCallers is

stored in the configuration file. This file is shared between the MemoryManager, the MemoryCaller and the OperatingSystem. The values stored in the configuration file and size of the master heap is used by the AbstractMemoryProcessing to determine the optimum block size, which will satisfy all the memory needs of the MemoryCallers.

Sequence Diagram name: MM_LL_SD_002

Figure 15: MemoryManager Low-Level Sequence Diagram 002

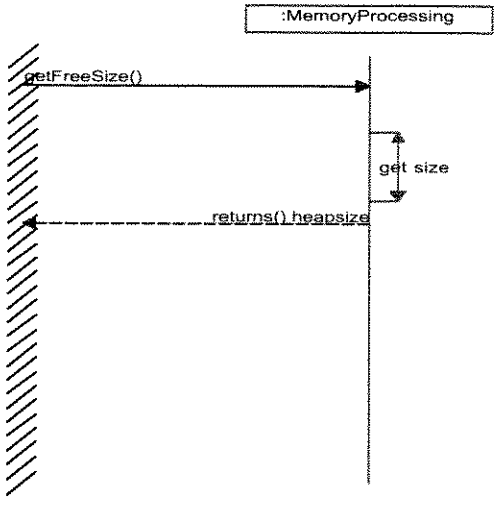


Figure 15, is a Low-Level Sequence Diagram for the MemoryManager to illustrate how the MemoryManager request for free size. The amount of memory not used (free size) is returned to the MemoryCaller once it queries the MemoryManager.

Sequence Diagram name: MM_LL_SD_003

Figure 16: MemoryManager Low-Level Sequence Diagram 003

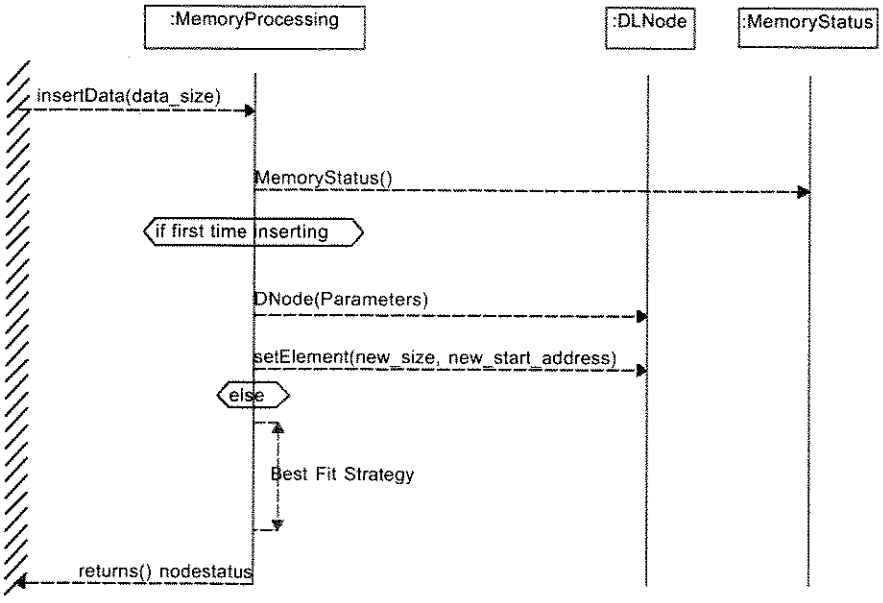


Figure 16, is a Low-Level Sequence Diagram for MemoryManager illustrating the insertion of data. The request of inserting data is made by calling the insertData() method. The AbstractMemoryProcessing is an abstract class, which defines the method but it is implemented in the MemoryProcessing class. If there is no data in the memory a new instance of a node is created and data is inserted by calling the setElement() method of the DLNode. If there is data, then the best-fit strategy is used to search for the location to insert the data. The strategy finds the best possible size to meet the requirements of the MemoryCaller. The status of the node, where the data is inserted is returned back to the MemoryCaller.

Sequence Diagram name: MM_LL_SD_004

Figure 17: MemoryManager Low-Level Sequence Diagram 004

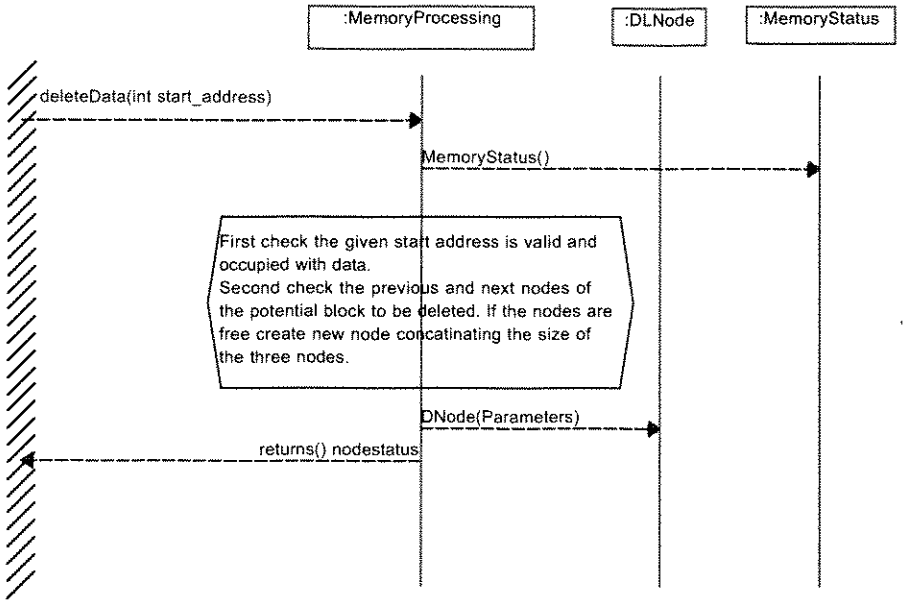


Figure 17, is a Low-Level Sequence Diagram for MemoryManager. The diagram illustrates the deletion of the memory from the block. The deletion request is made by the MemoryCaller with the starting address of the node to be deleted. The node is first validated to see if it has data and if its address is correct. The previous and the next nodes of the block to be deleted are checked to see if they have data in them. If they have data, then only a new node is created for the same size of the node to be deleted and the address is assigned to that node. If the previous and next blocks are empty then a new node is created with the size of the all three (previous node, node to be deleted and next node). The new concatenated node is assigned the address and the status is set to free.

Sequence Diagram name: MM_LL_SD_005

Figure 18: MemoryManager Low-Level Sequence Diagram 005

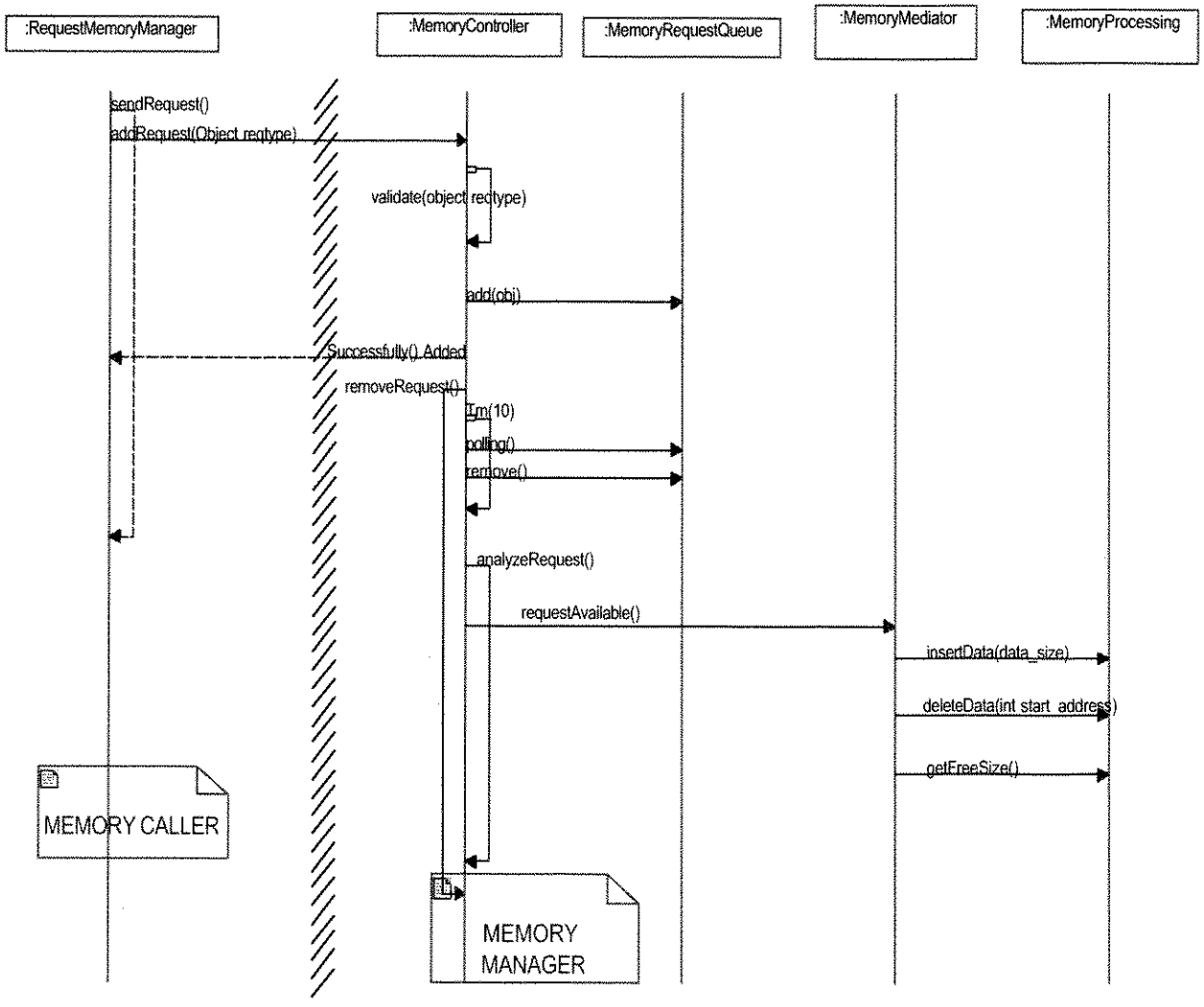


Figure 18, is a Low-Level Sequence Diagram. It depicts how a request from the MemoryCaller is made for the operations available in the interface of the MemoryManager. The sendRequest() method in the MemoryCaller calls the addRequest() method from the MemoryController to add the request onto the queue. The

request from MemoryCaller is validated by a MemoryController and added to the request queue. The parameter in the request is one of the methods available via the MemoryManagerInterface. MemoryController polls and removes the request from the queue in the removeRequest() method. If the request is found on the queue, it is removed. Otherwise, a timeout occurs. The MemoryCaller removes the request from the queue, analyses the sender of the request and passes the request to the MemoryMediator class. The MemoryMediator decomposes the request and invokes the correct method. This Sequence Diagram shows all the three possible method calls. The MemoryController is responsible for mapping the correct MemoryCaller with its request. It stores the identification number of the MemoryCaller, which is used when response is sent to the MemoryCaller.

Sequence Diagram name: MM_LL_SD_006

Figure 19: MemoryManager Low-Level Sequence Diagram 006

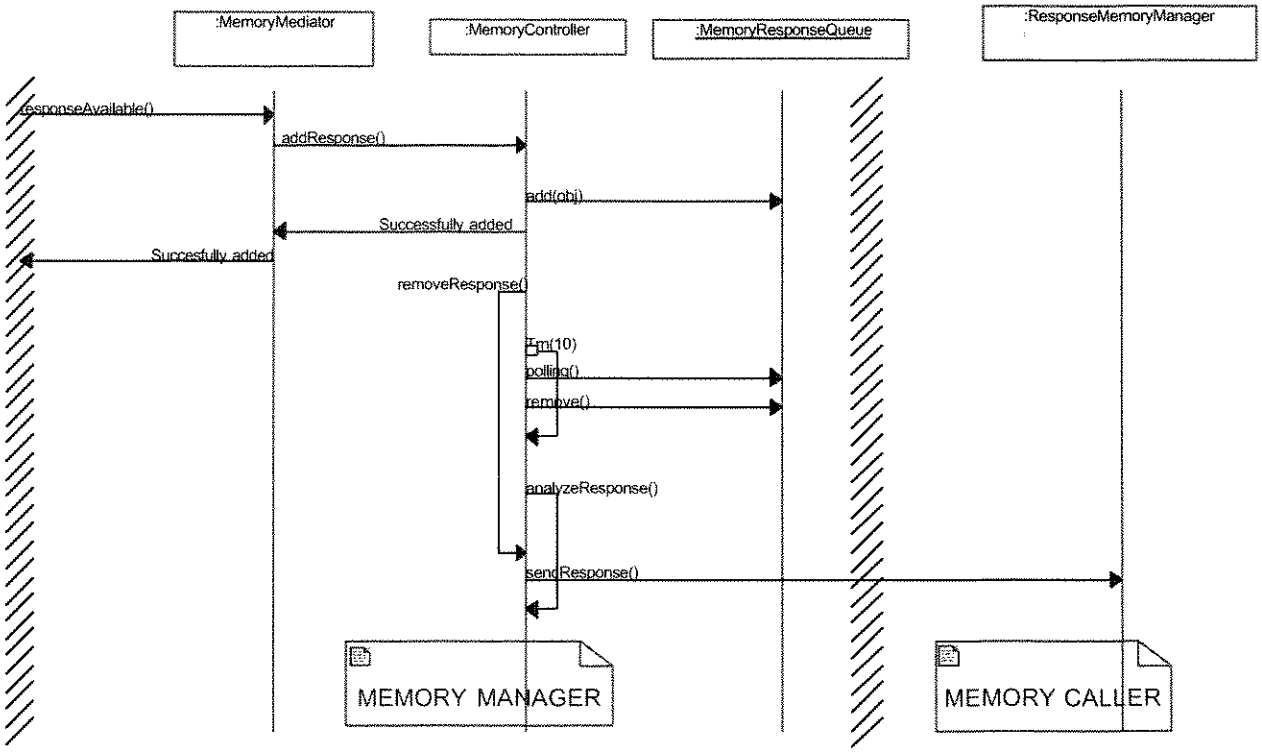


Figure 19, is a Low-Level Sequence Diagram showing the response from the MemoryManager. The addResponse() method of MemoryController is called by MemoryMediator and the status of inserting or deleting data is sent to the MemoryController. The MemoryMediator receives the response via the responseAvailable() method, which is called by the MemoryProcessing. MemoryController adds the data in to the response queue. The MemoryCaller polls and removes the response from the queue in the removeResponse() method. If there is data on the queue, the data is removed. Otherwise, a timeout occurs. The MemoryController is responsible for mapping the right MemoryCaller with its response. The identification number of the request stored when the request is made is validated with the response. The

mapping is checked by the analyzeResponse() method. The message is sent to the MemoryCaller by calling the sendResponse() method of the MemoryCaller.

Object Model Diagram

The Object Model Diagram for the components is a Low-Level artifact and provides an overview on how the classes interact with each other in the components. The following Low-Level artifacts become the part of the SDD.

Object Model Diagram name: MC_OMD_001

Figure 20: MemoryCaller Object Model Diagram 001

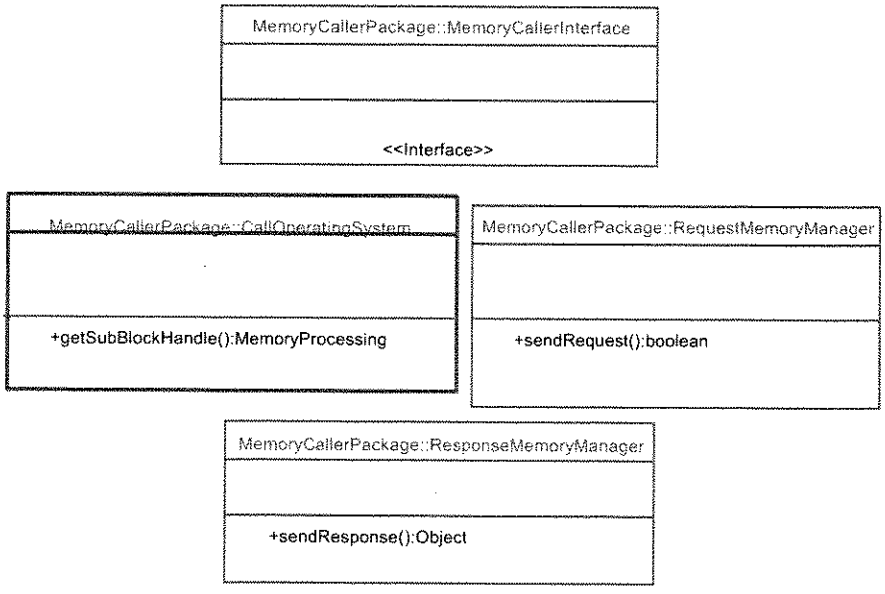


Figure 20, depicts the MemoryCaller component which calls for the services offered by the MemoryManager and the OperatingSystem. The purpose of the following Object Model Diagram is to display the classes and methods, which are used for interaction with

MemoryManager and OperatingSystem. The design of this component is not developed in the sample model.

Object Model Diagram name: OS_OMD_001

Figure 21: OperatingSystem Object Model Diagram 001

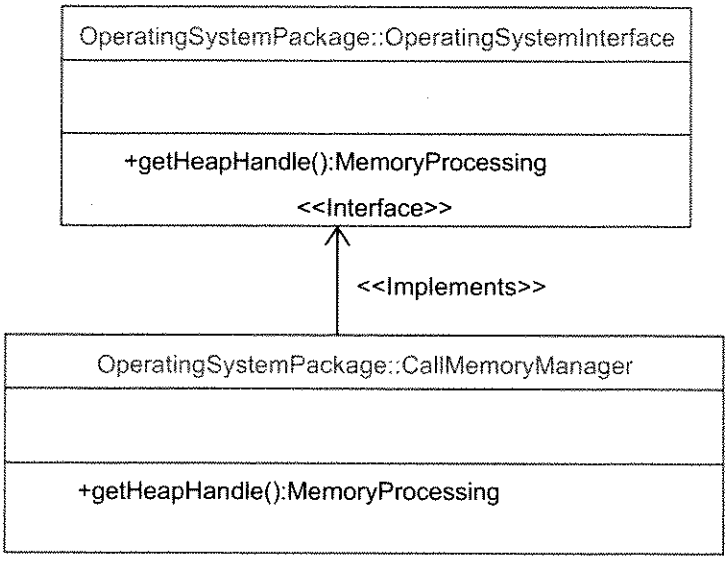


Figure 21, depicts the OperatingSystem entity. The OperatingSystemInterface defines the methods, which are available to the MemoryCaller and MemoryManager. The CallMemoryManager class implements the interface, which calls the initialization of the MemoryManager. The Object Model Diagram shows very minimal operations of the OperatingSystem as stated in the assumption in Chapter I. The design of this component is not developed in the sample model.

Object Model Diagram name: MM_OMD_001

Figure 22: MemoryManager Object Model Diagram 001

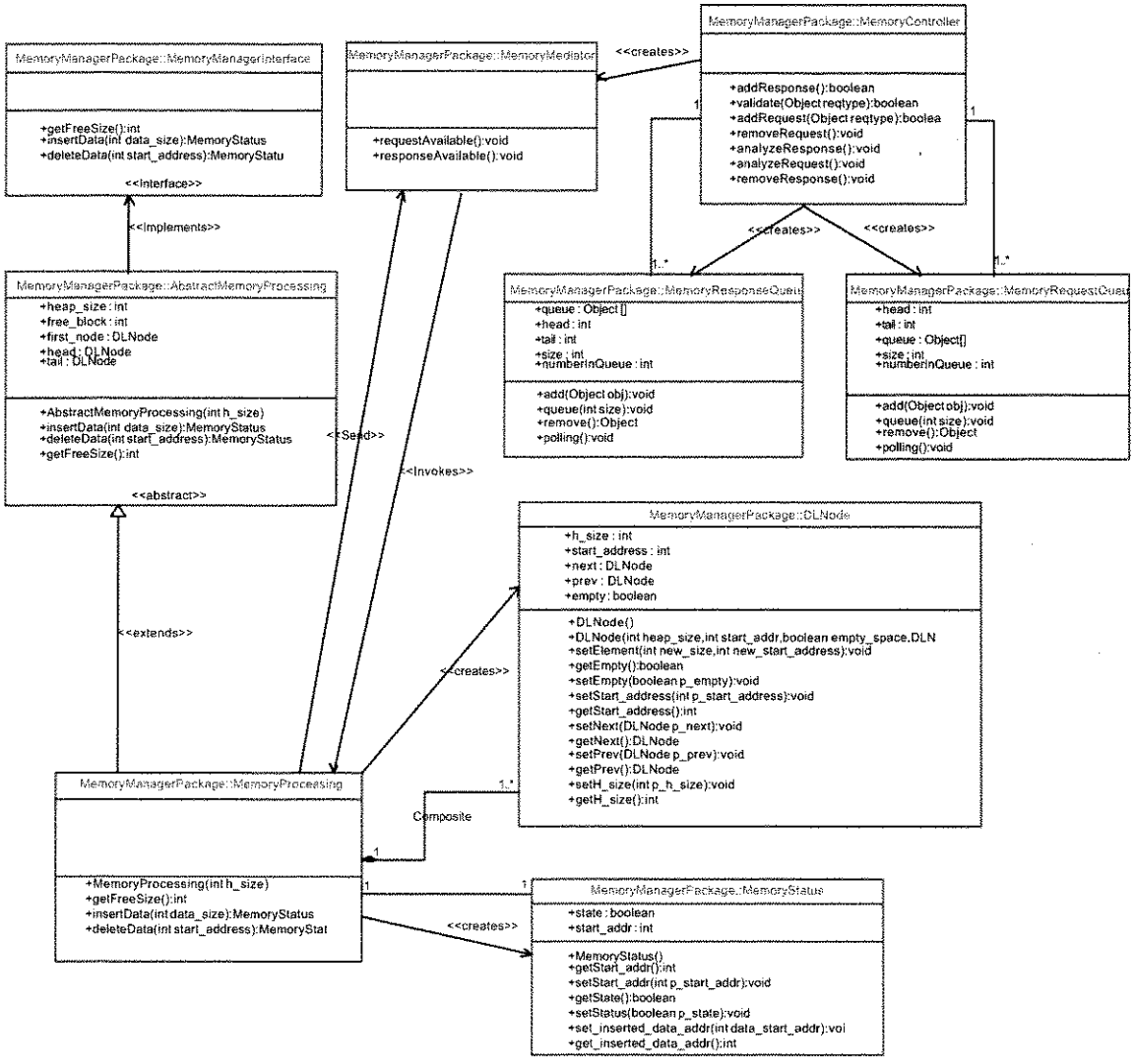


Figure 22, depicts the MemoryManager entity, which is the principal component of the sample model. The interaction and dependencies between the internal classes are illustrated in Figure 22. The stereotype <creates> depicts the creation of a new instance

of the classes. The dotted lines in Figure 22, which end touching a class with an arrow, depict the instantiation of that class, by the class from where the line starts. The <extends> stereotype illustrates the extension of the class. The <sends> stereotype shows the message, which are sent from one class to another. This is explicitly shown in the diagram between the MemoryProcessing class and the MemoryMediator class. The MemoryMediator is the key class, which binds the messages between the MemoryController and MemoryProcessing. MemoryController can have one or more request/response queues, but the queues just have one controller depicted as the MemoryController. MemoryPocessing can have only one MemoryStatus per node as seen by the one-to-one relationship between them. There is a composite relationship between the MemoryProcessing and DLNode class. MemoryProcessing can have one or many DLNode. In the "whole/part relationship" the whole of the composite relationship is MemoryProcessing and the part is DLNode (Hoffer, Prescott & McFadden, 2002, p.545). The assessors and modifiers of the classes are used to perform check, traversal, retrieval and updates on the attributes of the classes. The Object Model Diagram for MemoryManager component is developed using Design Patterns to create an optimized design and increase the usability and flexibility. The design patterns used in the model are defined in Chapter IV.

Traceability Matrix

Table 7 below, demonstrates a High-Level traceability matrix for the artifacts of the sample component. The High-Level Use Case (HL_UC) can be traced vertically on the rows to the High-Level Sequence Diagram (HL_SD), Low-Level Use Case (LL_UC), Low-Level Sequence Diagram (LL_SD) and the Object Model Diagram OMD).

Table 7: Traceability Matrix

HL-UC	HL-SD	LL-UC	LL-SD	OMD
SUB_SYS_HL_UC_001	SUB_SYS_HL_SD_001	MM_LL_UC_001	MM_LL_SD_001 MM_LL_SD_005 MM_LL_SD_006	OS_OMD_001
SUB_SYS_HL_UC_002	SUB_SYS_HL_SD_001	MM_LL_UC_001	MM_LL_SD_001 MM_LL_SD_005 MM_LL_SD_006	MM_OMD_001
SUB_SYS_HL_UC_003	SUB_SYS_HL_SD_002_01 SUB_SYS_HL_SD_002_02	MM_LL_UC_002	MM_LL_SD_002	MM_OMD_001
SUB_SYS_HL_UC_004	SUB_SYS_HL_SD_002_01 SUB_SYS_HL_SD_002_02	MM_LL_UC_002	MM_LL_SD_003	MM_OMD_001
SUB_SYS_HL_UC_005	SUB_SYS_HL_SD_002_01 SUB_SYS_HL_SD_002_02	MM_LL_UC_002	MM_LL_SD_004	MM_OMD_001

Figure 23: Static Instance Traceability

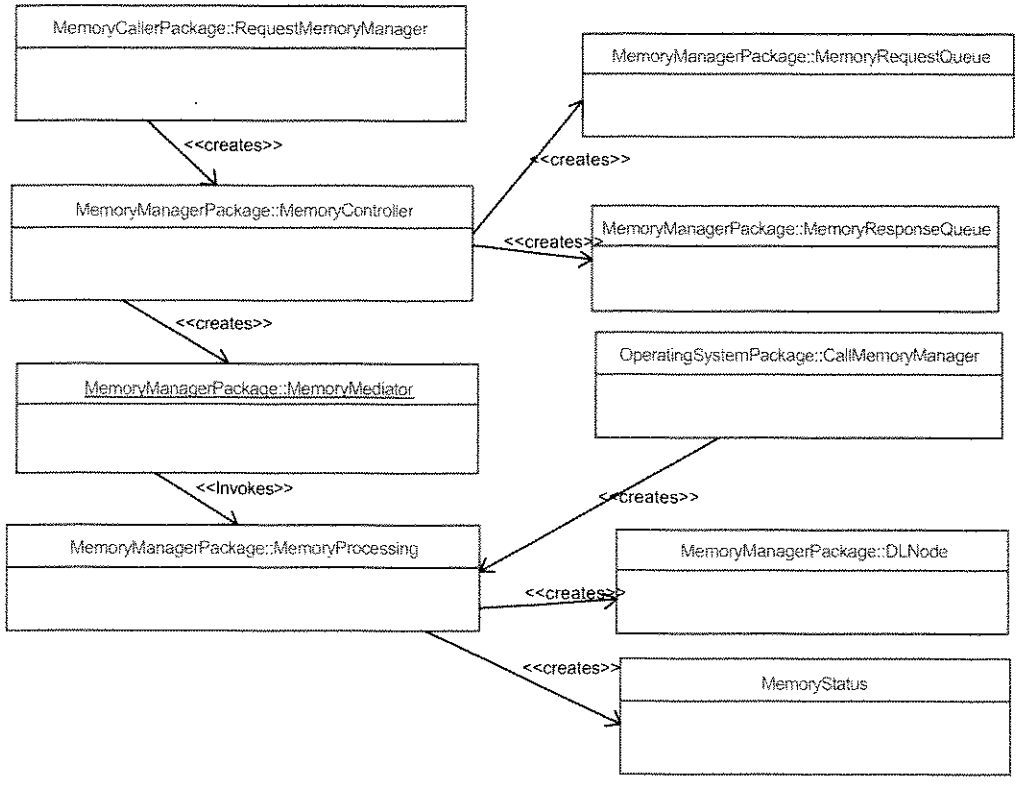


Figure 23 above, shows the static instance traceability for one of the High-Level Use Cases SUB_SYS_HL_UC_004, which is traced to the MM_OMD_001. This traceability instance is a Low-Level artifact and becomes the part of the SDD. The Use Case depicts the command from MemoryCaller to insert data into the memory block and the response of the MemoryStatus, which is given back to the MemoryCaller. The internal collaboration of object helps to process the request from the MemoryCaller. The <<create>> stereotype illustrates all the instances which are created in order to insert data into the memory block in the MemoryManager. The MemoryCaller creates the first instance when the request is sent to the MemoryController. MemoryController creates two instances for queue (request and response) to manage the request from the caller and response back to the callers. MemoryController creates an instance of MemoryMediator informing him that the request is available. MemoryMediator invokes the method from MemoryProcessing. MemoryProcessing create a new instance of DLNode to insert the data and also an instance of MemoryStatus to hold the status of the node. MemoryStatus is sent back to the MemoryCaller via the existing objects.

In the above methodology, a detail description of the sample component is provided. MemoryManager, the primary component of the model is designed as a RSC using a set of requirements. The High-Level and Low-Level requirements are defined and Design Patterns are used to create the design for the robust and reusable component. The traceability matrix is explained in the methodology. The Appendix references for the detailed information about the package are specified.

CHAPTER IV

RESULTS

The design for the MemoryManager component of the sample model illustrates the use of Design Patterns to create an RSC. All three types of Design Patterns: creational, structural, and behavioural are used for the design of the MemoryManager. The Factory Method Pattern and the Abstract Factory Pattern are the creational patterns, which create objects in the MemoryManager component. The Facade and Composite are structural patterns which organize the objects into groups and the Chain of Responsibility, Mediator, Momento, and Iterator are behavioural patterns that allow a means of communications between the objects of the MemoryManager component. The Fixed Sized Buffer pattern is used to create equal size of memory blocks. The guidance rules specified by the OOTiA Handbook, which pertain to the planning and the design phase, are considered throughout the development of the sample component. The compliance for the rules is seen as the results of sample component.

Memory Manager and Design Patterns

Factory Method Pattern

The main objective of Factory Method Pattern is to "define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses"(Gamma et al., 1995, p. 107). The Object Model Diagram (Figure 22: MM_OMD_001) for the MemoryManager component in Chapter III illustrates the use of this pattern in the design. The classes considered in the design when using this pattern are:

- CallMemoryManager: Part of the OperatingSystemPackage.
- AbstractMemoryProcessing: Part of the MemoryManagerPackage
- MemoryProcessing: Part of the MemoryManagerPackage
- DLNode: Part of the MemoryManagerPackage

The description of the classes is defined in Appendix E and F.

Figure 24: Factory Method Pattern Analysis

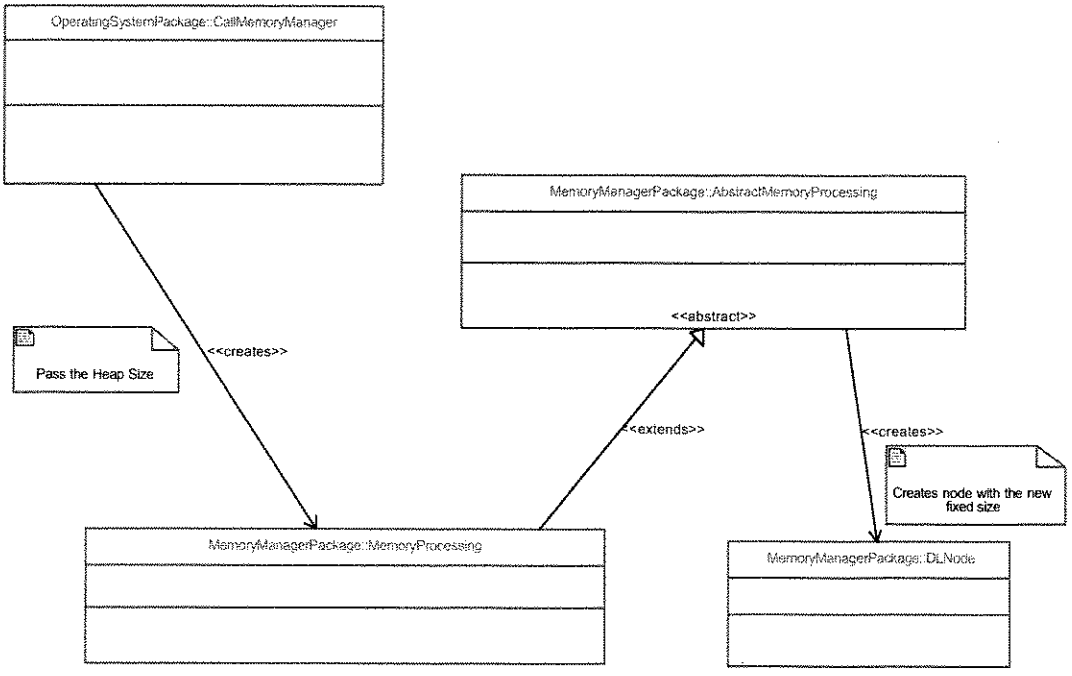


Figure 24 above, shows the use of Factory Method Pattern in the MemoryManager component. The base class in the MemoryManager component is AbstractMemoryProcessing, which is an abstract class defining the signature of the

memory processing operations. The MemoryProcessing class is instantiated by the OperatingSystem acting as a concrete creator. The creator calling the MemoryManager component needs to have the knowledge of the MemoryProcessing subclass for instantiation. The concrete MemoryProcessing class extends the AbstractMemoryProcessing class and passes it a parameter of heap size provided by the creator. The AbstractMemoryProcessing class in its constructor creates an instance of DLNode class for initializing the heap of memory with a fixed block size. MemoryProcessing subclass makes the decision of instantiating DLNode class by passing the parameter (***heap_size***) to the constructor of AbstractMemoryProcessing.

Abstract Factory Pattern

The main objective of Abstract Factory Pattern is to "provide an interface for creating families of related or dependent objects without specifying their concrete classes" (Gamma et al., 1995, p. 87). The Object Model Diagram (Figure 22: MM_OMD_001) for the MemoryManager component in Chapter III illustrates the use of this pattern in the design. The classes considered in the design when using this pattern are:

- MemoryManagerInterface: Part of the MemoryManagerPackage
- AbstractMemoryProcessing: Part of the MemoryManagerPackage
- MemoryProcessing: Part of the MemoryManagerPackage
- MemoryStatus: Part of the MemoryManagerPackage

The description of the classes is defined in Appendix E.

Figure 25: Abstract Factory Pattern Analysis

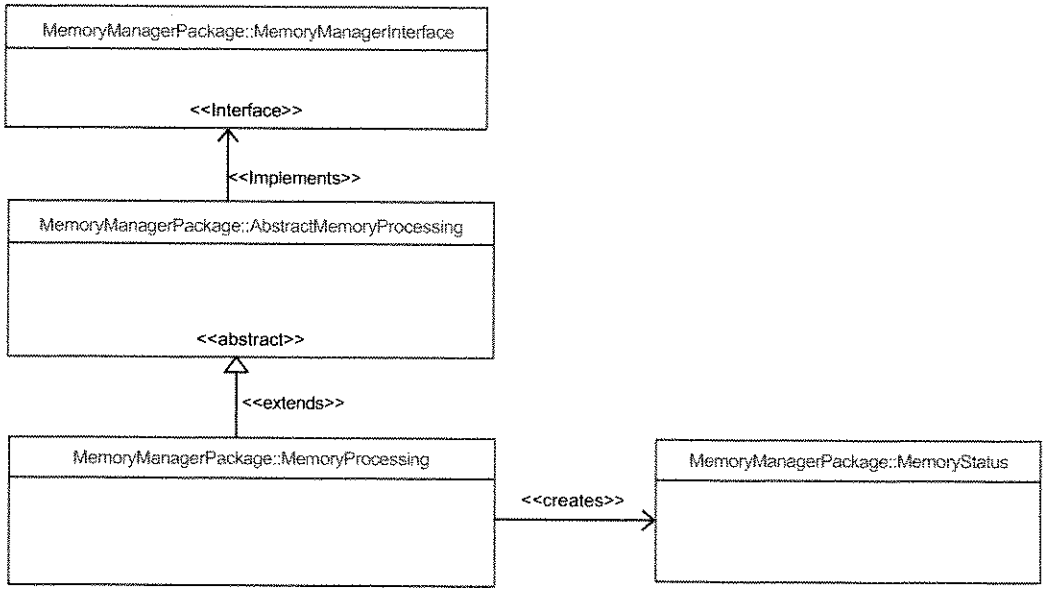


Figure 25 above, shows the use of Abstract Factory Pattern in the MemoryManager component. The MemoryManagerInterface provides the insert, delete, and getfreesize methods. AbstractMemoryProcessing is the Abstract Factory implementing the interface. MemoryProcessing is the concrete class which implements the methods from the abstract class and returns the MemoryStatus for inserting or deleting data. MemoryStatus class returns the status and start address of the block. The related objects created in the MemoryManager component service the requests from the MemoryCaller without the caller knowing the details of the concrete MemoryProcessing class implementation.

Adapter Pattern

The following pattern is not used in the MemoryManager component, but the approach of its use is illustrated because of the benefit of interacting with incompatible

interfaces. Adapter Pattern is primarily used in "systems after they are designed" to create a means of communication (Gamma et al., 1995, p. 161).

The objective of the Adapter pattern is to "convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces" (Gamma et al., 1995, p.139).

One approach of using the Adapter Pattern in the MemoryManager component is by creating an interface with additional primitives respective to the calling OperatingSystem. The enhanced interface acts as the Adapter to the adaptee (OperatingSystem) and the target (MemoryManager). MemoryManager with the aid of Adapter can interact with other components increasing its reusability.

Composite Pattern

The objective of the Composite Pattern is to "compose objects into tree structures to represent part whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly" (Gamma et al., 1995, p.163). The Object Model Diagram (Figure 22: MM_OMD_001) for the MemoryManager component in Chapter III illustrates the use of this pattern in the design. The classes considered in the design when using this pattern are:

- AbstractMemoryProcessing: Part of the MemoryManagerPackage
- MemoryProcessing: Part of the MemoryManagerPackage
- DLNode: Part of the MemoryManagerPackage

The description of the classes is defined in Appendix E.

Figure 26: Composite Pattern Analysis

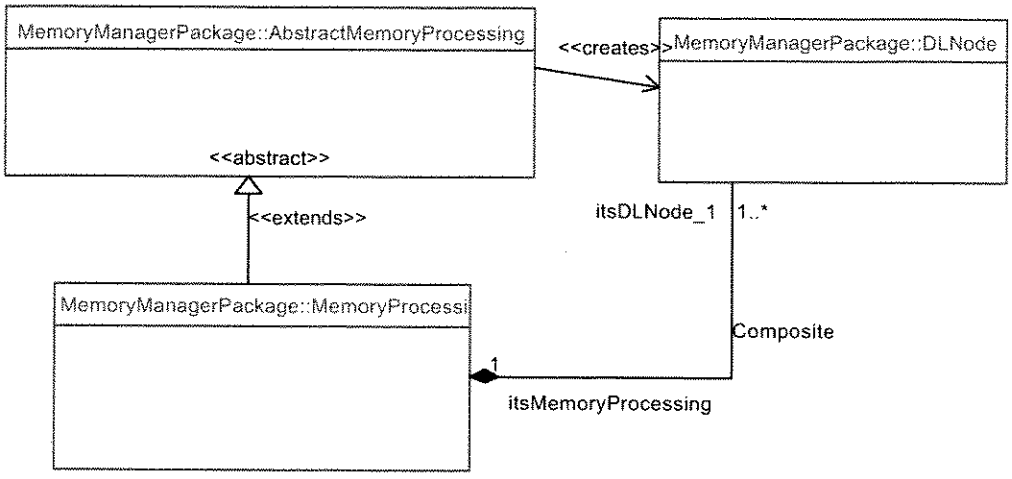


Figure 26 above, shows the use of Composite Pattern in the MemoryManager component. The composite pattern is used between the MemoryProcessing class and the DLNode class. In the "whole/part relationship" between the two classes, the whole is the MemoryProcessing, and the part is the DLNode (Hoffer et al., 2002, p. 545). The nodes in the DLNode class belong to only one MemoryProcessing and "live and die" with it (Hoffer et al., 2002, p. 544). AbstractMemoryProcessing creates the initial block. MemoryProcessing class creates the new blocks as the data is inserted and deleted on request from the MemoryCaller.

Facade Pattern

The objective of the Facade Pattern is to "provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher level interface that makes a subsystem

easier to use" (Gamma et al., 1995, p.185). The Object Model Diagram (Figure 22: MM_OMD_001) for the MemoryManager component in Chapter III illustrates the use of this pattern in the design. The classes considered in the design when using this pattern are:

- RequestMemoryManager: Part of the MemoryCallerPackage
- MemoryManagerInterface: Part of the MemoryManagerPackage
- MemoryController: Part of the MemoryManagerPackage

The description of the classes is defined in Appendix D and E.

Figure 27: Facade Pattern Analysis

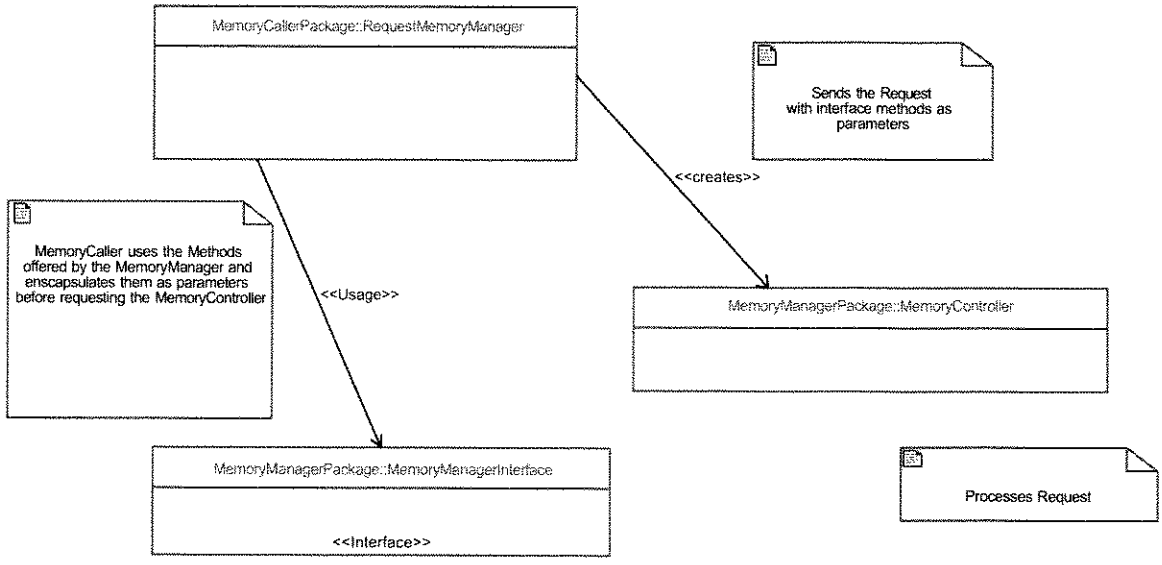


Figure 27 above, shows the use of Facade Pattern in the MemoryManager component. The MemoryManager component uses a variance of Facade pattern and provides a

simplified interface to the MemoryCaller for inserting, deleting and querying data. The complexity of the methods provided by the interface is hidden from the MemoryCaller. The status of the insertion or deletion is returned back to the MemoryCaller as MemoryStatus. The details of the insertion and deletion methods are hidden from the client caller. By applying this pattern, the coupling between the MemoryCaller and MemoryManager is reduced making the MemoryManager component more flexible and reusable.

Chain of Responsibility

The objective of this pattern "is to avoid coupling the sender of a request to its receiver by giving more than one object to handle the request. Chain the receiving object and pass the request along the chain until an object handles it" (Gamma et al., 1995, p.223). The Object Model Diagram (Figure 22: MM_OMD_001) for the MemoryManager component in Chapter III illustrates the use of this pattern in the design. The classes considered in the design when using this pattern are:

- RequestMemoryManager: Part of the MemoryCallerPackage
- MemoryRequestQueue: Part of the MemoryManagerPackage
- MemoryController: Part of the MemoryManagerPackage
- MemoryMediator: Part of the MemoryManagerPackage
- MemoryResponseQueue: Part of the MemoryManagerPackage
- MemoryProcessing: Part of the MemoryManagerPackage

The description of the classes is defined in Appendix D and E.

Figure 28: Chain of Responsibility Analysis

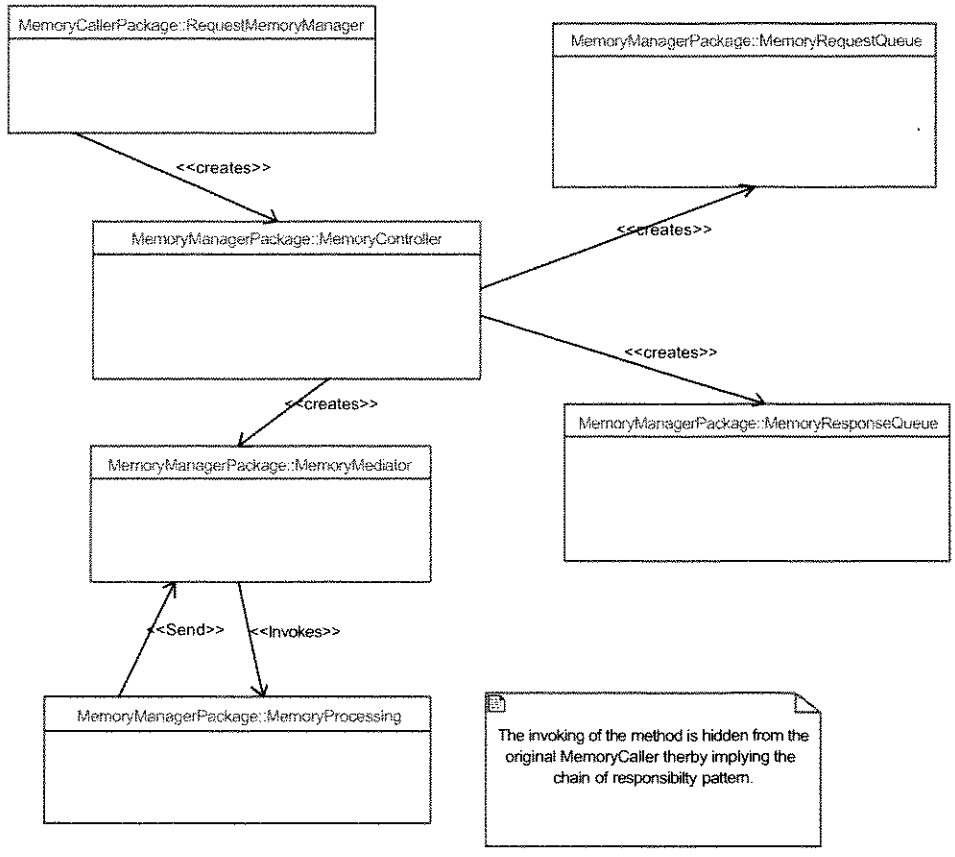


Figure 28 above, shows the use of Chain of Responsibility Pattern in the MemoryManager component. The sender of the request is MemoryCaller and the receiver of the request is MemoryProcessing class. There are many objects instantiated from the time the request is made from MemoryCaller to the time it is processed by MemoryProcessing. The objects created by the MemoryController collaborate to handle the request and invoke respective methods thereby applying the Chain of Responsibility Pattern.

Command Pattern

The objective of this pattern "is to encapsulate a request as an object thereby letting you parameterize clients with different request queues or lock request and support undoable operations" (Gamma et al., 1995, p.233). The Object Model Diagram (Figure 22: MM_OMD_001) for the MemoryManager component in Chapter III illustrates the use of this pattern in the design. The classes considered in the design when using this pattern are:

- CallMemoryManager: Part of the OperatingSystemPackage
- MemoryProcessing: Part of the MemoryManagerPackage
- AbstractMemoryProcessing: Part of the MemoryManagerPackage
- DLNode: Part of the MemoryManagerPackage

The description of the classes is defined in Appendix E and F.

Figure 29: Command Pattern Analysis

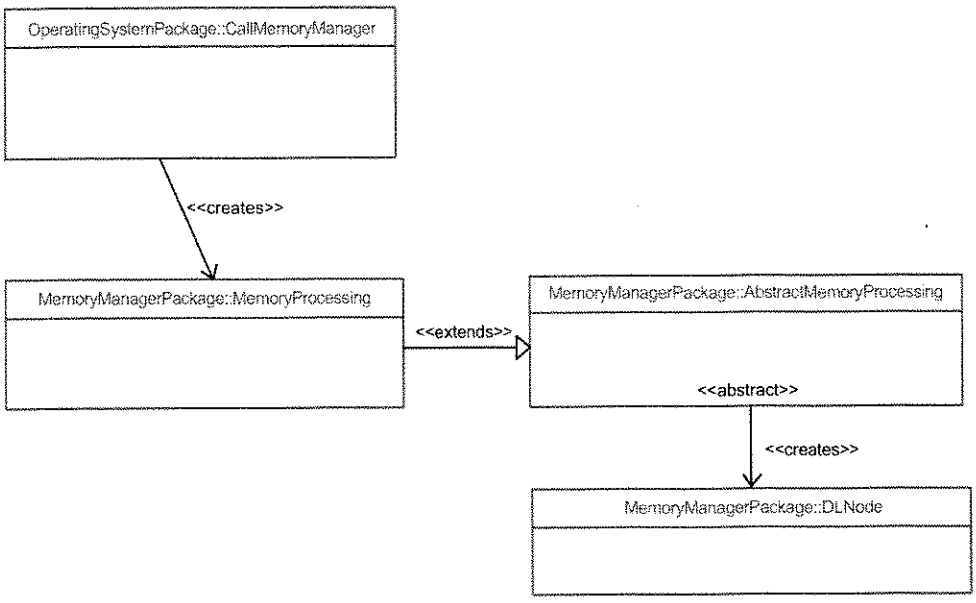


Figure 29 above, shows the use of Command Pattern in the MemoryManager component. This pattern is used when the OperatingSystem invokes the MemoryProcessing with the initial size of the heap as a parameter. The parameter is passed to the abstract class by explicitly calling super() in the constructor. The fixedblocksize() method in the AbstractMemoryProcessing class takes the size provided by the OperatingSystem and generates a fixed size for the block. The DLNode class uses this size to create the heap structure. The execution command used in MemoryManager is the call to fixedblocksize() method in the AbstractMemoryProcessing.

In another instance of usage of this pattern in the MemoryManager component the request for the method by MemoryCaller is encapsulated as an object and sent to the MemoryController. The object holds the MemoryCallers identification and requests for

services. MemoryController analyzes the request, stores the identification of the MemoryCaller and passes the service request to the MemoryMediator. MemoryMediator decomposes the service request and call the respective command. After processing the response encapsulated as a MemoryStatus object is sent back to MemoryCaller via the MemoryMediator and MemoryController.

Iterator

The objective of this pattern "is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying information" (Gamma et al., 1995 p.257). The Object Model Diagram (Figure 22: MM_OMD_001) for the MemoryManager component in Chapter III illustrates the use of this pattern in the design. The classes considered in the design when using this pattern are:

- MemoryProcessing: Part of the MemoryManagerPackage
- DLNode: Part of the MemoryManagerPackage

The description of the classes is defined in Appendix E.

This pattern is used between MemoryProcessing and DLNode class. Figure 26, illustrates the composite relationship between these classes. The composite objects of DLNode are accessed without exposing the internal details of the object via the assessor. Another approach of using this pattern in the MemoryManager component is the traversal of the queue for handling the requests and responses.

Momento

The objective of the following pattern is to "without violating encapsulation capture and externalize an object's internal state so that the object can be restored to this state later" (Gamma et al., 1995, p. 283). The Object Model Diagram (Figure 22: MM_OMD_001) for the MemoryManager component in Chapter III illustrates the use of this pattern in the design. The classes considered in the design when using this pattern are:

- RequestMemoryManager: Part of the MemoryCallerPackage
- ResponseMemoryManager: Part of the MemoryCallerPackage
- MemoryController: Part of the MemoryManagerPackage
- MemoryMediator: Part of the MemoryManagerPackage
- MemoryProcessing: Part of the MemoryManagerPackage
- AbstractMemoryProcessing: Part of the MemoryManagerPackage
- MemoryStatus: Part of the MemoryManagerPackage

The description of the classes is defined in Appendix D and E.

Figure 30: Momento Pattern Analysis

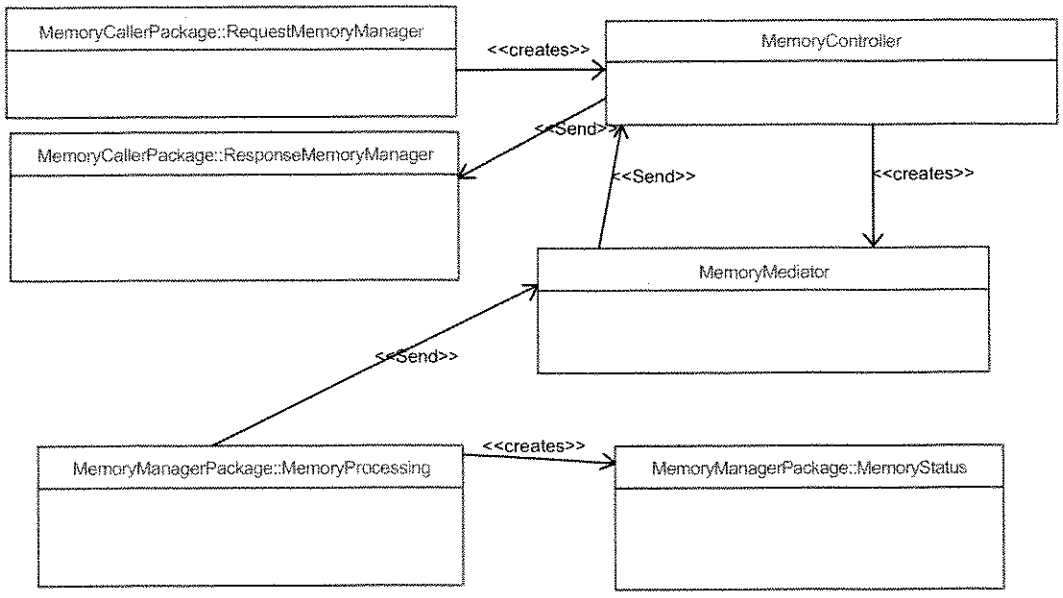


Figure 30 above, shows the use of Momento Pattern in the MemoryManager component. The MemoryManager component uses a variant of this pattern. The MemoryStatus class captures the status, as the data in the nodes is inserted; the state of the node is set. The MemoryProcessing class is the only authoritative class accessing and modifying the state of the nodes, which is further passed to the MemoryCaller.

Mediator

The objective of the Mediator Pattern is to "define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping object from referring to each other explicitly, and it lets you vary their interaction independently" (Gamma et al., 1995, p.273). The Object Model Diagram (Figure 22: MM_OMD_001)

for the MemoryManager component in Chapter III illustrates the use of this pattern in the design. The classes considered in the design when using this pattern are:

- MemoryController: Part of the MemoryManagerPackage
- MemoryMediator: Part of the MemoryManagerPackage
- MemoryProcessing: Part of the MemoryManagerPackage

The description of the classes is defined in Appendix E.

Figure 31: Mediator Pattern Analysis

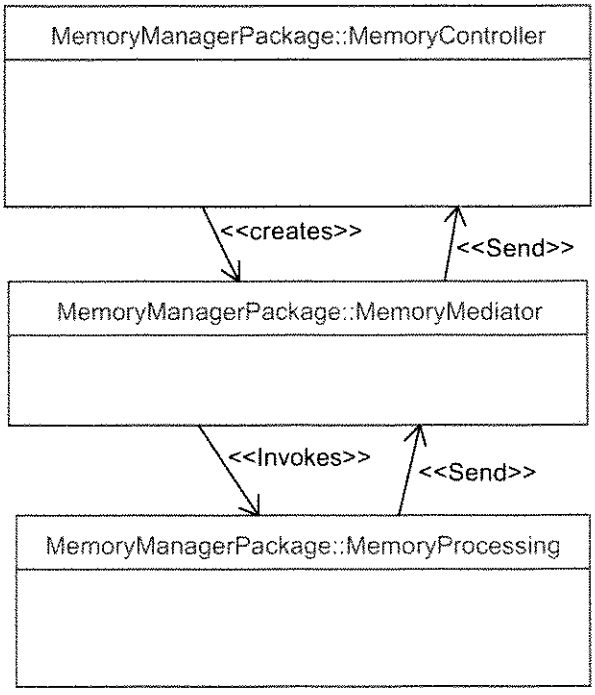


Figure 31 above, shows the use of Mediator Pattern in the MemoryManager component. The MemoryMediator class acts as a mediator between the

MemoryController and MemoryProcessing. The analyzeRequest() method in MemoryController class calls the availableRequest() method in MemoryMediator. MemoryMediator knows the encapsulated methods and after decomposition of the object invokes the insert, delete or getfreesize methods. MemoryMediator manages the group of objects preventing them to refer each other explicitly. The objects only know the MemoryMediator and further provide flexibility by reducing the number of interconnections. The response (MemoryStatus) encapsulated as a parameter is sent by MemoryProcessing by calling the availableResponse() method of the MemoryMediator class. The MemoryMediator passes the method to the MemoryController providing transparency and independence between the two (MemoryProcessing and MemoryController) communicating classes.

Fixed Size Buffer Pattern

The main objective of this pattern is to create fixed size block of memory. It "provides an approach for true dynamic memory allocation that does not suffer from one of the major problems that affect most such systems: memory fragmentation" (Douglass, 2003, p.273). The Object Model Diagram (Figure 22: MM_OMD_001) for the MemoryManager component in Chapter III illustrates the use of this pattern in the design. The classes considered in the design when using this pattern are:

- CallMemoryManager: Part of the OperatingSystemPackage
- MemoryProcessing: Part of the MemoryManagerPackage
- AbstractMemoryProcessing: Part of the MemoryManagerPackage

The description of the classes is defined in Appendix E and F.

The following pattern is used in the MemoryManager to create fixed sized blocks of memory. The AbstractMemoryProcessing calls the fixedblocksize() to create a fixed blocks based on the size of the heap and value from the configuration file. The value in the configuration file represents the largest block size the MemoryCallers need for their respective operations. The fixed block size pattern provides an approach to remove memory fragmentation (Douglass, 2003, p.273). The memory callers can perform their functions and be more reliable as the memory remains un-fragmented (Douglass, 2003, p.273). The AbstractMemoryProcessing class creates fixed block nodes, which keep track of the empty and used blocks in the memory. The node is defined by its size, start address and the end address. When the node is freed by the delete() method, a new empty node is created. The status of the nodes is sent back to the MemoryCaller. Using the fixed buffer size pattern a robust and reliable system is created which removes fragmentation (Douglass, 2003, p.277). The pre-defined maximum block size in the configuration file is a compromise to make the system deterministic. The aerospace applications require a deterministic system hence increasing the safety and avoiding hazardous condition.

Memory Manager and OOTiA objectives

Single and Multiple Inheritance

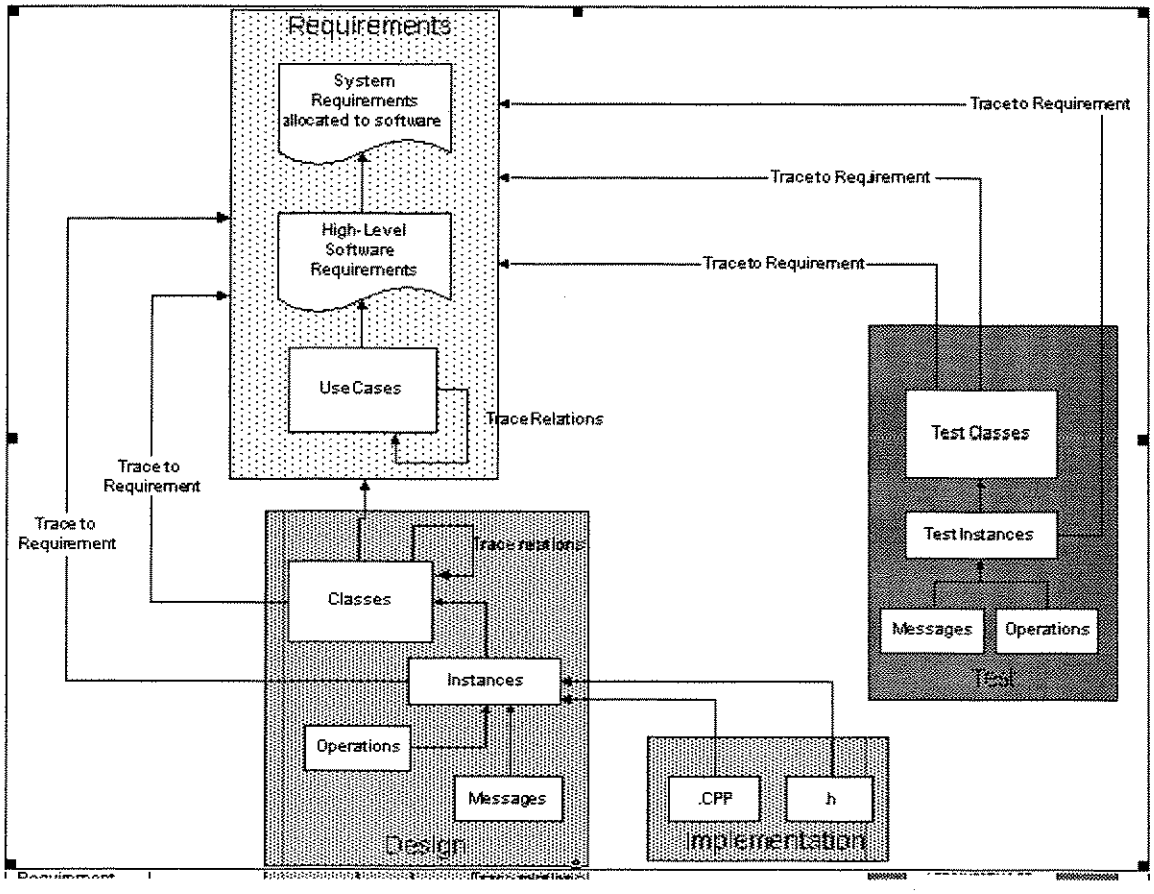
According to the OOTiA Handbook Volume 3, Section 3.3.2, "Single inheritance allows each class to have at most one immediate superclass, while multiple inheritance permits a class to have more than one immediate superclass". Inheritance is a static and compile-time concept supported by various programming languages (Gamma et al., 1995, p.19). MemoryManager component designed using the Design Patterns follows the

notion of developing software using interface and not implementation. The design of MemoryManager using Design Patterns follows the single inheritance guidelines as only the MemoryProcessing subclass extends the AbstractMemoryProcessing superclass. The MemoryManagerInterface provides the signature of the methods, but does not implement them. The Use of creational patterns like the Abstract Factory and Factory Method allow definition of signatures and invocation of methods to remain separated from the actual implementation by inner classes. The AbstractMemoryProcessing class implements the interface but defines the methods as abstract. The abstract nature of the class and methods prevents the instantiation of the AbstractMemoryProcessing. MemoryProcessing class extends the AbstractMemoryProcessing class and implements the signatures of methods defined in the MemoryManagerInterface. There is no extra payload carried down to the actual method implementation in the MemoryProcessing class, as only the signatures of the methods are overridden and not the implementation. The abstract class only provides minimal functionality hence creating a reusable and flexible design. MemoryManager does not use multiple inheritance in its design. MemoryManager uses the concept of object composition in its design between the MemoryProcessing class and the DLNode class. Object composition is more favourable over inheritance as it keeps the internals of the classes encapsulated. MemoryProcessing class in the MemoryManager component explicitly invokes the AbstractMemoryProcessing class by calling the super() library method and thus following the method exception rule specified in Volume-3 of the OOTiA Handbook (RTCA Practitioners Course CD, 2005, p. 3-18).

Traceability

To satisfy the DO-178B objective, traceability is required between High-Level requirements, Low-Level requirements, source code and the test cases for the system. The objective of Traceability, when developing OO software is difficult to satisfy because the "Functional requirements are specified by Use Cases, Dynamic dispatch, polymorphism, and inheritance; and Overloading and overriding functionality" (RTCA Practitioners Course CD, 2005, OOTiA Handbook, 2004, Vol-3, p. 3-56). The functional requirements that are developed as Use Cases are traced as per the proposed example traceability approach Figure 32, for the OO based system by OOTiA Handbook. (RTCA Practitioners Course CD, 2005, Vol-3, 2004, p. 3-58)

Figure 32: Traceability Chart (RTCA Practitioners Course CD, 2005, Vol-3, 2004, p. 3-59)



As per the OOTiA committee's proposal, the traceability for OO design "should be done at the instance level" (RTCA Practitioners Course CD, 2005, OOTiA Handbook, 2004, Vol-3, p 3-58). The Sequence Diagrams in Chapter III for the sample component illustrate the static instantiation by arrows going from the instantiator to the instantiated class. The Object Model Diagrams in Chapter III for the components illustrate instantiation using the <create> stereotype. Static traceability for instances created in the

design is seen by Figure 23 in Chapter III. All the Low-Level requirements for the classes that are implemented in the Low-Level Sequence Diagrams are traced back up to the High-Level requirements and High-Level Sequence Diagrams as per the traceability matrix in Chapter III.

The guidelines from the OOTiA suggest that "the developer should have a process that enforces the guidelines for traceability, some of the basic guidelines include:

- Every object in the software should be traced to its class.
- Every class in the software should be traced to its superclass.
- Every operation call should be traced to its class.
- Each overridden or overloaded operation should be traced to some requirement(s) or Use Case(s). Every class or superclass should be traced to the Use Case(s) that they realize." (RTCA Practitioners Course CD, 2005, OOTiA, Vol-3, 2004, p.3-59)

One approach to satisfy the objectives above for the sample component is by inspection of the static Sequence Diagrams in Chapter III. There is no method overloading in the design of the sample component and only the signatures of the methods are overridden. The Traceability matrix in Chapter III illustrates the Object Model Diagrams, which are traced back up to the Use Cases of the sample component. Analysing the Object Model Diagrams of the sample components the hierarchies of classes and dependencies are seen. The visibility of the relationships specified between

the classes satisfies the objectives of the OOTiA. (RTCA Practitioners Course CD, 2005, OOTiA Handbook, Vol-3, 2004, p. 3-58)

MemoryManager is developed following an iterative methodology. Its artifacts are created at every iteration. Furthermore, these artifacts have to be managed and traced as per the guideline by OOTiA. The complexity of traceability required to satisfy the objectives makes manual traceability difficult. The limited scope of the sample component makes it feasible to create a manual traceability matrix to shows the static traceability between the High-Level and Low-Level artifacts. As per OOTiA Volume 3, there are no current OO tools available which help to satisfy the objectives of traceability. (RTCA Practitioners Course CD, 2005, OOTiA Handbook, Vol-3 2004, p. 3-57)

OO Tools

The MDA tool Rhapsody is used for sample component to create static models for Use Cases, Sequence Diagrams and Object Model Diagrams. The tool does not have any built-in traceability features and a manual traceability matrix in Chapter III is used for management of the High-Level and Low-Level requirements for the sample component. The Rhapsody Automatic Code Generator (ACG) is not used in the MemoryManagement component. Rhapsody does not perform any structural coverage analysis for the design. Therefore, the testing, verification and validation activities should be detailed to catch the anomalies. However, this activity is out of the scope of the component.

Inlining, Template, Structural Coverage, Type Conversion and Overloading

The issues of inlining, templates and method overloading described in OOTiA are not discussed in the scope of the sample component.

Dead and Deactivated Code and Reuse

The MemoryManager is created as an RSC, which is used by the OS to handle memory management. OS as per the Volume 3 of OOTiA Handbook is a "non-application specific reusable component" which may link to the system to provide their functionality (RTCA Practitioners Course CD, 2005, p.3-47). The issue of dead and deactivated code is not a concern in the MemoryManager sample component as all the design is static and the OS is responsible for providing the functionality for compatibility with the system. In an instance where MemoryManager is supposed to service different OS the issue of deactivated code becomes a concern, as different OS will have different memory management requirements. Using Design Patterns like Adapter or Bridge, the details of the interaction can be hidden and the reusability of the component increased between different OS and the MemoryManager.

The results of the research essay illustrated the use of Design Patterns in the MemoryManager component and how the technical areas defined by the OOTiA Handbook are considered to satisfy the certification criterion. The results of the study provide information on how the use of Design Patterns can help in creating an RSC and address technical areas defined by the OOTiA Handbook within the scope of the sample component.

CHAPTER V

CONCLUSION AND RECOMMENDATION

In conclusion to the research essay, the safety critical standards and guidelines such as DO-178B and OOTiA Handbook respectively, along with process methodologies create a robust and reliable environment for the development of a trustworthy system. The research essay can be used as a reference for organizations planning to utilize Design Patterns and OO software in the aerospace projects. The maximum benefit to any project can be achieved by reviewing this research in the initial phases, as it provides information on integration of the three domains: Design Patterns, certification information, and a development approach to create a safety critical RSC. The sample component illustrates the incorporation of patterns in the design to ease the process of certification.

The safety critical feature analyzed in the sample component presented in the research essay is memory management and allocation. A hierarchical decomposition of classes that interact with each other and with other components via interfaces was designed using Design Patterns for the safe implementation of the feature. The design for sample component uses Design Patterns to create the Object Model Diagrams for the MemoryManager component. The purpose of patterns used in the component design is seen via the Object Model Diagram for the MemoryManager component in Chapter III. The Abstract Factory Pattern and Factory Method Pattern are the creational patterns, which create objects in the MemoryManager component. The Composite and Facade are structural patterns, which organize the objects into groups, and the Command, Iterator,

Mediator, Momento and Chain of Responsibility are behavioural patterns that allow a means of communications between the objects of the MemoryManager component.

The requirements defined for the sample component are designed to be unique to meet the consistency criteria of DO-178B. The distinctiveness of the requirements helps in the safe implementation of the Low-Level design using Design Patterns. The reusability of the sample component is achieved by hiding the instantiation of classes by the interface. The inner classes are not visible to the caller making the system more robust and safety critical.

The MemoryManager component creates virtual fixed partition blocks of the same size to develop a reliable and predictable system. The Fixed Size Buffer pattern embedded in the decision to define equal sized blocks of memory creates a deterministic design, making the component more suitable for aerospace software. The sample component follows the best-fit strategy for data allocation. The predictable creation of blocks via the configuration file storing the memory requirements of different MemoryCallers is one approach of making the component safer and more reliable.

The MemoryCaller component materializes the requests of fixed size blocks from the OperatingSystem. Each block is unique for each request. These non-overlapping equal sized blocks result in creating a safety critical memory manager. The uniqueness of all the blocks assigned is significant to maintain the integrity of system. The OperatingSystem component is assumed to be generic with primitives to handle communication with other components in the design. The OperatingSystem initializes the

MemoryManager with the size of the heap and receives the handle of the allocated heap as a response.

Design Patterns ease the implementation of the safety critical features of the sample component. The Factory Method pattern and the Abstract Factory pattern in the design provide the hiding of the implementation and instantiation issues, defining an interface for creating the objects. The Abstract Factory pattern hides the related objects, which service the requests of the MemoryCaller, without the caller knowing the internal details of the implementation.

A variant of the Facade pattern is used in the sample component to reduce coupling between the MemoryCaller and the MemoryManager components. The complexity of the MemoryManager is hidden and only the signatures of the methods are available to the MemoryCaller. The composite pattern in the design is used on the DLNode objects created, as they are part compositions for the whole MemoryProcessing class.

The Command Pattern in the design is used firstly to encapsulate the parameter holding the size of the heap and then to encapsulate the MemoryCallers Identification and requests for the services offered by the MemoryManager. The Iterator Pattern is used in the design to access the nodes without exposing the internal details of the object.

A variant of Momento pattern in the design is used to provide the MemoryManager component with a technique to manage the status of requests made by the MemoryCaller. The status is captured as data is inserted in the nodes.

The Mediator pattern in the design provides loose coupling by keeping objects from referring to each other explicitly. The use of the Mediator pattern provides flexibility by reducing the number of interconnections between the classes of the MemoryManager. As a result the identity of the MemoryCaller is hidden from the concrete classes processing the request of the caller.

The Chain of Responsibility Pattern in the design is used to reduce the coupling of the request from the sender to processing by the receiver. It also facilitates the collaboration of the objects instantiated from the time the request is made from MemoryCaller to the time it is processed by MemoryManager.

The objectives of DO-178B support procedural approach to software design and may become "unclear and/or complicated" for the sample component (RTCA Practitioners Course CD, 2005, OOTiA Handbook Vol-1, 2004, p.1-1). The OO design of the sample component follows the relevant technical areas defined in the OOTiA Handbook as guideline for certification. The technical areas considered in the Chapter IV for the sample component adhere with the criterion to satisfy the objectives for the certification process. The set of requirements in the sample component are designed to be independent, satisfying the consistency criteria. The relation between the High-Level requirements and Low-Level requirements via the traceability matrix in Chapter III is one approach to partially satisfy the traceability criterion.

The benefits of using the OO principles in the aerospace software come with an immense amount of additional work to create and to manage artifacts. The requirements created using specification are easier to manage than having a Use Case for each

requirement. The use of Sequence Diagrams is beneficial to see the message communication between the various entities and the visual interaction between the classes is easily seen via the Object Model Diagrams.

The scope of certification is vast and the future research can provide more information on phases like implementation, testing, and verification which can follow the OOTiA Handbook as guideline for technical areas like inlining, template, structural coverage, type conversion, and satisfy the verifiability, accuracy, testing and completeness criteria to develop safe and reliable software.

The implementation phase of the development can be explored in future research using the ACG features of the development tools. The use of tools to generate source code and binaries not only reduces the development time but also makes the certification process difficult because the tool have to be qualified for their integrity. An approach to reduce the effort for certification is by providing information on the anomalies (pre-processing directives, coding style) of the tools and the language chosen for development in the SCS and the PSAC of the project.

The two types of testing, which can be explored in future research, are black-box or functional testing and white-box or structural testing. An approach to satisfy the testing criteria under DO-178B is by doing requirements-based testing. An independent test case is developed to satisfy each specific requirement. The test cases will cover all the normal range and robustness testing for the functional requirements. To verify the coverage of the source code, structural testing can be performed via tools like LDRA or manually. The authentication of tests against the requirements and the source via the traceability

matrix, checklists and reviews can provide an approach to satisfy the verification criteria of DO-178B.

REFERENCES

Aho, V. Alfred., Hopcroft, E. John., & Ullman, D. Jeffrey., (1987).*Data Structures and Algorithms*. USA: Bell Telephone Laboratories, Incorporated.

Carpenter, B. Paul. (1999). *Verification of requirements for safety-critical software*: Annual International Conference on Ada. Proceedings of the 1999 annual ACM SIGAda international conference Redondo Beach, California, United States pp: 23 – 29. Website:http://0-doi.acm.org.aupac.lib.athabascau.ca:80/10.1145/319294.319299

Copper, W. James, (2000). *Java Design Patterns A Tutorial*. USA: Addison-Wesley.

Crowley, Charles. (1997). *Operating System-A Design Oriented Approach*. USA: Times Mirror Higher Education Group Inc.

Douglas, Powel. Bruce. (2003). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston: Pearson Education Inc.

Eckel, Bruce. (2000). *Thinking in Java* (2nd ed.).New Jersey: Prentice-Hall.

Ferrett. K Lisa., & Offutt, Jeff. (2002). *An Empirical Comparison of Modularity of Procedural and Object-Oriented*. Retrieved January 20th 2005, from George Mason University. Fairfax VA, Website: http://www.isse.gmu.edu/faculty/ofut/rsrch/abstracts/oometrics02.html >

Goodrich, T. Michael., & Tamassia, Roberto. (2001). *Data Structure and Algorithms in Java* (2nd ed.). New York: John Wiley and Sons, Inc.

Gamma, Erich., Helm, Richard., Johnson, Ralph., & Vlissides, John., (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. USA: Addison-Wesley.

GreenHills Software Inc. (2005) Safety Critical Products: INTEGRITY®-178B RTOS. October 5, 2005 from, Website: http://www.ghs.com/products/safety_critical/integrity-do-178b.html

Hachman, Mark. (2004). NASA: *DOS Glitch Nearly Killed Mars Rover*. ExtremeTech Build It, Tweak It, Know It. Retrieved January 5, 2005 from, Website: http://www.extremetech.com/article2/0,1558,1638764,00.asp

Hayhurst, J. Kelly., Holloway, C. Michael., (2003). *Considering Object Oriented Technology in Aviation Applications*. Virginia: NASA Langley Research Center, Hampton. pp. 1-8, Retrieved January 25, 2005 from, Website: http://techreports.larc.nasa.gov/ltrs/PDF/2003/mtg/NASA-2003-22dasc-kjh.pdf

Hoffer, A. Jeffery., Prescott, B. Mary., & McFadden, R. Fred., (2002). *Modern Database Management* (6th ed.). Prentice Hall.

Ilogix. (2005). Retrieved March 5 2005 from, Website: www.**ilogix**.com/

Larman, Craig., (2005). *Applying UML Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development.* (3rd ed.). New Jersey: Pearson Education Inc.

LDRA Inc. (2005). Retrieved March 5 2005, from, Website: http://www.ldra.co.uk/index.asp

Phillips et al., (1999). *"Semiotic Modeling for Text String De-Duplication"* Applied Semiotics. Learned Journal of Literary Research on the World Wide Web. Issue N° 6/7 Frontiers of Semiotics. Article 1. pp. 7/11. Retrieved December 15, 2004 from, Website: http://www.chass.utoronto.ca/french/as-sa/ASSA-6-7/JP7.html

Portunity., (2002). *9.1 General Brief Introduction to OOP. Difference to Procedural Programming*. Portunity International. Germany. Retrieved February 20, 2005 from, Website: http://www.portunity.info/article4404-1841.html

Romanski, G. (2002). *Certification of an operating system as a reusable component*. Retrieved March 10 2005, from, VeroCel The Software Certification Company, pp.1-9 Website: http://demo.verocel.com/papers/index.htm

RTCA/DO-178B (1992). *Software Considerations in Airborne Systems and Equipment Certification Standard.*

RTCA Practitioners Course. (2005) *DO-178B Software Considerations in Airborne Systems and Equipment Certification.* Seminar Slides from Certification Services Inc (CSI) Lecture 001-032.

RTCA Practitioners Course CD # 04-099-1092, Rev. B. (2005). *Supplemental Information for the RTCA/DO-178B Training Seminar.* Certification Authority Software Team (CAST) Paper- 4 (2000), pp. 1-14, Paper-6 (2001), pp. 1-9.

RTCA Practitioners Course CD # 04-099-1092, Rev. B. (2005). *Supplemental Information for the RTCA/DO-178B Training Seminar.* Federal Aviation Authority. (2004). *Handbook for Object Oriented Technology in Aviation (OOTiA).* Volume 1, 3.

RTCA Practitioners Course CD # 04-099-1092, Rev. B. (2005). *Supplemental Information for the RTCA/DO-178B Training Seminar.* Certification of Civil Avionics (CCA) Version 2.0 Dist. Avionics Handbook Chapter 20, pp.1-19.

RTCA Practitioners Course CD # 04-099-1092, Rev. B. (2005). *Supplemental Information for the RTCA/DO-178B Training Seminar.* Federal Aviation Administration. (2004). Advisory Circular (AC) 20-148 Reusable Software Components. USA: Ministry of Transportation.

Schmidt, C. Douglas., (2004). *Introduction to Patterns and Frameworks.* Retrieved December 23, 2004 from, Department of EECS Vanderbilt University, UCLA Extension Course Slides 1-48, Website: http://ww.cs.wustl.edu/~schmidt/PDF/patterns-intro4.pdf

Selena, Sol., (1998). *Limitations of Procedural-Oriented Programming.* Retrieved January 22, 2005 from, Web Developers Virtual Library. Website: http://wdvl.internet.com/Authoring/Languages/Perl/5/procedural_programming_limits.html

TeleLogic (2005). *DOORS Requirement Traceability Tool.* Retrieved March, 5 2005 from, Website: http://www.telelogic.com/products/doorsers/doors/index.cfm

VeroCel. (2005). *VeroTrace Requirement Traceability Tool.* Retrieved March, 5 2005 from, VeroCel The Software Verification Company, Website: http://www.verocel.com/

Wlad, Joseph., (2001). *DO-178B and Safety-Critical Software Technical Overview.* Retrieved October, 26 2004 from Real Time Embedded Forum Meting notes, Website: http://www.opengroup.org/rtforum/jul2001/slides/wlad.pdf

Zhang, Kun, Xiao., (2003). *Computer Science 617 Designing Real-Time Software.* Course Notes. Unit 1 Section 1: Introduction to Real-Time Systems. Athabasca University.

APPENDIX A

Artifact Description

PSAC

This is a standard "Plan for Software Aspects of Certification" document, corresponding to the guidelines in RTCA DO-178B. It describes the general characteristics of the system and its software, certification considerations, life-cycle and life-cycle data, and scheduling of the software development effort (RTCA DO-178B Standard, 1992, p. 69). Example template of this document is provided in Appendix B.

SDP

This is a standard "Software Development Plan" document, corresponding to the guidelines in RTCA DO-178B. It describes the standards, software life-cycle, and development environment used in the software-development effort (RTCA DO-178B Standard, 1992, p. 65).

SQAP

This is a standard "Software Quality Assurance Plan" document, corresponding to the guidelines in RTCA DO-178B. In the words of DO-178B, it "establishes the methods to be used to achieve the objectives of the software quality assurance (SQA) process" (RTCA DO-178B Standard, 1992, p. 68).

SCMP

This is a standard "Software Configuration Management Plan" document, corresponding to the guidelines in RTCA DO-178B. In the words of DO-178B, it "establishes the methods to be used to achieve the objectives of the software configuration management (SCM) process throughout the software life-cycle" (RTCA DO-178B Standard, 1992, p. 66).

SVP

This is a standard "Software Verification Plan" document, corresponding to the guidelines in RTCA DO-178B. It describes the verification procedures used in the software development effort. As a "Plan", the discussion herein does not involve details specific to the software under development. Refer instead to the Software Verification Cases and Procedures (RTCA DO-178B Standard, 1992, p. 66).

SRD

This is a standard "Software Requirements Data" document, corresponding to the guidelines in RTCA DO-178B. In the words of DO-178B, it "is a definition of the high-level requirements including the derived requirements" (RTCA DO-178B Standard, 1992, p. 70).

SDD

This is a standard "Software Design Description" document, corresponding to the guidelines in RTCA DO-178B. In the words of DO-178B, it "is a definition of the software architecture and the low-level requirements that will satisfy the high-level requirements" (RTCA DO-178B Standard, 1992, p. 70).

STCP

This is a standard "Software Test Cases and Procedures" document, corresponding to the guidelines in RTCA DO-178B. It details the software testing activities. In particular it provides test cases and test procedures to be performed on the software under development (RTCA DO-178B Standard, 1992, p. 71).

SCI

This is a standard "Software Configuration Index" document, corresponding to the guidelines in RTCA DO-178B. It identifies the configuration of the software (RTCA DO-178B Standard, 1992, p. 72).

SVR

This is a standard "Software Verification Report" document, corresponding to the guidelines in RTCA DO-178B. It includes results from the Verification and Validation activities of the system and its software (RTCA DO-178B Standard, 1992, p. 72).

SAS

This is a standard "Software Accomplishment Summary" document, corresponding to the guidelines in RTCA DO-178B. It includes overviews of the system and its software, certification considerations, software characteristics, life-cycle information, and the change-history of the software (RTCA DO-178B Standard, 1992, p. 74).

APPENDIX B

Module Comparison

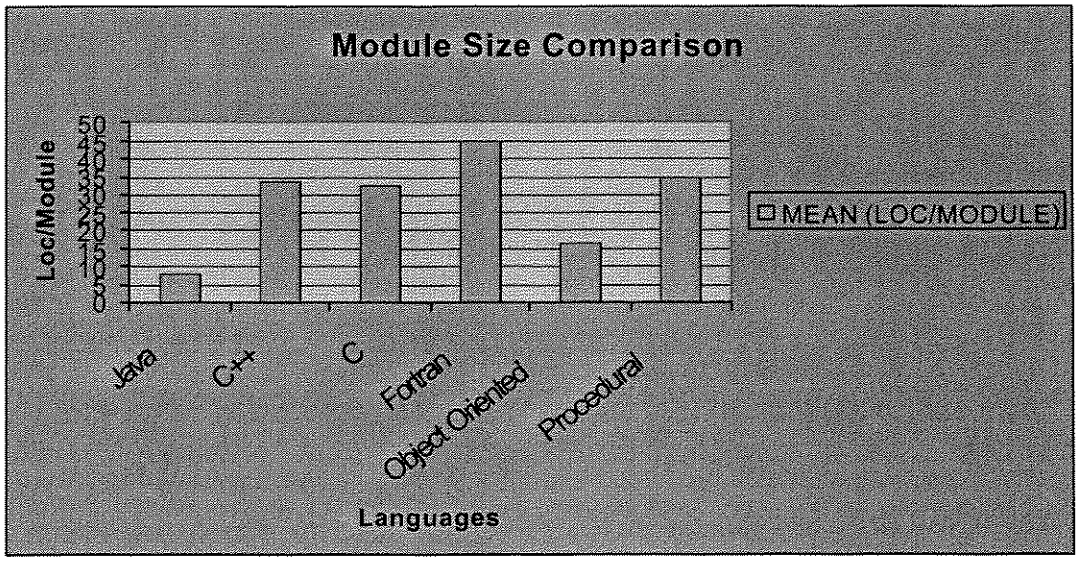
Module Size Comparison Table

The first column of the table below displays languages used in the study to compare the module size of the program. The second column displays the mean ratio of Lines of Code (LOC) in the program per module. "The average size of the OO (Java and C++) modules (16.6 lines of code) is slightly less than half (47%) the size of the procedural modules(C and Fortran) (34.8 lines of code)" (Farrett & Offutt, 2002).

Language	*MEAN (LOC/MODULE)*
Java	7.9
C++	33.4
C	32.6
Fortran	44.6
OO (Java and C++)	16.6
Procedural (C and Fortran)	34.8

Module Size Comparison Chart

The following is the graph representing the table above with languages on the x-axis and the line of code/module on the y-axis (Farrett & Offutt, 2002).



APPENDIX C

PSAC Template

The following sample template is created from the DO-178B standard to illustrate the type of information required for the PSAC documents (RTCA DO-178B Standard, 1992, p. 69).

Plan for Software Aspects of Certification
<Project Title>
<Level of DO-178B Certification>

<Organization Name and Address>

Document No.	**XXXXXX**
Revision	X
Publication Date	XX/XX/XXXX

Copyright information outlining the years the organization has been in business. Legal notice illustrating restrictions on the distribution of the documents.

Note: *The sign-of matrix is a means for identifying the reviewers of the specific document. Every individual on the list reviews the document once it is ready for reviewing. Once everyone on the list is satisfied, the document is baselined and given a revision.*

Sign-off Matrix:

	Name	Date		Name	Date
Author	Varun Malik *(Signature)*	XX/09/2004	Quality	XXXXXXX *(Signature)*	
Technical Lead	XXXXXXX *(Signature)*		Software Manager	XXXXXXX *(Signature)*	

Revision	Summary	Date
< X>	Original Release *<This section can provide information along with the section number which was updated. Page number of the section update in the document. Summary of the change. >*	<XX/XX/XXXX>

Table of Contents

1. Introduction

1.1 Overview

<Generic Overview>

<This Plan for Software Aspects of Certification (PSAC) has been prepared to provide an outline of the activities relating to the design and development of the <Project Name>. This section provides more project specific information depending on the decision of how to handle the generic issue. This section will mention the name of the client>

1.2 Purpose

<Purpose of the PSAC>

<This document is intended to be used as the primary means available to the certification authority for determining whether the applicant's life-cycle data is commensurate with the rigor required for the level of software required for the <Project Name>. >

1.3 Scope

<This document has been prepared in accordance with the Technical Commission for Aeronautics, Software Consideration in Airborne Systems and Equipment Certification Document No. RTCA DO-178B.
The planned software life-cycle processes for the <Project Name> will be contained in following additional plans:
Software Development Plan (SDP),
Software Configuration Management Plan (SCMP),
Software Verification Plan (SVP),
Software Quality Assurance Plan (SQAP). >

2. Referenced Documents

2.1 Company Specific Documents

<List the documents which were used to create the PSAC>

<Doc Number> <Doc Name>

2.2 Other Documents

<List all other external documents the documents which were used to create the PSAC>
DO-178B RTCA Software Considerations in Airborne Systems and Equipment

3. System Overview

<Description of the project and the detailed overview of the entities involved. This section explains how the component fits in to the whole system. Subsections will give the details of the architecture, inputs, outputs and the interface of the component. >

<If this document applies to the MemoryManager Component. The High-Level System Component Diagram will be explained in this section and explanation will be provided according to the interface (MemoryManagerInterface, MemoryCallerInterface and the OperatingSystemInterface and the components which interact to provide the functionality of the Memory Management. >

3.1 Architecture

<Functional Block Diagram and explanation of each of the entities>

3.2 Interface

<Table outlining all the inputs and outputs of the components. >

Type	Name	Input/Output	Description

3.3 Safety Features

<Failure Detection: Detection of the failure for the Component>

4.0 Software Overview

4.1 Fault Tolerance

<Project specific information>

4.2 Resource Sharing and Redundancy

<Project specific information>

4.3 Timing and Scheduling

<Project specific information>

4.4 Software Version Control

<Each software component will be configured and controlled using a unique part number. Separate configuration index documents will identify the software modules and development environment required to create each software program. >

5. Certification Considerations

5.1 Software Level Justification

<This section will explain why the component is Level X (A-E) software (that is. Software whose failure would cause or contribute to a catastrophic failure of the aircraft). This section will also explain what kind of failure in the software can affect the functionality, which will lead to a catastrophic event? >

5.2 Certification Liaison Process

<It will be the responsibility of <Company Name> to submit the certification documents to the Customer/DER. Customer/DER will then submit the documents to Federal Aviation Authority (FAA) or Joint Aviation Authority (JAA). If the PSAC is required to be revised later in the program it will be re-submitted to DER for approval. >

5.3 Means of Compliance

<The Planned activities to obtain the means of compliance as specified by DO-178B are described in the plans for the project listed in Section 1.3. Along with the plans other documents with the results of the verification (SVR) and accomplishment of the software (SAS) will also be submitted to the certification authority. The issues raised by the authorities will be addressed in the next release of the documents providing evidence of the changes made to comply with the authorities. >

5.4 Hazard Analysis

<Every entity of the software will have a hazard level assigned to it for example. DER is supposed to do the Functional Hazard Assessment and then we support it in the software>

Number	Function	Failure Condition	Classification
1	Upload Software	Partial Upload	Minor/Hazardous/Catastrophic Major

6. Development Schedule

<This section outlines the development schedule for the <Project Name>>

7. Software Life-Cycle

7.1 Software Life-Cycle Process

<The following steps follow the following <development model> of software development. Activities in bold can be the milestones for the project. (The following cycle is just a proposed cycle; this depends on the project also on the criticality, functionality and stakeholders of the project)
Feasibility Analysis
Development of Plans and Proposal

Requirement Analysis
Requirement Design
Requirement Analysis Review
Requirement Analysis Complete
Design Analysis
Design
Design Analysis Review
Design Analysis Complete
Prototype Development Preliminary Implementation
Prototype Review
Critical Design Review
Implementation Complete
Preliminary Testing Review
Validation and Testing
Testing Results Review
Testing Complete
Verification Complete>

7.2 Software Development Process

<The plan for this process is detailed in the Software Development Plan (SDP),
The objectives of this process will be satisfied by the formal release of the following documents:
Plan for Software Aspects of Certification.
Software Development Plan.
System Requirements Document
Software Requirements Documents.
Software Design Description Documents. >

7.3 Software Verification Process

<The plan for this process is detailed in the Software Verification Plan (SVP). Document reviews, standards checklists, test coverage analysis and a verification matrix are used to verify the outputs of this process.
The following assurances will be provided:

- *System requirements traceable to High-Level requirements,*
- *High-Level software requirements traceable to Low-level software design,*
- *High-Level software requirements traceable to requirements based tests,*
- *Low-Level software design traceable to module tests,*
- *Backwards traceability of all of the above items. >*

<The objectives of this process will be satisfied by the formal release of the following documents:

- *Software Verification Plan.*
- *Software Test Case and Procedures Documents.*
- *Software Verification Results Documents.*
- *Software Accomplishment Summary. >*

7.4 Software Configuration Management Process

<The plan for this process is detailed in the Software Configuration Management Plan (SCMP). The objectives of this process will be satisfied by the formal release of the following documents: Software Configuration Management Plan
Software Configuration Index's>

7.5 Software Quality Assurance Process

<The plan for this process is detailed in the Software Quality Assurance Plan (SQAP). Software Quality Assurance Plan >

7.6 Organization

<Flow- chart describing the organizational chart of <Company Name> and giving brief description of the entities. Describing the roles of each entity in the development process>

7.7 Certification Liaison

<Information on who will liaison with the Certification Authority>

7.8 Additional Considerations

<Tool Qualification Documents: Documents the qualification evidence for a verification tool against the requirements established in the PSAC and Tool Requirements Document>

8. Software Life-Cycle Artifacts

<The artifacts produced from the different phases of the software development are described in the following sections. >

8.1 Certification

<The two documents provided to the certification authority are PSAC and SAS. >

8.2 Design

<Design artifacts SDP, SRD, SDD, and Source Code. >

8.3 Tests and Verification

<Verification artificats SVP, STCP and SVR. >

8.4 Configuration

<Configuration artifacts SCMP, SCI. >

8.5 Quality

<Software Quality artifacts SQAP and Software Quality Audit Report. >

8.6 Standards

<Standards used in the project>

SRS	*Software Requirements Standards*
SDS	*Software Design Standards*
SCS	*Software Code Standards*

9. Appendices

9.1 Appendix A: Acronyms and Definitions

PSAC	*Plan for Software Aspects of Certification*
RTCA	*Radio Technical Commission for Aeronautics*
SDP	*Software Development Plan*
SQAP	*Software Quality Assurance Plan*
SVP	*Software Verification Plan*
SCMP	*Software Configuration Management Plan*

9.2 Appendix B: Development Tools

<Description of the all the tools used in the project. Classification of tool outlines if the tool is a Development tool or a Verification Tool>

Tool Name	*Description*	*Classification and Justification*
Microsoft Word	*Used for Editing*	***Development Tool** but not qualified as only used as editor and the output of the tool is verified.*

APPENDIX D

Memory Caller Package

The MemoryCaller component comprises of the MemoryCallerPackage, which encapsulates all the internals of the component in the package. The MemoryCallerInterface is the only exposed class of the component.

Class name: MemoryCallerInterface

MemoryCallerInterface class depicts the exposed interface to the MemoryManager and the OperatingSystem. There are no methods defined in this interface and it is up to specific MemoryCaller to decide which methods it wants to expose to other components.

Class name: CallOperatingSystem

The following class belongs to the MemoryCaller component. The main purpose of this class is to request the OperatingSystem for a handle to the block of heap, so that the MemoryCaller can use that block to perform its operations.

Operation information for Class: CallOperatingSystem

Operation name: getSubBlockHandle

Argument information for Operation getSubBlockHandle

Direction	Type	Name
Input	None	None
Return	MemoryProcessing	None

The following public method is a key entity. It is the starting point of the sample model as the initial call for the block initialization is made in this method. It calls the OperatingSystem to return the handle for memory, which it uses for its requests. The

following method creates a new instance of the OperatingSystem.CallMemoryManager() and calls getHeapHandle() method. The MemoryManager returns the initial heap handle for the block back to the MemoryCaller via the OperatingSystem

Class name: RequestMemoryManager

The following class sends requests from MemoryCaller to the MemoryManager. The operations, which are requested, are offered by the interface of MemoryManager.

Operation information for Class: RequestMemoryManager

Operation name: sendRequest

Argument information for Operation sendRequest

Direction	Type	Name
Input	None	None
Return	Boolean	None

The following public method encapsulates the requests as parameters and passes them to the MemoryManager. The MemoryController class of MemoryManager is instantiated by this method and the MemoryController.addRequest(request) method is called with the request as a parameter. The parameter holds either of the operations (insert, delete, getfreesize) offered by the MemoryManagerInterface. The return type informs if the request was successfully sent to the MemoryManager.

Class name: ResponseMemoryManager

The following class polls for responses from the MemoryController class of the MemoryManager. When a response is available the MemoryCaller receives it.

Operation information for Class: ResponeMemoryManager

Operation name: recieveResponse

Argument information for Operation recieveResponse

Direction	Type	Name
Input	None	None
Return	MemoryStatus	None

The following public method receives the encapsulated response from the MemoryController class. The response is the starting address and the status of the memory block on which the MemoryManager processed the request. The handling of the response is not part of the model and is a design decision of the MemoryCaller.

APPENDIX E

Memory Manager Package

The MemoryManager component comprises of the MemoryManagerPackage, which encapsulates all the internals of the component in the package. The MemoryManagerInterface is the only exposed class of the component.

Class name: MemoryManagerInterface

MemoryManagerInterface class depicts the exposed interface to the MemoryCaller and the OperatingSystem. The methods defined in the interface are services offered by the MemoryManager, which are utilized by the MemoryCaller.

Operation information for Class: MemoryManagerInterface

Operation name: getFreeSize

Argument information for Operation getFreeSize

Direction	Type	Name
Input	None	None
Return	int	None

Interface definition of the method.

Operation name: insertData

Argument information for Operation insertData

Direction	Type	Name
Input	int	data_size
Return	MemoryStatus	None

Interface definition of the method.

Operation name: deleteData

Argument information for Operation deleteData

Direction	Type	Name
Input	int	start_address
Return	MemoryStatus	None

Interface definition of the method.

Class name: AbstractMemoryProcessing

Attribute information for Class: AbstractMemoryProcessing

Name	Type
heap_size	int
free_block	int
first_node	DLNode
head	DLNode
tail	DLNode

AbstractMemoryProcessing class is an abstract class, which implements the MemoryManagerInterface. All methods are abstract in this class and it is the superclass of MemoryProcessing class.

Operation information for Class: AbstractMemoryProcessing

Operation name: AbstractMemoryProcessing

Argument information for Operation AbstractMemoryProcessing

Direction	Type	Name
Input	int	h_size
Return	None	None

The following public method is the constructor for the AbstractMemoryProcessing class. The constructor of MemoryProcessing class invokes this method. The main objective of this constructor is to create the deterministic size from the pre-defined heap size (input parameter ***h_size***) passed by the OperatingSystem. The fixed size determined by calling fixedblocksize (***h_size***) method is the new fixed heap_size, which is used to create new

block. The whole size of the block is initialized to free, by assigning the size (***h_size***) to free_block. The head, tail of the node are initialized by instantiating DLNode() class. The first_ node of the memory block is created using the new heap_size, head and tail, start address for the node and the status of the node. The next and previous nodes of the first_ node are assigned by calling the setNext() and setPrev() modifiers from the DLNode() class.

Operation name: insertData

Argument information for Operation insertData

Direction	Type	Name
Input	int	data_size
Return	MemoryStatus	None

Abstract method not implemented in this class.

Operation name: deleteData

Argument information for Operation deleteData

Direction	Type	Name
Input	int	start_address
Return	MemoryStatus	None

Abstract method not implemented in this class.

Operation name: getFreeSize

Argument information for Operation getFreeSize

Direction	Type	Name
Input	None	None
Return	int	None

Abstract method not implemented in this class.

Operation name: fixedblocksize

Argument information for Operation fixedblocksize

Direction	Type	Name
Input	int	h_size
Return	int	None

The following public method sets the block size of the heap. This method is invoked by the constructor of AbstractMemoryProcessing and is passed the heap size provided by the OperatingSystem. The following method retrieves the memory requirement information from the configuration file for the MemoryCallers. The size in the configuration file defines the requirements of memory for different MemoryCallers. The memory requirements are analysed and a comparison is performed to get the maximum number of blocks requested by the callers. The maximum number is used to determine an optimum size for the fixed block. For example if the initial size of the memory which was assumed in the OperatingSystem was 100 and assuming that the Max(MemoryCaller requirement) is four blocks (this is available from the configuration file which is share between the MemoryManager and OperatingSystem) then the potential block size for each block in the memory model is 100/four: 25. The new fixed block size is returned back for further processing in the constructor of AbstractMemoryProcessing.

Relation information for Class: AbstractMemoryProcessing

Relation name: itsDLNode

Name	itsDLNode
Inverse	None
Source	AbstractMemoryProcessing
Target	DLNode
Multiplicity	1
Type	Association

Class name: MemoryProcessing

The following class is a subclass of AbstractMemoryProcessing class. This class returns the handle for the heap of block to the OperatingSystem. It also processes the requests made by the MemoryCaller.

Operation information for Class: MemoryProcessing

Operation name: MemoryProcessing

Argument information for Operation MemoryProcessing

Direction	Type	Name
Input	int	h_size
Return	None	None

The following is a constructor for MemoryProcessing class. It explicitly calls the super() method, which invokes the constructor of the AbstractMemoryProcessing class. The constructor is passed the heap size (***h_size***) sent by the OperatingSystem component.

Operation name: getFreeSize

Argument information for Operation getFreeSize

Direction	Type	Name
Input	None	None
Return	int	None

The following public method returns the size of the heap. The unused memory (free size) is returned to the MemoryCaller once it queries the MemoryManager. The size returned is the free size of the heap.

Operation name: insertData

Argument information for Operation insertData

Direction	Type	Name

Input	int	data_size
Return	MemoryStatus	None

The following public method inserts the data in the memory block, which is allocated to the MemoryCaller. Once the data is inserted the MemoryStatus of the block is returned back to the MemoryCaller. The strategy used for data insertion is best-fit strategy and the whole list of block is searched to find the best-fit block. When the best-fit node is found, a new node is created with the data_size provided by the MemoryCaller as a parameter. The next and the previous block of the best-fit nodes are set and calling the modifiers from the MemoryStatus class sets the MemoryStatus. The MemoryStatus inclusive of the state of the block and the starting address of the inserted data is returned back to the MemoryCaller.

Operation name: deleteData

Argument information for Operation deleteData

Direction	Type	Name
Input	int	start_address
Return	None	None

The following public method is called by the MemoryCaller to delete the data from the block with the specified address. The method checks the validity of the starting address. If there is no data at the address, then the status is set and returned back to the MemoryCaller. If there is data at the address then checks are performed to see if there is data in the previous node, next node or both the nodes. A new node is created based on the new size after taking into consideration the previous and next nodes. The next and the previous nodes for the new node are set using the modifiers from the DLNode() class.

The new node is added to the free space and the MemoryStatus is returned back to MemoryManager

Generalization information for Class: MemoryProcessing

Generalization name: AbstractMemoryProcessing

Name	AbstractMemoryProcessing
Base Class	AbstractMemoryProcessing
Derived Class	MemoryProcessing

Relation information for Class: MemoryProcessing

Relation name: itsDLNode_1

Name	itsDLNode_1
Inverse	itsMemoryProcessing
Source	MemoryProcessing
Target	DLNode
Multiplicity	1..*
Type	Composition
LinkName	Composite

Relation name: itsMemoryStatus

Name	itsMemoryStatus
Inverse	itsMemoryProcessing
Source	MemoryProcessing
Target	MemoryStatus
Multiplicity	1
Type	Association

Class name: MemoryStatus

Attribute information for Class: MemoryStatus

Name	**Type**
state	boolean
start_addr	int

The following class is instantiated by the MemoryProcessing class to store the status of the memory blocks. The state attribute stores a Boolean value to specify the full or empty

state of the block and the start_addr stores the starting address. Both the attributes are manipulated by operations performed on the node and a status of the node is returned back calling the assessors of the MemoryStatus class.

Operation information for Class: MemoryStatus

Operation name: MemoryStatus

The following is the constructor to initialize the memory status and start address.

Operation name: getStart_addr

Argument information for Operation getStart_addr

Direction	Type	Name
Input	None	None
Return	int	None

The following public method is an assessor and gets the start address of the block.

Operation name: setStart_addr

Argument information for Operation setStart_addr

Direction	Type	Name
Input	int	p_start_addr
Return	Void	None

The following public method is a modifier and sets the start address of the block.

Operation name: getState

Argument information for Operation getState

Direction	Type	Name
Input	None	None
Return	boolean	None

The following public method is an assessor and gets the state of the block.

Operation name: setState

Argument information for Operation setStart_addr

Direction	Type	Name
Input	boolean	p_state
Return	Void	None

The following public method is a modifier and sets state of the block.

Operation name: set_inserted_data_addr

Argument information for Operation set_inserted_data_addr

Direction	Type	Name
Input	int	data_start_addr
Return	Void	None

The following public method is a modifier and sets the starting address for the best-fit block found to insert data.

Operation name: get_inserted_data_addr

Argument information for Operation get_inserted_data_addr

Direction	Type	Name
Input	None	None
Return	int	None

The following public method is an assessor and gets the starting address for the best-fit block where data was inserted.

Relation information for Class: MemoryStatus

Relation name: itsMemoryProcessing

Name	itsMemoryProcessing
Inverse	itsMemoryStatus
Source	MemoryStatus
Target	MemoryProcessing
Multiplicity	1
Type	Association

Class name: DLNode

Attribute information for Class: DLNode

Name	Type
h_size	int
start_address	int
next	DLNode
prev	DLNode
empty	boolean

The following class depicts the actual block, which stores the data for the MemoryCaller operations. One approach for implementation for the DLNode is by using a double linked list data structure. Double linked-list is an efficient choice for the heap management as the traversal of the nodes can be done in both directions as each node has two references to traverse; one to the next node, and one to the previous node (Goodrich, 2001, p.198). The insertion and deletion of nodes is easier as the nodes can be inserted before, after a node or in front or back of the list. The deletion of nodes is easier when trying to reclaim the memory by concatenating the empty nodes to the free list.

Operation information for Class: DLNode

Operation name: DLNode

Argument information for Operation DLNode

Direction	Type	Name
Input	None	None
Return	None	None

The following method is a constructor to initialize a new node. This constructor is called by AbstractMemoryProcessing to set the values for the initial node. The DLNode class attributes are initialized as follows:

Name	Initial Value
h_size	0

start_address	0
next	null
prev	null
empty	true

Operation name: DLNode

Argument information for Operation DLNode

Direction	Type	Name
Input	int	heap_size
Input	int	start_addr
Input	DLNode	next_block
Input	DLNode	prev_block
Input	boolean	empty_space

The following method is a constructor to initialize a new node, which takes parameters from the callers of this method. AbstractMemoryProcessing and MemoryProcessing call this constructor with the parameters to set the values for the node.

Operation name: setElement

Argument information for Operation setElement

Direction	Type	Name
Input	int	new_size
Input	int	new_start_address
Return	void	None

The following public method is a modifier and sets the new size and starting address of the block.

Operation name: getEmpty

Argument information for Operation getEmpty

Direction	Type	Name
Input	None	None
Return	boolean	None

The following public method is an assessor and returns a Boolean value informing if the block of memory is empty.

Operation name: setEmpty

Argument information for Operation getEmpty

Direction	Type	Name
Input	boolean	p_empty
Return	void	None

The following public method is a modifier and sets the block of memory to empty.

Operation name: setStart_address

Argument information for Operation setStart_addr

Direction	Type	Name
Input	int	p_start_addr
Return	void	None

The following public method is a modifier and sets the start address of the block.

Operation name: getStart_address

Argument information for Operation getStart_address

Direction	Type	Name
Input	None	None
Return	int	None

The following public method is an assessor and gets the start address of the memory block.

Operation name: setNext

Argument information for Operation setNext

Direction	Type	Name
Input	final DLNode	p_next
Return	Void	None

The following public method is a modifier and sets the next node of the memory list. The final used the object DLNode makes it a constant handle providing more integrity and the method cannot directly change them (Eckel, 2000, p.295).

Operation name: getNext

Argument information for Operation getNext

Direction	Type	Name
Input	None	None
Return	DLNode	None

The following public method is an assessor and gets the next block of memory.

Operation name: setPrev

Argument information for Operation setPrev

Direction	Type	Name
Input	final DLNode	p_prev
Return	void	None

The following public method is a modifier and sets the previous memory block of current node. The final used the object DLNode makes it a constant handle providing more integrity and the method cannot directly change them (Eckel, 2000, p.295).

Operation name: getPrev

Argument information for Operation getPrev

Direction	Type	Name
Input	None	None
Return	DLNode	None

The following public method is an assessor and gets the previous block of memory.

Operation name: setH_size

Argument information for Operation setH_size

Direction	Type	Name
Input	int	p_h_size
Return	void	None

The following public method is a modifier and sets the size of the memory block.

Operation name: getH_size

Argument information for Operation getH_size

Direction	Type	Name
Input	None	None
Return	int	None

The following public method is an assessor and gets the size of the memory block.

Relation information for Class: DLNode

Relation name: itsMemoryProcessing

Name	itsMemoryProcessing
Inverse	itsDLNode_1
Source	DLNode
Target	MemoryProcessing
Multiplicity	1
Type	Association
LinkName	Composite

Class name: MemoryController

The following class is a controller for all the requests, which are sent by the MemoryCaller and the responses returned by the MemoryManager. The MemoryCaller sends it requests encapsulated as parameters to the MemoryController. The parameter also has an identification number depicting the identity of the caller. MemoryController validates the request and stores the request on a request queue. The MemoryController class also has a method responsible for polling the request queue to remove any requests

within a timeframe. The analysis of the removed request stores the identification number of the MemoryCaller and passes the request to the MemoryMediator as a parameter of the requestAvailable() method. The response of the request is passed back to the MemoryController, which adds the response to the response queue. The removed response from the queue is mapped to the request identification number of the original MemoryCaller and sent to the MemoryCaller respectively.

Operation information for Class: MemoryController

Operation name: addRequest

Argument information for Operation addRequest

Direction	Type	Name
Input	Object	reqtype
Return	None	None

The RequestMemoryManager class of the MemoryCaller calls the following public method with the request encapsulated as the parameter. The parameter has the identification of the MemoryCaller and the method from the interface of the MemoryManager.

Operation name: validate

Argument information for Operation validate

Direction	Type	Name
Input	Object	reqtype
Return	None	None

The following public method validates the request sent by MemoryCaller. The validation criterion is to check if the request is from a legitimate caller and for an available service by the MemoryManager. If the request is not valid, a message is sent back to the caller. If

the request is valid it is added to the request queue by calling add() method from the MemoryRequestQueue and the MemoryManager is informed about the success.

Operation name: removeRequest

Argument information for Operation removeRequest

Direction	Type	Name
Input	None	None
Return	Object	None

The following public method called by MemoryController, polls the MemoryRequestQueue for requests on the queue by the MemoryCaller and removes them in a FIFO (First in First Out) manner from the queue.

Operation name: analyzeRequest

Argument information for Operation removeRequest

Direction	Type	Name
Input	Object	None
Return	None	None

The following public method analyzes the request, which is removed by the removeRequest method. The main objective of this method is to decompose the object to store the identification of the MemoryCaller who sent the request. This method informs the MemoryMediator by sending the request (service request) as a parameter for the requestAvailable() method of the MemoryMediator class.

Operation name: addResponse

Argument information for Operation addResponse

Direction	Type	Name
Input	Object	None

Return	None	None

The MemoryMediator class calls the following public method and has the response, which is the MemoryStatus, encapsulated as a parameter. The response is added on the MemoryResponseQueue and a message is sent to back to the sender reporting success.

Operation name: removeResponse

Argument information for Operation removeResponse

Direction	Type	Name
Input	None	None
Return	Object	None

The following public method is called by the MemoryController class, which polls the MemoryResponseQueue and removes the response from the queue.

Operation name: analyzeResponse

Argument information for Operation analyzeResponse

Direction	Type	Name
Input	Object	None
Return	None	None

The following public method analyzes the responses before they are sent to the MemoryCaller. In the analysis, the response is mapped to the request by comparing the MemoryCaller identification. The verified response is sent to MemoryCaller by invoking the sendResponse method of the respective MemoryCaller.

Relation information for Class: MemoryController

Relation name: itsMemoryRequestQueue

Name	itsMemoryRequestQueue
Inverse	itsMemoryController

Source	MemoryController
Target	MemoryRequestQueue
Multiplicity	1
Type	Association

Relation name: itsMemoryResponseQueue

Name	itsMemoryResponseQueue
Inverse	itsMemoryController
Source	MemoryController
Target	MemoryResponseQueue
Multiplicity	1
Type	Association

Relation name: itsMemoryRequestQueue_1

Name	itsMemoryRequestQueue_1
Inverse	itsMemoryController
Source	MemoryController
Target	MemoryRequestQueue
Multiplicity	1..*
Type	Association

Relation name: itsMemoryResponseQueue_1

Name	itsMemoryResponseQueue_1
Inverse	itsMemoryController
Source	MemoryController
Target	MemoryResponseQueue
Multiplicity	1..*
Type	Association

Class name: MemoryRequestQueue

Attribute information for Class:MemoryReqestQueue

Name	**Type**
head	int
tail	int
queue	Object
size	int
numberInQueue	int

The following class implements an independent queue to handle the memory requests from the MemoryCallers.

Operation information for Class: MemoryRequestQueue

Operation name: MemoryRequestQueue

Argument information for Operation MemoryRequestQueue

Direction	Type	Name
Input	None	None
Return	None	None

The following is a constructor for the MemoryRequestQueue where initialization of the queue takes place. The size, head, and tail are initialized for the queue operations.

Operation name: add

Argument information for Operation add

Direction	Type	Name
Input	Object	obj
Return	None	None

The following public method is called by the MemoryController to add the requests to the queue. The size is updated as the elements are inserted in the queue.

Operation name: polling

Argument information for Operation polling

Direction	Type	Name
Input	None	None
Return	boolean	None

The following public method is called by the MemoryController to verify if any requests are available on the request queue. A Boolean value is returned. The verification continues until something is found or a time-out event occurs.

Operation name: remove

Argument information for Operation remove

Direction	Type	Name
Input	None	None
Return	Object	None

The following public method is called by the MemoryController to remove the requests from the queue. The size is updated as the elements are removed from the queue.

Relation information for Class: MemoryRequestQueue

Relation name: itsMemoryController

Name	itsMemoryController
Inverse	itsMemoryRequestQueue
Source	MemoryRequestQueue
Target	MemoryController
Multiplicity	1
Type	Association

Relation name: itsMemoryController_1

Name	itsMemoryController_1
Inverse	itsMemoryRequestQueue_1
Source	MemoryRequestQueue
Target	MemoryController
Multiplicity	1
Type	Association

Class name: MemoryResponseQueue

Attribute Information for Class: MemoryResponseQueue

Name	Type
head	int
tail	int
queue	Object
size	int
numberInQueue	int

The following class implements an independent queue to handle the memory responses to the MemoryCallers.

Operation information for Class: MemoryResponseQueue

Operation name: MemoryResponseQueue

Argument information for Operation MemoryResponseQueue

Direction	Type	Name
Input	None	None
Return	None	None

The following is a constructor for the MemoryResponseQueue where initialization of the queue takes place. The size, head, tail are initialized for the queue operations.

Operation name: add

Argument information for Operation add

Direction	Type	Name
Input	Object	obj
Return	None	None

The following public method is called by the MemoryController to add the response to the queue. The size is updated as the elements are inserted in the queue.

Operation name: polling

Argument information for Operation polling

Direction	Type	Name
Input	None	None
Return	boolean	None

The following public method is called by the MemoryController to verify if any response is available on the response queue. A Boolean value is returned. The verification continues until something is found or a time-out event occurs.

Operation name: remove

Argument information for Operation remove

Direction	Type	Name
Input	None	None
Return	Object	None

The following public method is called by the MemoryController to remove the responses from the queue. The size is updated as the elements are removed from the queue.

Relation information for Class: MemoryResponseQueue

Relation name: itsMemoryController

Name	itsMemoryController
Inverse	itsMemoryResponseQueue
Source	MemoryResponseQueue
Target	MemoryController
Multiplicity	1
Type	Association

Relation name: itsMemoryController_1

Name	itsMemoryController_1
Inverse	itsMemoryResponseQueue_1
Source	MemoryResponseQueue
Target	MemoryController
Multiplicity	1
Type	Association

Class name: MemoryMediator

The following class acts as a mediator between the MemoryController and MemoryProcessing. It hides the objects created in both the classes from each other providing independence.

Operation name: requestAvailable

Argument information for requestAvailable

Direction	Type	Name
Input	Object	None
Return	None	None

The following public method is called by the MemoryController once a request is available. This method extracts the request from the MemoryController passed as an encapsulated parameter. The request is analyzed and the appropriate methods are invoked for processing. This method performs the logic of negotiation and knows the appropriate objects to which the request is sent.

Operation name: responseAvailable

Argument information for responseAvailable

Direction	Type	Name
Input	Object	None
Return	None	None

The MemoryProcessing calls the following public method once a response (MemoryStatus) is available. This method passes the response to the MemoryMediator as a parameter. This method performs the logic of negotiation and knows the appropriate objects to which the response is sent.

APPENDIX F

Operating System Package

The OperatingSystem component comprises of the OperatingSystemPackage, which encapsulates all the internals of the component in the package. The OperatingSystemInterface is the only exposed class of the component.

Class name: OperatingSystemInterface

The following is the OperatingSystemInterface to the MemoryManager and the MemoryCaller. The signature of the getHeapHandle() method is defined in the interface, which is invoked by the MemoryCaller.

Operation information for Class: OperatingSystemInterface

Operation name: getHeapHandle

Interface definition of the method. It is not implemented in this class.

Class name: CallMemoryManager

The following class implements the OperatingSystemInterface.

Operation information for Class: CallMemoryManager

Operation name: getHeapHandle

Argument information for Operation getHeapHandle

Direction	Type	Name
Input	int	heapsize
Return	MemoryProcessing	None

The MemoryCaller invokes the following public method. It creates a new instance of MemoryProcessing class and passes it the size of the master block, which is defined by the OperatingSystem. MemoryProcessing class of the MemoryManager is initialized with a size of the master block to create the nodes for storing data in the heap. The MemoryManager handles the determination of the fixed size blocks. OperatingSystem passes the heap size and after processing a handle is returned back by the MemoryManager, which is further passed on to the MemoryCaller.