1-1-1998

A flexible architecture for manufacturing planning software maintenance

Yousif A. Mustafa

A FLEXIBLE ARCHITECTURE FOR MANUFACTURING PLANNING SOFTWARE MAINTENANCE

by

YOUSIF A. MUSTAFA

DISSERTATION

Submitted to the Graduate School of

Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

1998

MAJOR: INDUSTRIAL ENGINEERING

Approved by:

[illegible] 12/11/97

Advisor Date

[illegible]

[illegible]

[illegible]

[illegible]

To the memory of

my father and brother

ACKNOWLEDGMENTS

I would like to thank, almighty Allah first and foremost for giving me the will, energy, and guidance without which I couldn't accomplish this prestigious degree . I also would like to express my gratitude to my advisor Dr. Olugbenga Mejabi, with whom I had disagreements most of the time, for his support. I have learned a lot from Dr. Mejabi and he opens my eyes on many critical issues. I appreciate all the support of my committee members especially Dr. Donald Falkenburg for his unconditional support..

My appreciation and thanks to those few who encouraged and supported me throughout my journey to the highest academic level there is, the Ph.D. My appreciation to my dear wife Feryal, and children Shayma, Sumaya, and Amina. All the help and support that they gave me, even those little things like typing a report helped to get closer to my goal.

Finally, I would like to express my thanks to all the women and men who cared and helped during my journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

FLEXIBILITY IN MANUFACTURING PLANNING SOFTWARE

1.1. Introduction

Computer software systems took on a new role in manufacturing planning with the introduction of Material Requirement Planning (MRP) in 1965. These systems generate material requirement lists in response to given production requirements. In this way, inventory management, purchasing, and shipping activities are linked to manufacturing. In 1979, Manufacturing Resource Planning (MRP II) systems were introduced [VerDuin 1995]. MRP II typically includes planning applications, customer order entry, finished goods inventory, forecasting, sales analysis, production control, purchasing, inventory control, product data management, cost accounting, general ledger processing, payables, receivables, and payroll [Turbide 1995]. An emerging market is developing for software systems that expand the scope of MRP II farther to encompass activities for the entire organization. Among these systems are Enterprise Resource Planning (ERP), Customer-Oriented Manufacturing Management System (COMMS), and Manufacturing Execution Systems (MES). These systems integrate marketing, manufacturing, sales, finance, and distribution to move beyond optimizing production alone, to optimizing the organization's multiple objectives of low cost, rapid delivery, high quality, and customer satisfaction [VerDuin 1995].

MRP II is the dominant solution for manufacturing in tens of thousands of companies. These companies range in size from less than a million dollars in sales right up to the top Fortune 500 companies. By the end of 1990, there were 90,000 MRP II systems installed in the U.S. However, this is a market penetration of only 11% which clearly shows the size and potential of the opportunity for MRP II development. Yet, despite the commonality of needs across the scope of manufacturing, there are distinct differences when comparing plant to plant, company to company, and industry to industry. The specifics of planning, manufacturing, and distribution vary considerably. One MRP II system, therefore, may lack features that allow proper definition of the products or processes of some industries. Often MRP II has to be modified to adapt to a particular situation [Turbide 1993]. This modification often pushes the cost even higher and makes MRP II more out-of-reach for many companies. Therefore, it would be highly beneficial for the overall scope of manufacturing if a highly flexible low-cost MRP system can be developed.

1.2. Software Development and Maintenance

Many software development methodologies including the traditional structured approaches such as the *waterfall*, *spiral*, and the *fountain* approach, as well as the object-oriented approaches such as Abbot (Abbot),OOSA (Shlaer and Mellor), GOOD (Seidewitz), OOSD (Wasserman), Booch (Booch), OOA (Coad and Yourdon), and the OMT (Rumbaugh) usually go sequentially through a sequence of steps called the System Development Life Cycle (SDLC). The Life Cycle typically consists of Analysis, Design, Implementation, and Maintenance phases. For complex software systems, the life cycle may cover many months, or even years, before the system is satisfactorily completed. Since software is meant to represent the real-world, that changes continuously, the software, therefore, has to be continuously maintained to accommodate these changes. This makes maintenance particularly important in the development cycle. Software maintenance, according to the IEEE 1990 Glossary of terms, is defined as "the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment".. In fact, software maintenance, according to [Li & Henry,1993], is one of the most difficult and costly phases of the software development cycle.

Software systems often have to be maintained, customized, or modified because :

1. Changes have to be made to meet different users' requirements and demands. For example, an MRP system designed for an auto manufacturer may require different functions from the same system designed for a pharmaceutical company. Many software products are only effectively used when they are customized to closely match the way in which each company conducts business. Thus, there might arise the need to customize the basic package for different users.
2. The software may need to be updated over time as developers learn more about user requirements. Software often goes through an evolutionary process due to lack of availability of complete information on user requirements at the time of initial system development. New information and functionality might be needed to enhance system performance or expand its scope.
3. A software might need to be changed due to errors made, despite extensive testing, during requirements analysis, design, implementation, or data collection.
4. Some software projects start with an initial prototype to verify user's requirements, which is then modified and extended to develop a complete and fully functional system.
5. Many application software environments such as the hardware or software platforms

change over time and somewhat unpredictably. Software applications therefore, require the ability to adapt to changes in the operating environment.

[Cox 1993] refers to change as the "enemy" to software development. He reported the following statistics about a group of software projects developed for the Federal government to highlight what he calls "the software crisis":

- 47% of the software projects paid for but never delivered.
- 29% delivered but never used.
- 19% abandoned or reworked.
- 3% used after change.
- only 2% used as delivered.

Software maintenance could be either: (a) corrective, to fix software defects discovered during operation; (b) perfective, to enhance performance; (c) or adaptive, to add features to the software in a changing environment [Ramamoorthy et al. 1984]. These modifications may sometimes be lengthier, expensive and even more difficult than the original development. It is important, therefore, to point out that software maintenance can be easily performed only if the software is *flexible*. This reflects the need for flexible software which can be easily developed and maintained with a minimal effort.

The following illustrative example highlights a typical need for modifying a manufacturing software system so it can be implemented in a different application domain.

An Inventory Control System within an (MRP) package for auto parts manufacturing plant ensures that all materials are available when and where needed, in the correct quantity and at the right time. In order to obtain the raw materials required for a particular shift, the gross production plan must be processed to obtain a list of items required and the quantity of each. The Inventory Control System must then locate these items and take them out of inventory so that production may start. As shown in Figure 1.1, the various functions of the Inventory Control System may include: *processProductionPlan, checkInventory, pullItem, checkSafetyLevel* and *printPurchaseOrder* modules. The *processProductionPlan* module processes the gross production plan, one finished product at a time. It identifies the items, and their corresponding quantities, required to produce a target finished product using the bill of material (BOM) database. It then passes the information to the next module, *checkInventory* module, in order to check whether the item is available in the required quantity by checking the quantity-on-hand for that item in the Inventory database. If the item is not available in the required quantity, control is passed to the *printPurchaseOrder* module which accesses the Inventory database, creates and then prints a purchase order for that item. If a sufficient quantity is available, the *checkInventory* module passes the item-id and its required quantity to the *pullItem* module which updates the quantity-on-hand in the Inventory database, and prints the physical location of the target item in the warehouse along with the required quantity. The *pullItem* module then passes the item-id to the *checkSafetyLevel* module which checks if the quantity-on-hand for that item is below its safety stock as indicated in the Inventory database. If below the safety stock, control is passed to the *printPurchaseOrder* module to generate a purchase order for that item.

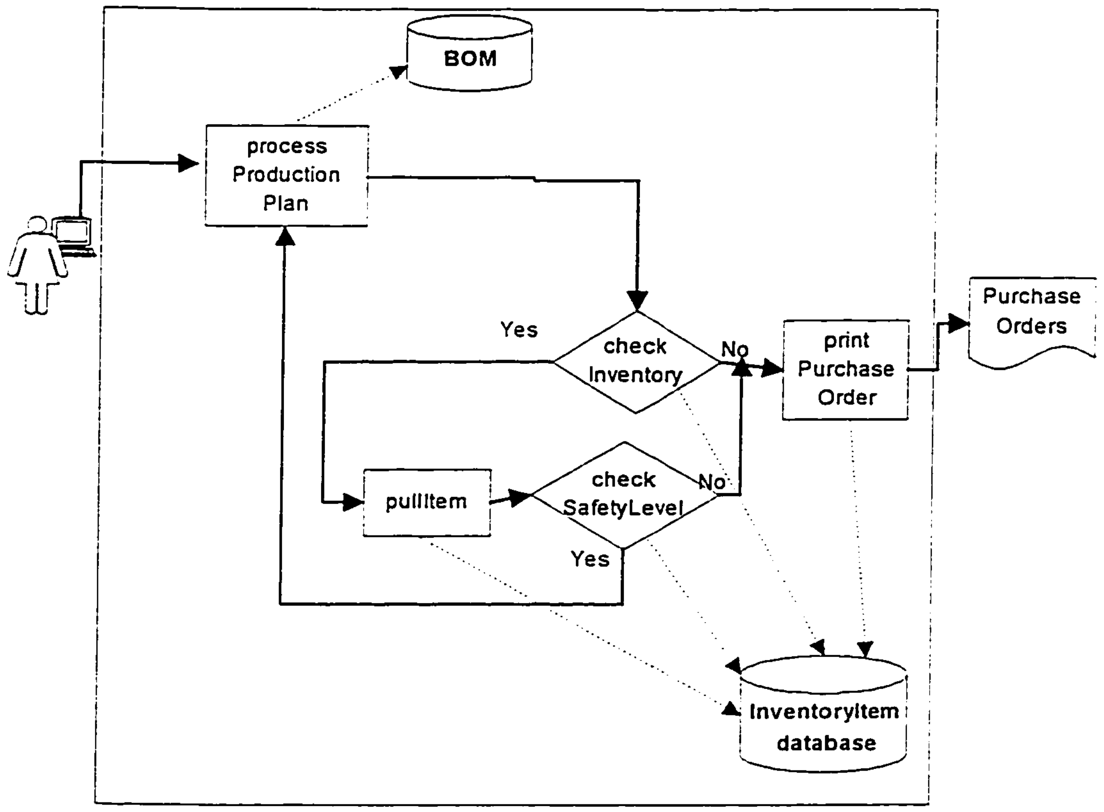


Figure 1.1, The Inventory Control System before modifications

In order to modify the system just described to make it applicable for use in a different environment, such as a pharmaceutical manufacturing plant or a nuclear reactor, several modifications may be required. The new environment requires a high level of security, as well as special procedures and policies for handling, shipping and disposal of materials. The functional modifications, as shown in Figure 1.2, are required to achieve the desired change in the system. Subsequently, some structural modifications may also be required in order to achieve the new functionalities. Examples of these changes are:

1. An "Access Permission" database has to be created to maintain an identification code for each personnel in the plant along with his/her corresponding access permission for requesting different items.
2. A *getAccessPermission* module should be developed to receive the requester's

identification code, use it as a look-up key to identify the requester's access permission from the Access Permission database, and append the requester's access permission to each item before it is passed to the next module, *checkAccessPermission.*

3. A *checkAccessPermission* module should be developed to perform the task of determining whether a requester is authorized to request certain item by matching the item access code of the Inventory database with the requester's access permission, before proceeding to the next module *checkInventory*.
4. The *processProductionPlan* module must be modified so that it invokes the *getAccessPermission* and *checkAccessPermission* instead of going directly to the *checkInventory* module. The *processProductionPlan* module must also be modified so that it prompts the requester and asks for his/her identification code.
5. In order for the *checkAccessPermission* module to function properly, the Inventory database has to modified so that each item will be assigned an item access code to control access to it.
6. The Inventory database has to also be modified, in order to make the system capable of addressing the needs to special handling, shipping, and disposing of the various items due to their chemical or physical hazardous nature to ensure safety. An additional field needs to be added to each record to indicate any special handling requirements.
7. The *pullItem* module has also to be modified in order to print the special handling procedures and policies.

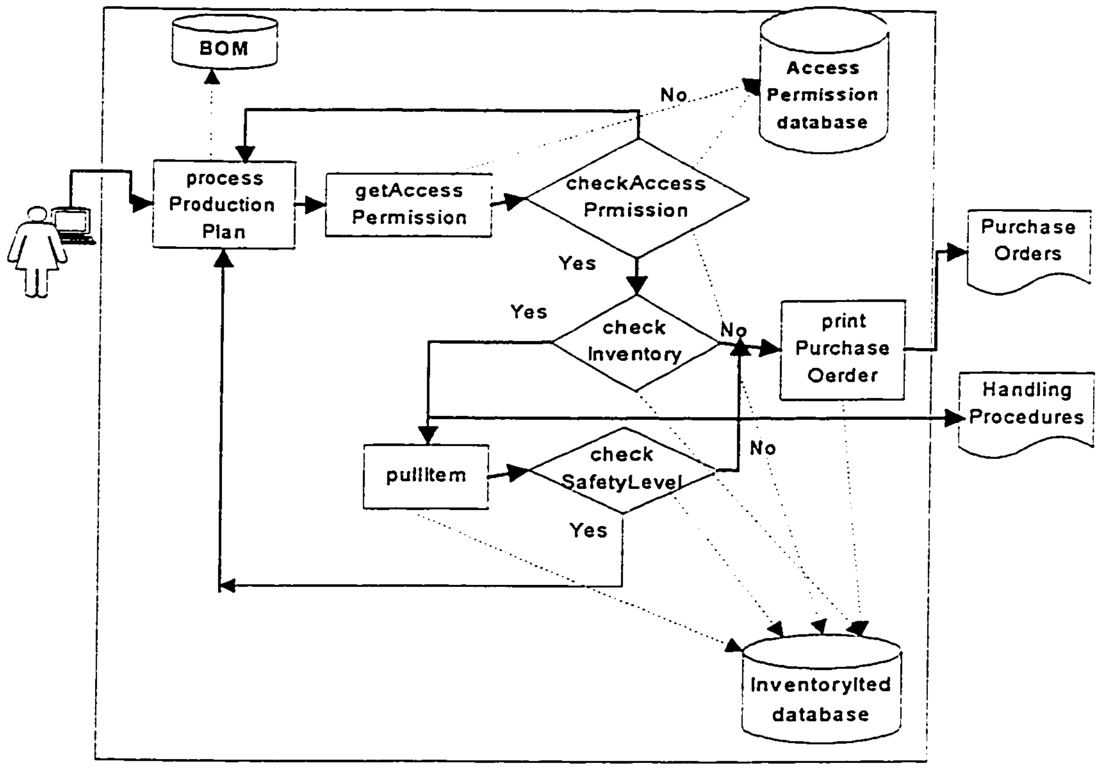


Figure 1.2, The Inventory Control System after modifications

1.3. Flexibility and Cost of Manufacturing Software Maintenance

Effective manufacturing software should not be based upon rigid processes. Rather, successful systems must be easily modified, fundamentally sound, and agile enough to adapt to frequently changing environment. Implementing inflexible, closed proprietary systems is dramatically opposed to a strategy that recognizes the increased degree of unpredictability in today's marketplace [Chamberlain et al. 1995].

Manufacturing software systems have to be responsive to today's manufacturing climate characterized by flexible manufacturing systems, high variety product mixes, shorter lead times, unexpected customer orders, and higher quality.

In order to easily maintain a software system, the software has to be *flexible*. In fact, as a general rule for software development, a basic assumption is established that the requirements of a system will always change; the question is only when and to what extent. Actually, the requirements change continuously; we only find this out periodically, therefore ***flexibility*** should be a primary criterion by which software systems are measured [Schapiro 1989]. Often, the task of maintaining a rigid software is very costly in terms of money, time, and human resources. Studies indicate that the cost of maintaining a software, as shown in Figure 1.3, increases at a geometric rate over the course of software development. The reason for this is that maintenance requires repeating many of the development tasks such as analysis, design, and implementation that were performed earlier [Burch 1992].

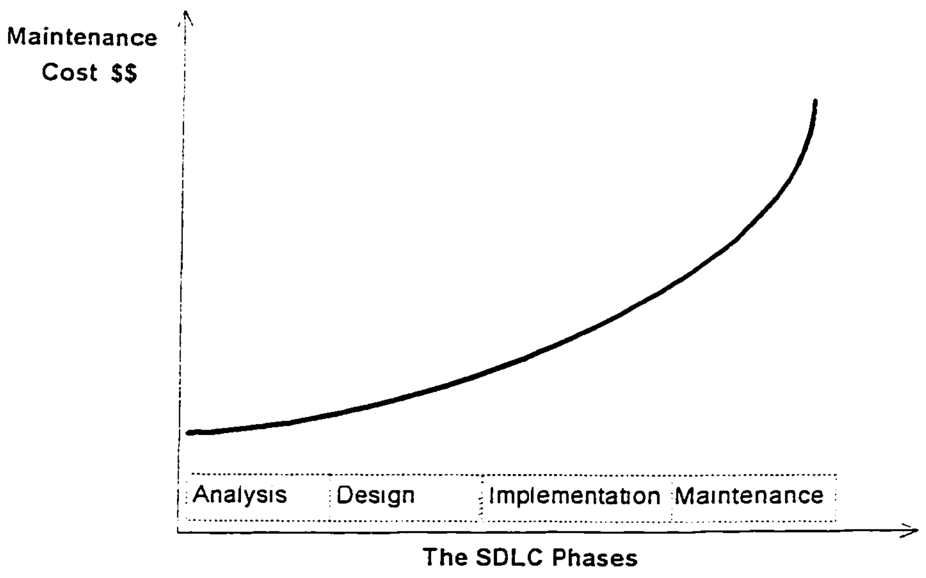


Figure 1.3, The cost of software maintenance

The following few examples reflects the high cost of software maintenance:

- In 1994 Hewlett-Packard utilized 60-80% of its software development personnel to maintain between 40-50 million lines of code [Coleman et al. 1994].
- The Federal government alone spends more than $5 billion a year on software maintenance [Borchardt et al. 1995].

- The Department of Defense indicates that it cost between $30 to $50 to write one line of code and about $400 to maintain it [Burch 1992].
- In general, software maintenance costs about five times the cost of the software license itself [Parker 1997].

Sometimes it may even be better to scrap the original system and start from scratch, as many programmers sometimes say "I can develop it quicker than I can understand somebody else's code" [Cox et al. 1991]. On the other hand, if the software is ***flexible***, it will be able to gracefully accommodate major as well as minor changes. The modification cost and effort will be considerably lower, and the software development cycle may be shorter since any new changes will be easily incorporated. It is reasonable to expect that the maintenance cost will often be higher with inflexible software systems. On the other hand, systems with high flexibility will require minimal effort and cost to be maintained.

1.4. Modes of Software Maintenance

An important distinction between compile-time and run-time software maintenance has to be highlighted. Compile-time maintenance involves modifications to source code through at least one Edit-Compile-Link-Test cycle. This is the traditional method of software maintenance which practically permits any change to be carried out. The compile-time approach is, however, often lengthy, expensive, and requires "shutting-off" the system. On the other hand, run-time maintenance is modifying the software while it is in actual operation. The run-time approach is the only option for time-critical manufacturing operations. A auto plant, for example, may require run-time maintenance of its control system because serious disruption and financial loss may result from "shutting the system down" even for few minutes.

Chapter 2

LITERATURE REVIEW AND PROBLEM STATEMENT

2.1. Literature Review

The subject of flexibility in software development has been dealt with in the literature in many different contexts. Some of the literature deals with the subject from an analysis, design, and software reuse stand point, while others focus on specific types of software flexibility issues such as adding or deleting classes. Some authors report research dealing with changes to database schema, these include the "Orion" database in [Kim et al. 1987], "Gemstone" in [Penny et al. 1987], "Iris" in [Fishman et al. 1987], "Vishnu" in [Moore et al. 1988], "Cactis" in [Hudson et al. 1988], and the "Mokum" database in [Riet 1989]. Others deal with the subject in terms of flexible operating systems such as the "Muse" operating system in [Yokote et al. 1989], or in terms of a prototyping tool for developing a flexible user interface as in [Morsi et al. 1991].

In this section the literature will be categorized, according to the approach used. Each approach will then be discussed to highlight its strengths and weaknesses. The categories developed are as follows:

1. The Software Reuse Approach
2. CLOS
3. CASE
4. The Template-Parametric-Modeling
5. The Conversion Approach
6. The Versioning Approach

7. Machine Learning

8. The Functionally-Defined Data Approach

9. The Graph-Based Approach

10. Meta-Object Approach

11. Relevant-Class Approach

12. Collection-Keepers Approach

13. The Indexing Approach

14. The Object-Specialization Approach

15. The Views Approach

2.1.1. *Software Reuse*

According to [Shriver 1987] reusing and reworking software is not new: it has been done since the very beginnings of our industry in the early 1950's. Reuse means using an entity in a different contents from that in which it was originally used. This is often called "black-box" reuse. When an entity is modified before it is used in the new setting, it is called "rework" or "white-box" reuse. [Tracz 1987] claims that the technical foundations from making software reuse a viable alternative to program development have been demonstrated. [Tracz 1987] also suggests that credibility to the used-program business has been accomplished and the technology of the future has been settled. [Gargaro et. al. 1987] propose the following criteria to follow when writing a program to be reusable:

1. be transportable (platform independent).

2. language independent.

In [Barnes 1991] the claim is made that the defining characteristics of good software reuse is not the use of software per se, but the reuse of human problem solving. Any work product that makes problem solving accessible, such as requirements specifications, designs, code, modules, documentation, test data, is a good candidate for reuse.

[Rine 1991] states that the notion of software reuse has been around for a long time beginning, for instance with the early notion of subroutines, which in turn led to the notion of software modules, as well as the numerous ideas about subroutine libraries and the use of modules in modern language development.

[Frakes et. al. 1994] indicate that software reuse is broadly defined as the use of engineering knowledge or artifacts from existing systems to build new ones. It is a technology for improving software quality and productivity. Systematic reuse is a domain focused on a repeatable process, and concerned primarily with the reuse of higher level life cycle artifacts, such as requirements, designs, and subsystems. A systematic reuse is defined as an application area, or more formally, a set of systems that share design decisions.

[Frakes et. al. 1994] identify software reuse as an important area of software engineering research that promises significant improvements in software productivity and quality. Reuse has proven to be a complex area affected by many factors. There are two basic technical approaches to reuse: parts-based and formal language- based. The parts-based approach assumes a human program integrating software parts into an application by hand. In the formal language-based approach, domain knowledge is encoded into an application generator or a programming language.

In the Object-Orientation paradigm (O. O.), the concept of reuse has been identified and introduced by many authors and practitioners in the field. [Meyer 1987] states that if one accepts that reusability is essential to better software quality, the object-oriented approach- defined as the construction of software system as structured collection of data types implementations- provides a promising set of solutions. An abstract data type is a class of objects characterized by the operations available on them and the abstract properties of these operations.

[Lewis et. al. 1991] have conducted an empirical study of the O.O. paradigm and software reuse and reported that the O.O. paradigm has a particular affinity to reuse. [Griss et. al. 1994] state that: "So far we have seen significant gains in productivity when reusing small-to-medium grain objects. This refers to both individual programmers and projects that I have seen stock piling objects abstracting common computation functions well as the use of commercially available object libraries such as Galaxy, InterViews, RougeWave, and Booch Components. I would also classify the success of Visual Basic in the same category. It illustrates, that there can be a market for small-to-medium objects and frameworks that can be used to make flexible and useful assemblies".

There have been many advancements and projects in the field of O.O. software reuse, the following section presents an overview of the most important ones.

[Lenz et. al. 1987] developed a library of building blocks oriented toward systems programming. The library is built on an extension to PL/S IBM-370 based language and a tool called BB/LX to allow the implementation of building blocks. The library contains building blocks such as: stack, queue, chain, list, table, map and sequence. Moreover there are building blocks for storage management and allocation. On the average, there are

10-20 operations available on each building block. Procedural building blocks include message handlers, command parsers, and input checkers. In order to call and use any building block from the library, developers have to write special BB/LX calls in their source code. Each program that needs to reuse a building block begins with an invocation of the "With" statement, which functions like the Ada "With" clause. The invocation also specifies the operating system environment, whose services are used when required (like for storage allocation). Building blocks operations are invoked through the "Do" statement and the developers have to identify where the building block should be inserted. Upon calling a building block, a correspor ding code will be fetched automatically from the building block macro-library to the target program. BB/LX is invoked by a building block operation and interprets it using the building block code. As a result, code is generated into the user program.

[Kaiser 1987] developed an O.O. language called Meld which has two essential aspects: (a) *Features*: a number of reusable building blocks which is a group of classes. A feature is larger than a subroutine and similar to an Ada package, it permits the reuse of the glue code among subroutines and abstract data types. (b) *Action-equations*: which are developed to define the relationships that must hold among classes and the dynamic interaction among classes and between classes and external agents such as users and operating systems. Meld is implemented by translating each feature into a conventional programming language and by a special run-time environment that support execution of systems built from features. [Kaiser 1987] derived the implementation algorithm from previous work on software generation, specifically language-based editors, where performance as good as handcoded editors has been achieved.

[Bassett 1987] introduced the concept of a *frame*, where a frame is a model solution to a class of related problems containing pre-defined change points. [Bassett 1987] built, Using COBOL, a library of three types of frames, (a) data-view frames to define data fields, (b) screen and report frames, and (c) one custom specification frame per program. Specification frames are created by programmers from template specification frames. The specification frame is automatically processed including compile and link operations. Frames do more than containing source code, they facilitate the design, manufacture and maintenance of software. In design, a generic frame is a rigorous parameterized analysis of a problem domain. Construction is an automated assembling process where frames play the role of standard subassemblies. Specifications in the form of screen and report definition, relational data frames are translated into executable code.

[Burton et. al. 1987] utilized the SoftCAD package to build a Reusable Software Library (RSL) to support the design and implementation phases of the SDLC. The library contains functions, procedures, packages, and programs. Using SoftCAD, a designer may specify the top-level design of a system by generating object-oriented graphs and Ada Program Design Language (PDL). The Ada PDL may be used as a template to produce the detailed design and, ultimately the source code. A program was developed on the VAX11/750 that automatically scans PDL and source code files and extracts specially labeled reuse component statements.

[Ratcliffe 1987] discussed a number of guidelines implemented regarding reusability in a project called "*ECLIPSE*". The library created for that project is characterized by:

1. Components are discrete, in Ada components are called "packages" and they model

objects as abstract data types

2. Interfaces are well defined with low coupling between components.

3. There are no subtle side effects.

4 Information hiding is used to large extent.

At Hewlett-Packard, [Berlin 1990], implemented the concept of reuse in their CLOS-based hypertext platform. The extent of the software reuse implemented in the project is integrating subsystems comprising over 200 classes and developing a glue code to integrate them.

[Dunn 1991] performed domain analysis for the manufacturing of electronic charts domain at Sperry Machine Inc., and a systematic reuse library was created to be used later in the development of four software systems. The systematic library consists of 20 classes having 133 methods. A glue code was required to incorporate the reusable classes as well as code for the developing the main driver and modifying the existing code in the reused classes.

[O'Connor et. al. 1994] define *synthesis* reuse methodology, where synthesis is an approach based on domain-specific reuse. Synthesis is a way to construct software systems as instances of a family of similar systems. The approach allows an organization to leverage the commonality among similar systems by creating standardized requirements and designs and their corresponding reusable components. At Rockwell International, [O'Connor et. al. 1994] created a synthesis-based project where an engineer would describe high level requirements and design decisions. Developers could then select, adapt, and compose the relevant reusable components based on the requirements and engineering decisions.

[Weide et. al. 1994] advocate the use of a systematic black-box style of reuse without source code modification They propose casting algorithms as objects just like defining abstract data types as objects. These algorithmic objects are designed to accept generic data types which will add a good deal of flexibility to the reuse process.

[Joos 1994] summarizes her experience at Motorola in implementing the software reuse approach. A library was built containing a "*Toolkit*" which consists of design capture, reuse engineering, and forward engineering. The developers, at Motorola, would then follow the procedure listed below in order to accomplish the needed reusability:

1. The user enters a set of requirements for which the reuse Toolkit finds a matching reusable software component.
2. Based on the user input, the reusable software components are customized to meet the software requirements.
3. Reusable software components are interfaced to new components.
4. The automated testing tools test the new software product.

[Lim 1994] reports on a manufacturing system development project at Hewlett-Packard. He defines reuse as the use of products such as code, design, and test plans, without modification, in the development of other software. Leverage reuse, on the other hand, is defined as modifying existing products to meet specific system requirements. [Ning et. al. 1994] developed, at Anderson Consulting, an Architecture-driven, Business-specific, and Component-based (ABC) system development approach based on the premise that component-based assembly/interconnection of domain-specific-components has the potential for achieving significant productivity gain and quality improvement. The domain-oriented library they used contains: requirement documentations, formal design, test script, and source code. Based on the client's requirements, the developer chooses a plausible design from the reuse library. The developer would then modify these components to match the client requirements. Finally, the reusable components are "glued" together, so packaging and installation procedures are generated and the final system is then delivered.

[de Cima et. al. 1994] developed a hypertext Design Environment for Reusable Object-Oriented System (DEROS) based on using *frameworks*. A framework is defined as a high level design or application architecture consisting of a set of classes that are specifically designed to be refined and used as a group. This set of classes embodies an abstract design for a family of problems (or problem domain) and encourages the software reuse in a granularity which is larger than a class. [de Cima et. al. 1994] used the white-box approach to frames, where a frame can be specialized by adding application-specific subclasses to it.

[Griss 1994] reports an ongoing efforts at Hewlett-Packard to encourage the implementation of systematic O.O. and promote the notion of domain-specific *kits*. A Kit consists of: reusable components, carefully designed *frameworks* (A framework refers to a set of collaborating classes from which an ensemble of interacting objects is generated primarily by inheritance) which captures the architecture and services for a family of related products, and some form of glue code which could be either a typical language such as C or C++ or some problem-oriented language. The kit also contains design tests, templates, macros, documentation, browser, glue-code editor, application builder, and generator.

[Rajlich et. al. 1996] introduced the concept of "Orthogonal Architecture" for reuse and developed a methodology for transforming one program into another program (within the same domain) in order to satisfy a new set of requirements. Their architecture consists of (a) six layers, where a layer consists of a number of classes, and (b) six threads, where threads are special cases of subsystems, and they consist of classes implementing the same functionality.

[Arnold et. al. 1997] report on the "San Francisco Project" at IBM Research Division. The project is currently in progress, and it utilizes the concept of *frameworks*, which is an O.O. infrastructure and application logic that can be expanded and enhanced by developers to provide far better software systems. A framework consists of the following three layers of extendible components: (a) the highest layer containing the core business processes to provide business objects and default business logic for selected "vertical" domains, (b) definitions of commonly used objects that can be used as the foundation for interoperability between applications, and (c) the base layer provides the infrastructure and services required to build industrial strength applications in distributed, managed-object, multiplatform applications. It contains object such as: command, entity, and Factory. Currently, the "San Francisco Project" contains one framework only, the General Ledger. Building application using "San Francisco Project" frameworks can be approached in three ways.

(1) Using the objects and classes without changing them. The Developer writes client code that uses a factory object to access the target framework objects.

(2) Modifying the framework by adding new domain classes from the third level of the framework. Developers who make these changes need to understand the methods and programming guidelines for creating, deleting, and updating framework objects.

(3) Modifying the frameworks by extending the supplied domain classes and methods. It may be necessary to add additional attributes to those defined for classes, or to replace the logic in one of the methods.

Reviewing all the previous software reuse approaches, one would make the following conclusions:

1. No reuse approach is specifically directed to implement classes and behaviors needed to develop MRP systems.
2. Most approaches reuse a program, module, framework, or a class as the basic building blocks to build an application. There is a need to reuse very atomic building blocks to gain more flexibility and productivity. According to [Griss et. al. 1994], the approach of reusing atomic building blocks (as well as small-to-medium grain objects), as opposed to a whole class or program results in a significant gains in productivity and there can be a market for small-to-medium objects that can be used to make flexible and useful assemblies [Griss et. al. 1994]
3. All the previous approaches require developing a "Glue Code" in order to integrate the reused components. This would require some skills in certain programming language.
4. Most of the approaches will require going through the cycle of "Edit-compile-link-test" which requires following the compile-time maintenance approach.

2.1.2. *CLOS*

Common Lisp Object System (CLOS) is an object oriented extension of Lisp, which is a high level programming language like COBOL or FORTRAN [Steele 1990]. It enables the programmer to define classes to model objects and their various relationships, and use these classes as building blocks to create new classes. CLOS provides tools that can help build modular and extensible programs. CLOS supports flexible means of redefining classes and methods so that the programmer has the freedom to modify the original design including the organization of classes [Keene 1989].

The basic concepts of CLOS are:

Class: Is an object which determines the structure and behavior of a set of other objects, which are called *instances*. Every CLOS object is an instance of a class. A class can inherit structure and behavior from other classes. A class whose definition refers to other classes is called a Subclass of those classes, conversely, each one of those classes is called a superclass of the inheriting class.

Slots: The structure of a class structure is defined in terms of slots. Each slot has a name and a value. The value of a slot describes the state of the slot at any given time. There are two kinds of slots: (a) *local slots* which stores information about the state of a particular instance. Each instance maintains its individual copy of the of the slot with its own value, and (b) *shared slot* which store information on the state of a whole class. There is only one value of a shared slot which is associated with the class and shared by all instances of the class [Kiczales et al. 1991].

Generic function: Is a function whose behavior depends on the class, it contains a set of methods. Each method defines the class-specific and operations of the generic function [Steele 1990]. A generic function specifies only the interface. while the implementation is distributed across a set of methods and is dependent on the classes of the method's arguments [Keene 1989].

Access to a slot of a given class is provided for instances of that class by an automatically generated generic function called an *accessor.*

Metaclass: A metaclass of a class is the class of its class, it contains the following information about the class: (a) class name, its direct superclasses, and slot specifications, (b) list of class's superclasses and their slots specifications, and (c) list of subclasses and link to methods that reference the class.

To define a class, in CLOS, the programmer utilizes a macro called *defclass*, and provides all the necessary information (a) through (c) above for each class.

CLOS also permits the programmer to redefine classes using the *defclass* macro. Upon redefining a class structure, which is adding or deleting slots, CLOS will replace the old structure with the new one. CLOS automatically propagates the changes to all the instances of the class and its subclasses. Any accessor method that was created by the old class definition will be removed. The programmer can use the *defmethod* macro to redefine a method [Keene 1989]. CLOS is also capable of changing the class of an instance using the *change-class* macro [Steele 1990].

The major advantages of the CLOS approach can be summarized as follows:

1. Enhances software flexibility.
2. It is a good tool for rapid prototyping and incremental software evolution.
3. It provides an automatic run-time change propagation mechanism upon changing class structure and object migration.

However, CLOS has the following weaknesses:

1. Changes to class structure and methods are only permitted to existing classes and methods. No new method can be added. According to [Keene 1989] to redefine an element of CLOS (such as a method) that element already exists.

2. All changes require some knowledge of the CLOS terminology and syntax in order to write the code needed to accomplish the required change.

2.1.3. *CASE*

Computer Aided Software Engineering (CASE) is the approach which uses computer-based support in the software development process [Brown et al. 1994]. CASE provides the automated support required to build a flexible, high quality software systems quickly. CASE first appeared in the mid 80's as a possible solution to one of the most frustrating problems in the area of information systems. That was the ability of information technology groups to quickly respond to customers' needs for information systems. CASE tools consist of a set of software products, such as graphical editors, user-interface design systems, and compilers, integrated through a common repository, that help automate every stage in the Software Development Life Cycle.

In order to discuss CASE flexibility techniques, three terms must be first defined:

Reverse engineering: Is the process of analyzing a system and representing its components and interrelationships in an abstract form, independent of the physical implementation. It involves recovering design specifications or requirements from existing source code or file definitions. Typically, some sort of graphical representation is the result of Reverse Engineering. The tools available for Reverse Engineering retain this information in a data dictionary or knowledge base.

Reengineering: Is the examination and alteration of a system to reconstitute it in a new form. Once a system has been Reverse Engineered into an abstract form, Reengineering is the modification of this abstraction to meet the new needs, or to make corrections, enhancements, or modifications.

Forward engineering: Is the process of going from a high-level abstraction (implementation independent) to the physical implementation of a system.

The following are some of the CASE techniques which provide software flexibility:

1. CASE tools for Reengineering are designed to assist analysts in understanding existing systems and to make it easier and less costly for them when making changes. CASE re-engineering tools process existing systems to extract a structured analysis of the system and its design. From there, the system can be logically enhanced or corrected, then forward engineered to produce the new version. The benefit of this approach, according to industry studies, is that a reengineerd project can reduce maintenance costs by 30% to 70%. Some organizations that have implemented CASE Reengineering tools report remarkable productivity boosts (as high as 300%), particularly in Reverse Engineering, of files and databases [Borchardt et al. 1995].
2. In a CASE environment, design, prototyping tools, automatic code generators, and a repository of well-tested modules help developers stay responsive to user's needs. Specifications are not frozen until very late in the design phase.
3. CASE allows a person, or a small team, to perform analysis, design, and coding. This is feasible because CASE subdivides complex projects into small, easily manageable modules that interact only minimally. Independent teams of two or three persons can then accomplish development tasks in parallel.
4. Reusing objects and modules is another technique that CASE implements in order to enhance flexibility, since many systems perform similar functions and use similar objects. Thus, these functions may be designed, coded, and tested once and then reused several times. These reusable functional modules, or objects, can be stored in the

CASE repository of an organization. The modules and objects can be reused whenever necessary and modified as required. Time scheduled for a project may be reduced significantly when reusable designs, data models, or code are used. According to [Aggarwal et al. 1995]., projects created primarily with reusable software need approximately 25% of the resources required for designing new software from scratch. Reusing proven software designs and modules provides higher productivity and minimizes errors [Aggarwal et al. 1995].

5. Some CASE packages implement maintenance tools that analyze the impact of a change, extract identifier names, list them in alphabetical order, and perform dataflow analysis. These tools may also check for program style and standards violations, compute a variety of complexity metrics, restructure unstructured source code, display the structure of an application, and generate documentation automatically. The Data Dictionary (DD) is a CASE tool for maintenance which creates and manipulates a dictionary-like database containing cross-references to names used in the source code of an application. The cross-references are obtained by running the source code through the DD. The output from the DD could be any of the following:

   a. *Module Map Relation*: Containing a listing of the static lexical structure of the application's source code.

   b. *Defined Relation*: Containing a list of all identifiers in the system, and where each one is defined.

   c. *Referenced Relation*: Containing a list of all identifiers and where each one is used.

   d. *Uses Relation*: Containing a list of all modules and indicating what identifiers each module uses.

e. *Interface Relation*: Containing a list of all modules and functions, and listing their interfaces.

[Baxter 1992] suggests that software development must focus on problem specification and justification for the design specifications. The final programs would then be generated using specially designed CASE tool. This CASE tool uses a library of heuristic methods coded into a Transformation Control Language (TCL). [Baxter 1992] introduces the concept of "design maintenance" where any change to the software will be directed toward the design rather than the code. A change in the design (called delta) will then trigger the CASE tool to transform the "delta" into a final program.

CASE has its weaknesses [Senn et. al. 1995], [Borchardt et. al. 1995] which can be summarized as follows:

1. One of the most enduring complaints concerning CASE is that separate tools are not integrated since most of them are developed for different platforms. They might also require powerful processing platforms in terms of capacity and capability [Senn et. al. 1995].
2. Vendors claim that restructuring tools in CASE can produce a functionally equivalent structured program from even the most tangled of source code, but it is important to distinguish between standard code and structured program logic. Restructuring tools can generate syntactic structure, but they are still incapable of logical structure . Until CASE tools advance to where they can consistently make logical inferences from illogical process flows, this must still be left to developers [Borchardt et. al. 1995].
3. Most CASE tools have poor code generation abilities (most generate C++ declaration

and headers only, with no member function body code). None of the tools have good documentation generation ability [Church et al. 1995].

4. In addition to all the above weaknesses of CASE, it should be clearly pointed out that CASE can only handle maintenance to the logical design, CASE is not capable of propagating any change to the physical underlying persistent data.
5. All the changes to the logical design are carried out by CASE at compile-time and not at run-time.

2.1.4. *Template-Parametric-Modeling*

This approach was proposed by [Lim 1996] utilizing the concepts of Object Oriented simulation using *Model Builder* [Mejabi 1993]. The objective is to build flexible simulation software models with minimal effort. The research introduces the concept of the *gross structure* of a simulation model. The gross structure of a model is a description of its building blocks and the relationships between them. The approach uses a template with many parameters to define the gross structural information of a simulation model. The system then uses the values of the parameters in order to build a model. The approach provides a library of various types of Parametric templates representing various application domains. The Parametric-Modeling approach [Lim 1996] allows the user to select a template which is most appropriate for his/her domain needs, then prompts the user to provide values for the parameters for that template. The system will then use these values to build the desired simulation model. The approach also allow the user to control the characteristics of entities in the simulation model.

Although the architecture proposed by [Lim 1996] supports flexibility, it is specifically designed to model simulation systems. The architecture was not designed to handle persistent data.

2.1.5. *The Conversion Approach*

[Kim et al. 1987], [Banerjee et al. 1987], [Kim et al. 1987], [Banerjee et al. 1987], [Penny et al. 1987], [Fishman et al. 1987], [Morsi et al. 1991], [Morsi et al. 1992], [Chung et al. 1992], and [Bertino et al. 1993] all describe schemes which propagate changes by directly modifying all the affected objects in the system. However, these methods differ in the changes allowed, limitations and restrictions on the changes, and their specific implementation techniques.

In [Kim et al. 1987], [Banerjee et al. 1987], [Kim et al. 1987], and [Banerjee et al. 1987], the changes allowed in their ORION databases are: changes to class structure, adding, deleting, renaming a class, and changing class/superclass relationships. Multiple inheritance where a class inherits part of its structure and behavior from two or more superclasses is available in ORION, which may improve the modeling power and flexibility of the database. The objects affected by a change in ORION are modified only when accessed by the user. This type of conversion is called *screening* or *lazy conversion.* Although the ORION data model has a lot to offer in terms of flexibility, it has five invariants that must be maintained at all times. Two of the invariants are:

(a) the *Full Inheritance Invariant*: In multiple inheritance, a class may inherit from two or more superclasses and the superclasses may have attributes, where attributes is a set of variables used to define the state of an object, with the same name. In such a case, if the attributes have different types, then only one attribute can be inherited.

(b) the *Default Compatibility Invariant:* The domain (a range of acceptable values that an attribute can take) of an inherited attribute must be a subset of its corresponding value in the superclass.

ORION also has twelve rules used to propagate changes, while preserving the five invariants. An example of the rules is the *Domain Change Rule,* which stipulates two restrictions on the type of changes in an attribute's domain. First, the domain of an attribute can not be reduced. Second, the domain of an inherited attribute can not be expanded beyond its value in the superclass. All these invariants and rules, combined, may limit the capabilities of the model. ORION does not permit dynamic inheritance which is the ability to change one or more of a class's superclasses at run-time. It does not permit object migration either where an object changes its class and becomes a member of another class at run-time. The authors do not address the full effects of adding or deleting attributes. For instance, if there is a method, which is a procedure inside a class used to define and implement aspects of the class behavior, that depends on an attribute, and that attribute is deleted, this may introduce some instability into the system.

In their IRIS database model, [Fishman et al. 1987] adopt the concept of a *type* which is defined as a named collection of objects sharing common behavior. Objects of the same type may not necessarily have the same structure. This is different from the class concept which, according to [Bertino et. al 1993], is a set of objects having exactly the same internal structure and behavior. Types are related to each other through inheritance relationships. A subtype inherits all the behavior of its supertypes. Furthermore, adding and deleting of types is allowed, and objects can migrate from one type to another. However, other types of changes are not permitted in the IRIS database. For example, it does not allow two types that are declared disjoint to be subtypes of a common type. A type cannot be deleted if it has no instances, and new subtype/supertype relationships among existing types cannot be created. The IRIS data model does not have the capability to handle dynamic inheritance or changes to object behavior.

[Penny et al. 1987] considered some of the changes in their Gemstone data model, such as adding or removing a class or an attribute by implementing the conversion approach similar to that used in ORION and IRIS. The only difference is that in Gemstone, the modification of affected objects takes place immediately to ensure data consistency. This is called *eager* conversion. The model has six invariants on class modifications. One example of these invariants is that the constraint (which is a rule used to enforce system integrity by stipulating that attributes must be defined before being used in a method) on an inherited attribute must be the same as, or a subclass of, the corresponding constraint in the superclass. Classes in Gemstone are not allowed to change their superclasses. Furthermore, the Gemstone data model does not permit changes to object behavior or object-class membership.

In [Morsi et al. 1991], where multiple inheritance is allowed in their Goose data model, the domain of an attribute inherited from multiple superclasses is restricted to be a subset or equivalent to the intersection of all corresponding domains in the superclasses. The changes available include changes to class structure and inheritance relationships. These changes are propagated to main memory and made permanent upon the issuance of a SAVE command by the user. The Goose model does not handle object migration or changes to object behavior. In [Morsi et al 1992], changes such as addition, deletion, and modification of classes, attributes, and methods are possible. They utilize the C++ class library in developing an approach which they call the open class hierarchy. In this approach, the system meta-information is implemented as objects of system classes used to maintain the database. The approach does not handle changes to object behavior. However, object behavior changes are discussed in the article as a future research project. They proposed a code transformer to translate the references between all methods, attributes, classes in the database, and their respective object identifiers into a source code. The code transformer will also: (a) produce a list of dependencies between all methods, attributes and classes, and (b) produce updated source code for the methods affected by schema change operations. The user would furthermore, be notified and required to make some compile-time changes whenever automatic method updates are not possible. Their proposed approach of keeping a list of all the dependencies between the components of the database seems to involve some data storage overhead penalty which may have a significant impact on system performance.

2.1.6. *The Versioning Approach*

This approach is characterized by propagation of changes by creating a version of each of the affected objects before modifications are carried out. As discussed by [Ahlsen 1983] and [Ahlsen 1984], there is a need in some applications such as the Office Information System (OIS), to keep track of the history of the system over time by keeping the state of all objects and classes before and after each change is made to the system. This may be done for an indefinite period of time. Once a change occurs, the affected objects are not updated. Instead, new versions are created and old versions of the objects are kept. A view mechanism is then used to dynamically map instances of different versions of the same type to the version which is needed by the user. It is the responsibility of the user to specify the version number that is required. Moreover, when the different versions are not compatible, they must be processed separately. The approach in [Ahlsen 1984] do not have the capability to handle addition and deletion of types, changes in inheritance relationships, behavior changes, or object migration. Their approach is developed primarily to address the specific needs of the OIS domain.

[Zdonik 1985] suggests that when an object is modified, a new version of that object should be created to reflect the evolution stages that the object passes through. The data model includes the notion of versions as a primitive concept. A version set is an ordered collection of objects, each of which represents a snapshot of the state of that object at some point in time. The decision about which changes should be propagated is left to the discretion of the database schema designer. This approach handles the special needs of the OIS and only deals with incremental changes. This method is not capable of accommodating radical changes such as inheritance relationship changes, behavior modification, and object migration.

[Narayanaswamy 1988] proposes a method for allowing a design object to evolve into a family of versions of that object without having to introduce new subtypes. This is accomplished by introducing a mechanism called *instance-inheritance.* When an object evolves, a new version is created and then linked to the old one by a *has-version* link. The has-version link is multi-valued, which enables any object to link to all its versions. Instance inheritance takes precedence over default inheritance. This approach is based on the assumption that over time, a given concrete object may evolve into a family of closely related objects. Each member of the family shares certain characteristics with the other members and differs in other respects. This concept is only suitable for specific applications where changes are parameter-driven (where changes to an object attribute can be computed in terms of another attribute). Any major change in the structure or behavior of an object could make that object entirely different from the rest of its family objects. This would violate the basic assumption of this approach, that an object should evolve into a family of closely related objects. This approach according to [Narayanaswamy 1988], has the disadvantage of requiring additional mechanisms to handle instance-inheritance, which might result in some degree of performance loss.

[Katz et al. 1990] suggest creating a version each time an object is changed. These versions are grouped in a version-hierarchy maintaining *is-a-descendant-of* and *is-an-ancestor-of* relationships among the versions of the same object. Their approach is specific to Computer Aided Design (CAD) databases and only addresses incremental changes. None of the major changes such as deletion and addition of classes, modifying relationships, and object behavior changes are addressed.

[Clamen 1994] supports schema evolution and program compatibility by suggesting that class evolution be defined by creating a new version of the class upon each change. Each class has multiple versions; any version can then be represented in terms of another version through a pre-defined relation. Each instance of a class is composed of multiple facets to provide multiple interfaces to the instances. Since changes are not propagated immediately, each facet has a flag to indicate whether it is up-to-date or not with respect to the most recently modified facet. Each application program interacts with the instances through a single version. [Clamen 1994] divides class attributes into four groups: (1) *shared*, when an attribute is common to all versions of the class, (2) *independent*, where its value cannot be affected by any change in the attribute values of another facet, (3) *derived*, which is directly derived from the values of the attributes in another facet, and (4) *dependent*, when an attribute is affected by changes in the values of attributes in another facet, but cannot be derived from those values. Modifications to an attribute on the primary facets are immediately propagated to other facets depending on the attribute category. This is accomplished by using certain dependence and derivation rules between the attributes. The types of changes handled in [Clamen 1994] are limited only to the changing of attributes. Classifying the class attributes into the four categories, and relating them through dependency and derivation rules is only applicable to cases where clear dependency relationships exist between attributes. This limits the type of changes allowed and consequently makes the system less flexible, since required changes may not always be parameter-driven. Also, the model does not have the capability to handle changes to class inheritance.

2.1.7. *Machine Learning*

[McLeod 1988] proposes an approach based on a combination of three types of machine learning techniques: learning from instruction (LFI), learning from exception (LFE), learning from observation (LFO), as well as a set of learning heuristics, to an object-oriented data model called PKM. This approach implements a set of algorithms encoded in the system to implement a number of changes. The approach is efficient and easy to use and is particularly attractive because of the ease with which it can be automated. However, these algorithms implement a fixed set of variables and relationships upon which the system can learn efficiently. No new variables or relationships can be introduced. Consequently, the changes that the system can handle are limited to fairly predictable changes. Therefore, the key issue in evaluating this approach is the trade-off between efficiency and flexibility. The learning approach might be suitable for well-defined domains, but is unsuitable for the drastic and often unpredictable changes.

2.1.8. *The Functionally-Defined Data Approach*

[Moore et al. 1988] and [Hudson 1989] proposed an approach of having certain dependent attributes derived from other attributes by using algorithms attached to the independent attributes.

The model, Cactis, as defined in [Hudson 1989] has both attributes and integrity constraints, where integrity constraints are rules used to enforce system integrity by stipulating that attributes must be defined before being used in a functionally-defined method. The system automatically re-computes the derived values whenever a change takes place and enforces constraints whenever updates are made. Attributes of an object are classified into two types: derived and intrinsic. Derived attributes have incremental attribute evolution rules attached to them, to show how the attribute value should be derived at all times. Intrinsic attributes are simply stored in the object. Some attributes could be derived from other attributes; these are called dependent attributes. After the system computes a new attribute value, the update is not always implemented immediately, but may be deferred until the next time the attribute is accessed. When one attribute is affected by the change of another attribute, it is marked as being out-of-date and given a label as "important" or not. If "important," the attribute will be changed immediately, if not, the update will be carried out when the attribute is next accessed. The model in [Hudson 1989] has the capability of creating and deleting objects and relationships between objects. Object membership in a type (class) can also be dynamically chosen.

The approach described in both [Moore et al. 1988] and [Hudson 1989] has the major difficulty of assuming that attributes have to be related to each other and that changes are parameter-driven. This assumption does not hold for many applications. Also, this approach does not handle any changes to object behavior. The approach used in Vishnu as described in [Moore et al. 1988] also has limitations resulting from potential name conflicts. According to the authors, a name conflict causes the system to crash. This occurs when, implementing multiple inheritance, a class inherits an operation with the same name from different superclasses. The only solution is for the user to explicitly resolve any such name conflicts. The approach also involves some overhead, caused by attaching procedures to attributes.

2.1.9. *The Graph-Based Approach*

When a change is made to an object, [Chin-Purcell et al. 1989] implement a graph-based approach to propagate the change to all its dependent objects. For this purpose, objects are organized in structured networks. Some objects are directly represented in a view, while others represent more abstract concepts and serve to link the visible objects so that they represent a coherent design. A simple way to propagate a change in the system is to have each object in the network maintain a list of its dependent objects. Whenever the object changes, every object in its dependents list is notified, whereupon, those objects would update and notify their dependents, and so on. To avoid multiple updates on the same object and the associated performance loss, some graph analysis must be carried out before the update sequence begins. This is performed in two steps: the future phase, where the graph of objects is marked to see where the change will propagate, and the resolution phase, where the objects are told to update in the correct order.

[Gyssens et al. 1994] use graphs as the basic data type in their GOOD database. Objects are represented as nodes in a graph. The nodes in the graph represent object classes, and edges represent the relationships and properties of objects. The objects in the database can be manipulated by adding or deleting nodes and edges in the graph. The possible operations include addition and deletion of edges, deletion of nodes, and grouping of objects.

The approach in both [Chin-Purcel 1989] and [Gyssens et al. 1994] does not address the inheritance issue at all. It handles neither changes to object-class memberships nor object behavior. Working with graphs requires some additional knowledge and certain optimization techniques in order to minimize processing time by processing objects in the correct and optimal order.

2.1.10. *Meta-Objects Approach*

In their reflective-architecture based operating system called Muse, [Yasuhiko et al. 1989] implement the approach of meta-objects, where the system consists of objects and meta-objects. In a reflective architecture, meta-objects can be viewed as a definition of the computation of objects, or a virtual machine. An object is interpreted by its meta-object. Yet, an object can also influence the computation of its meta-object by sending messages to it. This relationship between an object and its meta-object is called the meta-object hierarchy. Objects are also grouped in classes, and this class hierarchy is independent from the meta-object hierarchy. An object can inherit from multiple classes, and is also free to migrate to a new meta-object, as long as it is compatible with its original meta-object. The approach is based on the assumption that a meta-object holds all the information required to compute its objects after a change. This assumption indicates that all the possible changes must be predictable. All changes to the meta-object hierarchy and the class hierarchy are implemented at compile-time only. The work does not handle any changes to object behavior. Furthermore, the researchers do not explain how changes to an object or class are carried out.

[Tan et al. 1990] introduced the concept of persistent meta-objects which can be used for general reflective computation of objects. Each object has a meta-object, and each program or message has a meta-program and meta-message respectively. In [Tan et al. 1990], a type is defined as a set of objects with identical structures and behaviors. This is equivalent to the definition of a class in most of the literature, and different from a type as defined by [Fishman et al. 1987]. When a type is changed, all the instances of the type must be notified, in order to carry out the update. In this work, changes are propagated in a *lazy* manner. The change is recorded in the meta-object for the class, and at a later time, when the instance is accessed, the change is implemented. Changes are propagated through a chain of meta-objects of classes and subclasses. Whenever object behavior, called programs in the article, is changed, then its meta-program sends a message to the meta-object of all classes used in that program to ensure that all of the classes are up-to-date. If there is any difference between the classes and their instances, then all the deferred changes are immediately performed. This article does not indicate, explicitly, whether this modification to a program is accomplished at compile-time or at run-time. The approach does not handle major changes such as changing class structure, inheritance relationships, or object membership to a class.

The approach in both [Yasuhiko et al. 1989] and [Tan et al 1990] may require some overhead to maintain all the meta-objects, meta-messages, meta-programs, and maintain consistency, and at the same time exchange messages between the metas and their objects.

2.1.11. *Relevant-Classes Approach*

The relevant classes concept was introduced in [Nguyen et al 1989] and implemented in their Sherpa data model. The approach is used to represent partial and meaningful designs which are characterized as potential steps toward a complete class definition. Every instance is attached to exactly one relevant class. Relevant classes can neither be related by subclass/superclass nor generalization/specialization relationships. They are only used to support incremental changes and correspond to meaningful combinations of class attributes and constraints. This approach is specific to incremental and predictable changes, and does not handle major changes. Also, it does not handle changes to object behavior and object migration.

In [Nguyen et al 1987] the authors introduced, in their Cadb data model, the concept of a *prototype*, which is a specific "element" maintained for each class. A prototype is produced automatically with the first object instance of the class and maintained dynamically to reflect updates performed on the objects belonging to the class. Prototypes form a database abstraction to model dynamic data structures and values. All the changes on a class are first applied to the class prototype, to allow detection of potential inconsistencies resulting from the change operations and any side-effects resulting from automatic computations of the operation. This approach relies on a dynamic characterization of the object's equivalence classes. The idea is, considering the initial schema of an object, to automatically produce all the potential structures of the sub-objects that it may contain. The only legal structures are those produced this way and no errors in the object structures are permitted. However, prototypes are allowed to have errors in their values in order to test different attribute domains. This approach is specific to CAD where changes are parameter-driven and the types of changes are mostly limited to changing attribute values.

[Lee et al. 1993] proposed a prototyping model called Object Prototyping which assumes that a database schema is fixed, and prototyping is achieved by applying alternative values to the attributes of an object, rather than by changing attribute or method definitions. When the user selects an object to be changed, he/she constructs a prototype with the desired properties, and functionally validates the performance of the prototype objects using a simulation mechanism. If the results satisfy the user, then the prototyping session stops. If not, another prototype is created with a different set of properties.

This approach is also limited to CAD and involves some prototyping and simulation overhead. Also, it does not handle changes such as addition and deletion of classes, and change in their relationships.

2.1.12. *Collection Keepers Approach*

[Riet 1989], in the Mokum object-oriented knowledge base, suggests that an object can be an instance of many types simultaneously, and that objects can be also grouped in collections. Collection keepers are objects specially created to control the activities of a collection. Adding or deleting objects to and from a collection, is only done by sending messages to these collection keepers. There is no explicit relationship between collections. This approach handles only creation and deletion of objects. It does not have the capability of handling major changes to class structure, relationships, or object behavior.

2.1.13. *The Indexing Approach*

[Al-Haji et al. 1992] implement the approach of using indexing tables to perform schema changes In their ODS database model, [Al-Haji et al. 1992] considered an object as consisting of two parts: (a) private memory part where the values of attributes are stored; (b) public part where the methods used to manipulate the private part are found. The private memory of an object is a contiguous series of words called a *chunk* (This definition of a chunk is different from the one adopted later on in this research). The approach utilizes five tables: IT, the inheritance table which relates the class instances to their immediate supers, NT the nesting table to relate the immediate nested objects, DOT, the disk object table to relate an object to its class, SOT the sub-objects table which maintains the sub-objects for each object, and ROT, the referencing objects table to indicate which objects reference other objects. Information in an object is distributed between the chunk, the IT, and the NT. Each of these requires some memory allocation called a *segment*. Adding an object instance to a class requires adjusting the IT, NT, ROT, and SOT, to reflect the relationship between the new instance and other instances existing in the database. The DOT is also adjusted also by adding one entry for each new instance. The deletion of an instance variable depends on whether it has an atomic value or a collection of values. If not atomic, the value can be deleted inside the NT by deleting the nodes that represent the value of the instance variable from the record of the chunk. The ROT is also adjusted to reflect the change. The deletion of an atomic valued instance variable can be done directly within the related segment.

This approach is capable of handling neither changes in object membership to a class nor inheritance relationships. The approach might also require a considerable amount of overhead for keeping all the five tables current and updated.

2.1.14. *The Object-Specialization Approach*

[Sciore 1989] introduced the concept of specialized objects, where each object is assigned a class and acquires the variables and methods of that class. However, inheritance is determined individually by each object. Object specialization means that objects are able to inherit from other objects. In general, a real-world entity is modeled by several objects. The objects corresponding to each entity are arranged in an *object hierarchy*. These objects are partially ordered to describe how the objects in the hierarchy should be viewed. Each object in the hierarchy denotes a *role* played by the entity. In this approach, there are two inheritance hierarchies, the class and the object hierarchy. There are also two special methods defined for all objects. *Super* which returns the object's parent in the hierarchy, and *schema* which returns the corresponding data dictionary for that object's class. There are two ways to add objects to a hierarchy. Existing objects can be copied from other hierarchies, or an object can be created for a completely new class. New objects can be created by cloning existing ones, and roles may be added by extending an existing object to include the features of another one.

This approach allows only adding and deleting of objects, and does not handle other major changes such as changes to class structure or inheritance relationships, changes to object membership to a class, or changes to object behavior.

2.1.15. *The Views Approach*

[Wiederhold 1991] defined a view on a database as consisting of a query which defines a suitably limited amount of data. Views are created to provide multiple abstractions of the objects in a database. Views provide flexibility to the extent that different users can view the same database according to their needs without affecting the underlying data. Views can be generated from the database itself or from other views as well. Objects which are defined at the time when the database is created are called the *base objects*, while those created as a result of query are called *view objects*. [Wiederhold 1991] proposes an architecture to generate views in object-oriented databases. The architecture consists of: (a) a set of view-object generators to extract data out of the base objects in relational form, and then assemble the data into a set of view objects, (b) a set of view-update generators to propagate changes made to view objects to the base objects.

Although this approach provides some flexibility, it has some weaknesses such as: (a) view generation requires a special mechanism such as SQL commands whose complexity depends on access paths of the underlying data and the complexity of view definition [Konomi et al. 1993], (b) view changes are not automatically propagated to other views and base objects, but it rather require additional programming tasks to be carried out, and (c) view objects are not persistent and require re-computation each time they are created, and must be explicitly written to files to be persistent.

Materialization of views, which is defining object views as real objects and not virtual objects, reduces the retrieval cost. However, when updates occur in base objects, some additional expensive update overhead is required to recompute views in order to keep the materialized views consistent with the base objects.. To overcome this problem, [Konomi et al. 1993] propose data structures that permit efficient incremental updates of materialized views. When classes are derived from a path of classes, in a class schema, the key of the path, called a super-key, is defined to be utilized for updates. If derived classes and the keys of the paths from which the classes are derived are contained in the same subgraphs of class schemas, then class schemas will satisfy a condition called super-key condition. If class schemas satisfy the super-key condition, incremental updates are performed efficiently by detecting multiple paths to underlying objects utilizing objects of classes in the keys. This approach involves a number of high cost procedures such as views materialization and incremental updates. The incremental update procedure, according to [Konomi et al. 1993], adds extra classes and introduces more redundancies into base objects. [Liu et al. 1993] suggest a mechanism, called change processing, to associate the same object across the different views so that changes made to the object in one view can be propagated to the other views. This approach is limited to composite objects, where the domain of a class is another class.

A view, according to [Rundensteiner 1993] as implemented in MultiView data model, is defined by an object-oriented query and is computed upon demand rather than stored. She introduces the concept in MultiView as the mechanism for the generation of customized views. These views assure that updates through views are consistently reflected in the underlying database. MultiView breaks up view specification into two tasks: (a) the derivation of virtual classes via an object-oriented query and their integration into one consistent global schema and (b) the definition of view schemata composed of both base and virtual classes on top of this augmented global schema. This approach provides a good level of flexibility since each user can customize his/her own views. However, the MultiView approach is specific to CAD and requires two additional expensive overhead mechanisms. The first mechanism is the creation of a schema, called global schema, to contain all the derived classes. The other mechanism is the specification of an arbitrarily complex view schema composed of both base schema and global schema.

Furthermore, [Rundensteiner 1993] points out that an additional mechanism is needed since relationships in object-oriented views must be validated so that they are consistent with the global schemas.

In the SDM model described in [Rovira et al. 1993] views are achieved through aggregation of lower level components to define a new type of component. These are clusters which represent groupings of objects that share common properties. The cluster is the basis for the support of views. Views are achieved by subclassing so that changes will be reflected automatically, assuring integrity and consistency among multiple views. This approach is not capable of handling view updates.

2.2. Literature Review Summary

Table 2.1 below shows a summary of the features and capabilities of the fifteen approaches discussed above, as well as the issues they address, plus some general remarks. The first column lists all fifteen categories of our classification scheme. The next three columns indicate whether the corresponding approach handles changes, during run-time to: (a) class structure, (b) class behavior, or (c) object-class membership. The column titled dynamic inheritance shows whether or not the approach handles changes to inheritance relationships at run-time.

The next column indicates whether the approach handles chunk management. Here a *Chunk* is defined as a logical grouping of objects from various classes[Mejabi 1994], [Lim 1996]. Each chunk has a set of parameters used to define the relationships between its objects. Chunks have a significant importance in manufacturing in order to represent instances such as, the Bill-of-Material (BOM) of a target finished product. Other examples of the use of chunks include modeling the assembly processes and machine/labor assignment to cells in cellular manufacturing. Finally, the last column contains some general comments.

It should be noticed that most of the approaches are directed toward special application domains such as CAD, OIS, or database schema evolution and none specifically addresses the special needs of manufacturing planning. In addition, no approach is capable of comprehensively handling all the various changes that are required. A clear need therefore exists for a method that is capable of handling a comprehensive range of changes, such as changes to class structure or relationships, object to class relationships, and object behavior, as well as chunk configuration at run-time.

A clear need exists for the development of a flexible architecture which addresses the needs of manufacturing as well as handling all the maintenance tasks at run-time with minimal effort.

Approach	Class Structure	Class Behavior	Object Migration	Dynamic Inheritance	Chunk Management	Remarks
Software Reuse	No	No	No	No	No	• Coarse level of granularity, and requires developing a glue code
CLOS	Yes	Yes	Yes	Yes	No	• Generic O.O. programming language which does not specifically address MRP needs
CASE	No	No	No	No	No	• CASE is capable of propagating changes only to the logical design BUT not to the physical underlying persistent data
Template-Parametric-Modeling	No	No	No	No	Yes	• Specific to simulation software
Conversion	Yes	No	Yes	No	No	• Requires attribute domain constraint • Involves name conflict
Versioning	Yes	No	No	No	No	• Specific for certain applications like OIS
Machine Learning	Yes	No	No	No	No	• Changes have to be predictable • Trade off between flexibility and efficiency.
Functionally-Defined Data	Yes	No	No	Yes	No	• Changes have to be incremental • Attributes have to be related
Graph-Based	Yes	No	No	No	No	• Requires knowledge in optimization techniques
Meta-Object	Yes	No	No	No	No	• Involves overhead mechanism to keep and maintain all the meta-objects in the system
Relevant Classes	Yes	No	No	No	No	• Specific to CAD. Change have to be incremental • Requires overhead mechanism for simulation
Collection Keepers	Yes	No	No	No	No	
Indexing	Yes	Yes	No	No	No	• Involves the overhead of keeping five tables current all the times
Object Specialization	Yes	No	No	No	No	
Views	Yes	No	Yes	No	No	

Table 2.1 A Summary of approaches to software flexibility

2.3. Research Problem Statement

The problem that this research addresses is *the high cost of manufacturing software maintenance*. The objective is to:

- Reduce the effort, in terms of human and machine resources, as well as time, associated with maintenance.
- Perform all the maintenance tasks at run-time and propagate the changes to the underlying data.

This will eventually lower the high cost required to maintain manufacturing software systems. For many manufacturing domains, it is highly desirable to maintain software, not just at compile-time, but rather at run-time while the system is running since "shutting the system off" is often not an acceptable option.

2.4. Research Approach

The approach of this research to solve the problem is to develop an architecture that can be used to develop and maintain highly flexible manufacturing software, especially MRP systems. The research will provide the capability to maintain the systems at run-time and permit propagation of changes to the underlying persistent data. The proposed architecture will utilize the concept of software reuse and be comparable to CASE tools, but will not follow the code generation philosophy used in most CASE systems. Though, it can be best described as a dynamic-object application builder tool built on the top of a run-time system; it will also incorporate the concept of chunking from the Template-Parametric modeling approach. The proposed architecture might be applicable to any software, however the focus will be on manufacturing software, especially MRP.

The uniqueness of this research will be demonstrated by:

1. Ease of developing manufacturing software systems, especially MRP by utilizing a specialized-classes library.
2. The capability to handle major and minor changes to software at ***run-time*** with minimal time, effort, cost, and also propagate the changes to the persistent underlying data.
3. Applicability to a wide range of manufacturing software domains since the system utilizes an expandable library of pre-compiled objects.

2.5. Deliverables

This research expects as its final result to produce the following:

1. A fully documented architecture.

2. A demonstration of the architecture's flexibility.

3. A working prototype implementation of the main elements of the architecture.

4. A manufacturing planning illustrative example to explain the use of the architecture.

Chapter 3

APPLICATION OF THE PROPOSED ARCHITECTURE, AN ILLUSTRATIVE EXAMPLE

Section 3.1 of this chapter presents an overview of the operating scenarios of the proposed architecture. Section 3.2 explains the procedure and platform adopted in order to implement a working prototype of the architecture. Section 3.3 and 3.4 present an Inventory control component of an MRP system designed for specific manufacturing domain as an illustrative example. Section 3.5 explains the functional and structural changes required to implement the same software in a different application domain and what this process might involve, the section also presents how to utilize the architecture to carry out these changes. Section 3.6 contrasts the architecture maintenance approach to the traditional software development tools.

3.1. Operating Scenarios of the Architecture, An Overview

The proposed architecture is a menu-driven system which utilizes a number of graphical user-interface tools such as screens, buttons, and radio buttons. As shown in Figure 3.1, the system starts with a main menu allowing the developer to select any of the following four options: Design, Implementation, Data population and Maintenance.

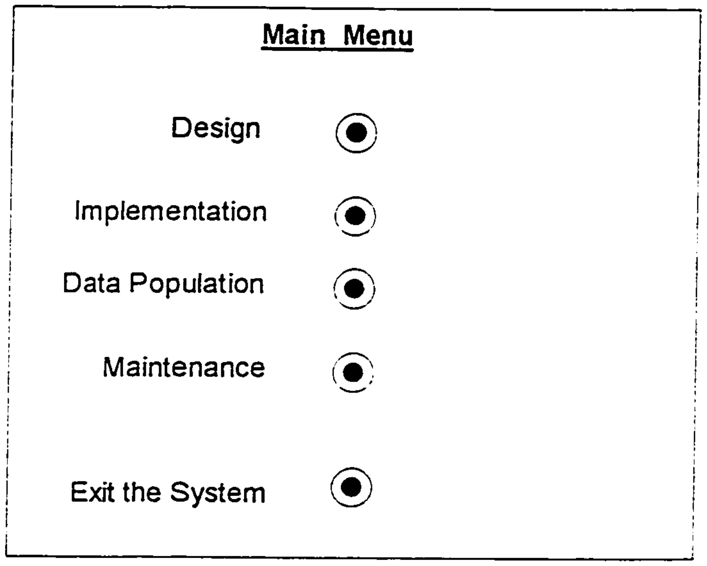


Figure 3.1, The Main Menu of the proposed Architecture

Upon selecting the maintenance option, for example, the system will display another user-interface with a number of maintenance options such as: adding or deleting a class, adding or deleting an attribute, adding, deleting or modifying a method. Figure 3.2, shows a sample user-interface to add an attribute to a class. In this example, the system displays all the classes defined in the MRP system under the title: (These are Your Classes) where the system can highlight to select any class. The system will also show a number of icons representing the data types where the system can choose from, and finally the developer will be prompted to provide a name, domain, and default value for the attribute. The system will then append the attribute to the selected class upon clicking the save button on the screen.

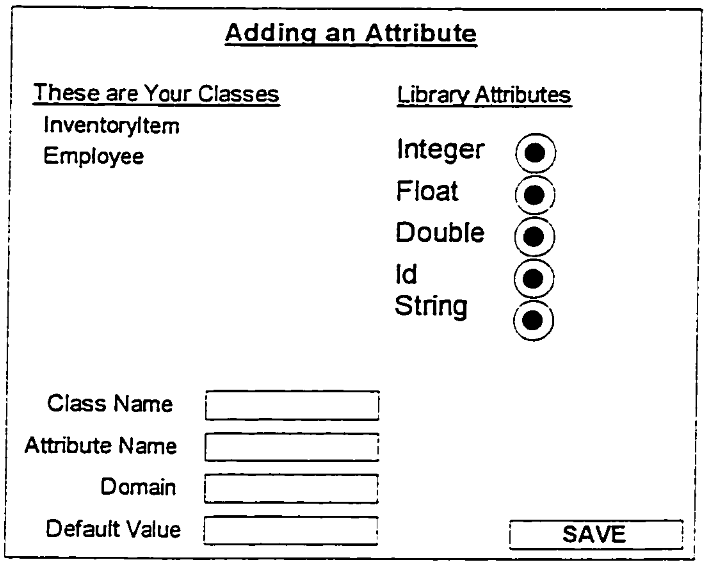


Figure 3.2, The "Adding an Attribute" user-interface

3.2. Implementation Procedure & Platform

In order to prove the concept of this research a working prototype containing a subset of the overall functions of the architecture have been implemented. The target prototype is sufficient to conduct a realistic maintenance task of an Inventory Control system at run-time and highlight the ease of maintenance.

For the overall implementation platform such as the programming language and tools, three options were considered, CLOS, visual C++, and Objective C. The following decision variables have been used in order to make a selection.

1. Flexibility issues of the language; such as dynamic binding versus static binding, static type checking versus dynamic type checking. These issues were very important in the decision making process since flexibility is the major characteristic of the proposed architecture.
2. Richness of the language class library; a rich class library such as Arrays and Dictionaries which provides important basic programming structures and functions.

3. Availability of user-interface and development building tools such as screens, buttons, and toolbars is considered to be a valuable option in the selection process.

Table 3.1, below, uses the four decision variables to compare the advantages and disadvantages of each selection.

	CLOS	**Visual C++**	**Objective C**
Flexibility: *Binding*	Low since dynamic binding is not available	High, however utilizing dynamic binding is not a straightforward	Very high due to the availability as well as the ease of use of dynamic binding
Flexibility: *Type Checking*	High since all variables are not type checked at compile-time.	Low because C++ is a static type-checking language	Very high because Objective C is a hybrid type-checking language and decision to choose is left to the developer
Class Library Richness	Low	Rich class library	Rich class library in addition to the availability of some re-usable source code from existing research
User-interface and Development tools	Poor	High	High

Table 3.1, Platforms Comparison

The conclusion is that Objective C is the best selection as an implementation language. Accordingly, the platform would be the OPENSTEP operating system since Objective C uses OPENSTEP as its primary environment.

3.3. <u>An Overview of the Inventory Control Subsystem</u>

A Inventory Control System within an (MRP) package for auto parts manufacturing plant ensures that all materials are available when and where needed, in the correct quantity and at the right time. In order to obtain the required raw materials or sub-assemblies for a particular shift, the gross production plan must be processed to obtain a list of items required for each final product. The Inventory Control System must then locate these items and take them out of inventory so that production can start. The system also generates purchase orders for those items whose quantities are below certain safety levels through proper timing of order placement . Figure 3.3 shows a schematic view of the Inventory Control System.

The Inventory Control System will take, as its <u>input</u>:

1. The gross production plan for a shift which consists of: (a) the Id for each finished product, and (b) the desired quantity.

and produces the following <u>outputs</u>:

1. A list of items and their corresponding quantities required for the shift production plan.
2. The physical location of each item.
3. Purchase orders for those items whose quantities reach below their corresponding safety levels.

The system can be functionally decomposed into the following five main modules:

1. *processProductionPlan,* this module accepts the gross production plan and processes one finished product at a time. It identifies the items needed, and their corresponding quantities, required to produce a target finished product (using its BOM database) and passes the information to the next module, *checkInventory*.
2. *checkInventory* which searches for each item and checks whether or not the item is available in the right quantity. This is accomplished by checking the quantity-on-hand for that item in the Inventory database. If available, then the *pullItem* module will be executed. Otherwise, the *printPurchaseOrder* module would be executed.

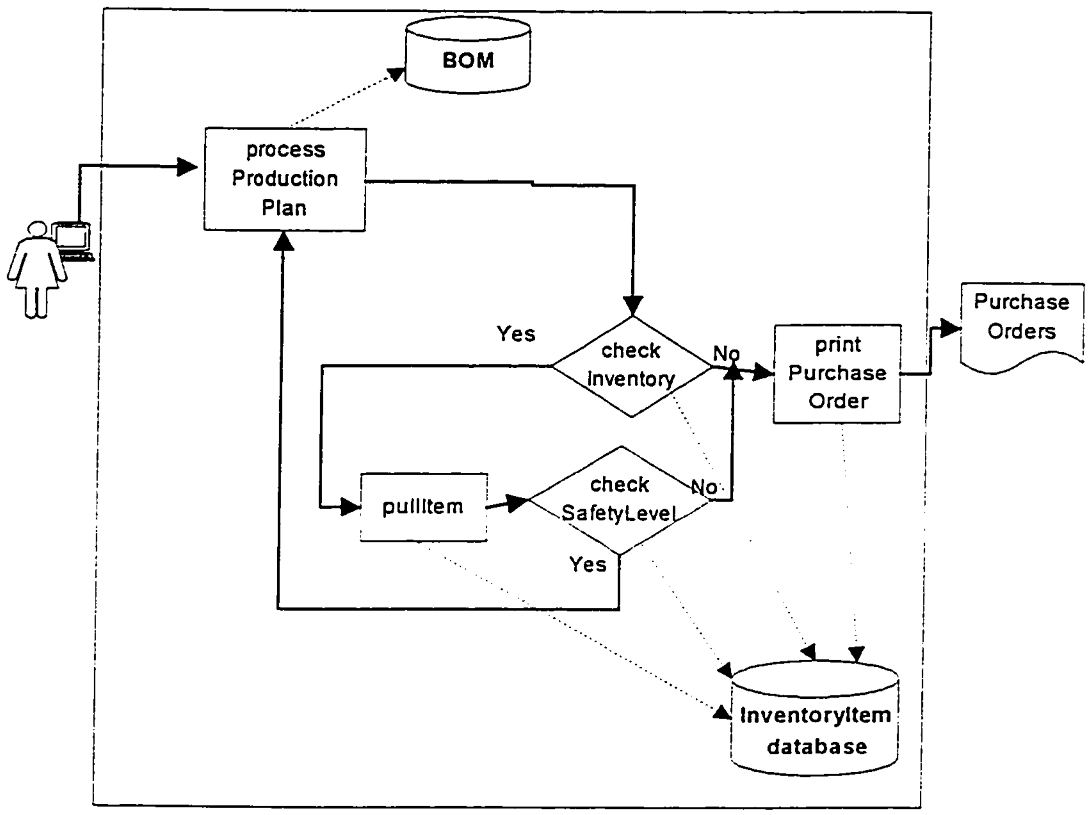


Figure 3.3 A schematic view of the original Inventory Control System

3. *pullItem* updates the quantity-on-hand in the Inventory database and prints shipping instructions, then passes control to the *checkSafetyLevel* module.

4. *checkSafetyLevel* module checks whether or not the quantity-on-hand of the target item is below its safety level. If it is below its safety level, the *printPurchaseOrder* module will be triggered.

5. *printPurchaseOrder* prints a purchase order for procurement of the target item based on its order-quantity.

3.4. Classes of the Inventory Control System

Based on the functional decomposition of the Inventory Control system, the candidate classes, their attributes and methods are listed in Table 3.2, below:

	RequestMgr	*InventoryItem*
Attributes	productNo *(string)*	itemId *(string)*
	productQuantity *(integer)*	itemDescription *(string)*
		quantityOnHand *(integer)*
		standardCost *(float)*
		safetyLevel *(integer)*
		vendorId *(string)*
		leadTime *(integer)*
		location *(string)*

Methods	*processProductionPlan*	*checkInventory*
		pullItem
		checkSafetyLevel
		printPurchaseOrder

Table 3.2 classes of the original Inventory Control System

3.5. Maintenance Plan of the Inventory Control System

In this illustrative example, the Inventory Control system just described will be maintained to make it applicable for use in a different environment, such as a hospital or nuclear reactor, where high level of security is required. In addition, the new requirements indicate a need for managing special procedures and policies for handling, shipping and disposal of these materials. Figure 3.4 a schematic view of the Inventory Control system after modifications.

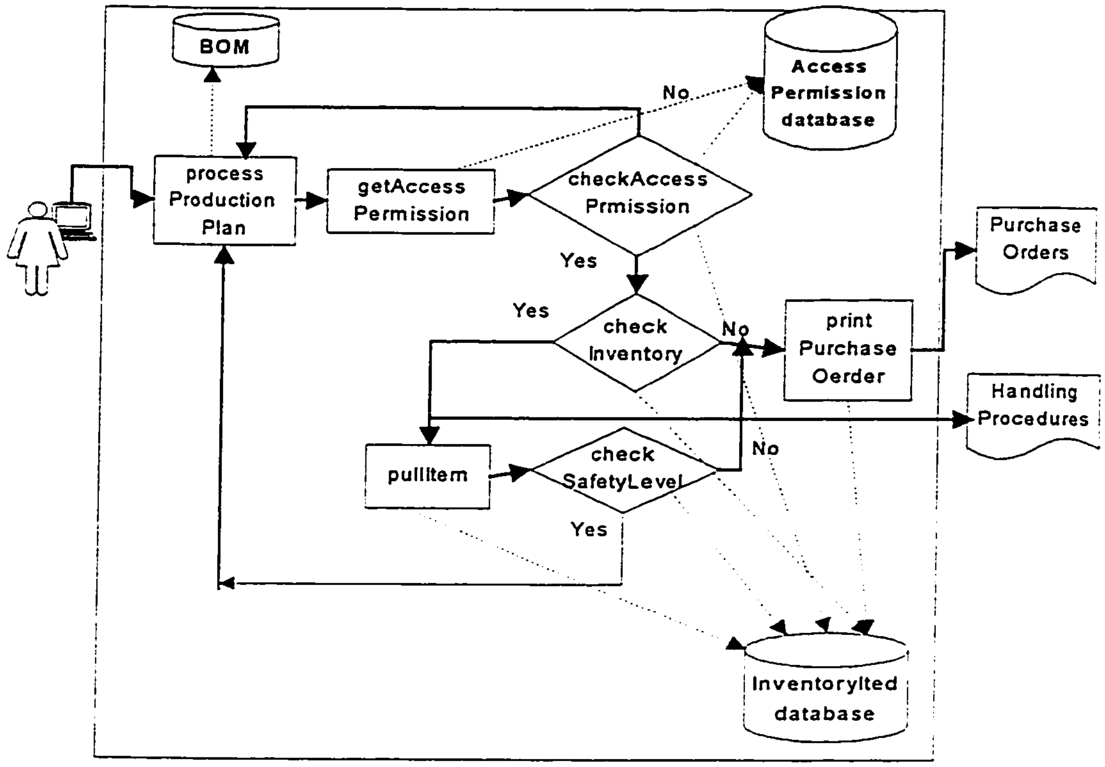


Figure 3.4 A schematic view of the Inventory Control System after modifications

With these new requirements, some re-analysis will be required to produce a different functional decomposition of the system. The output of the re-analysis highlights the following:

1. An "Access Permission" database has to be created to contain an identification code for each personnel in the plant along with his/her corresponding access permission for requesting various items. Table 3.3 shows that request-id "Smith" , for example, has a privilege of type "2" while "Johnson" has a privilege of type "6". In this setup type "9" may have higher access privileges than "7" or "5".

requester-id	requester-access-permission
Johnson	6
Whitener	9
Smith	2

Table 3.3, a sample requester's access Permission database

2. A *getAccessPermission* module should be developed to receive the requester's identification code, use it as a look-up key to identify the requester's access permission from the Access Permission database, and append the requester's access permission to each item before it is passed to the next module, *checkAccessPermission.*
3. A *checkAccessPermission* module should be developed to perform the task of determining whether a requester is authorized to request a certain item by matching the item access code of the Inventory database with the requester's access permission before proceeding to the next module *checkInventory*.
4. The *processProductionPlan* module should be modified so that it invokes the *getAccessPermission* and *checkAccessPermission* instead of just the *checkInventory* module. The *processProductionPlan* module should also be modified so that it prompts the requester to provide his/her identification code.
5. In order for the *checkAccessPermission* module to function properly, the Inventory database has to modified so that each item will be assigned an item access code to limit access to it.
6. To make the system capable of addressing the needs to special handling, shipping, and disposing of the various items due to their chemical or physical hazardous nature to ensure safety a new field, handling procedure, has to added to each item in the Inventory database has
7. The *pullItem* module has also to be modified in order to print the special handling procedures and policies.

Figure 3.5 shows a process diagram of the Inventory Control System after modifications.

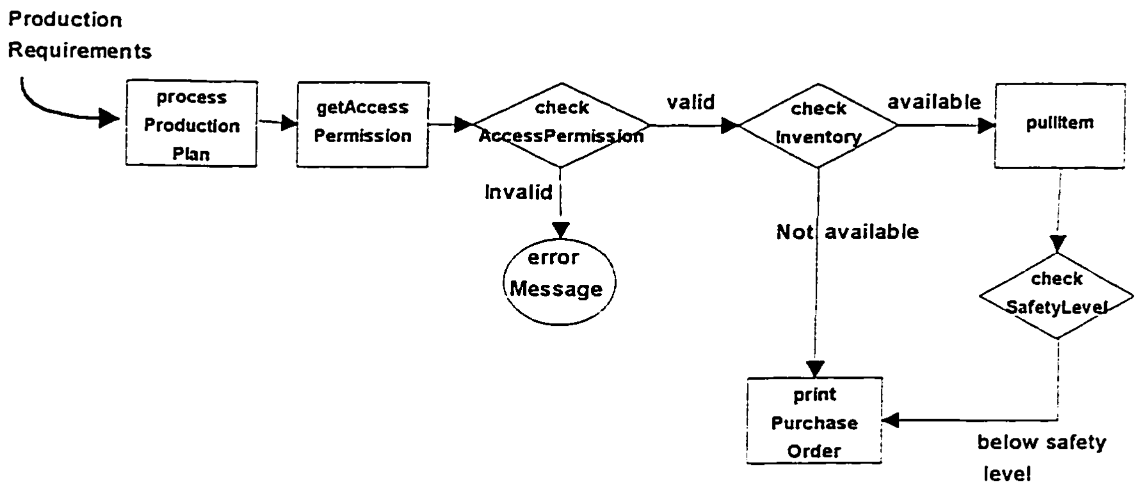


Figure 3.5, A Process diagram of the Inventory Control System after modifications

Subsequently, the re-analysis step will require a re-design step, as shown in Figure 3.6. The following set of changes has to be introduced:

- a new class has to be added.
- the structure and behavior of existing classes have to be modified.
- all these changes have to be propagated to the underlying objects.

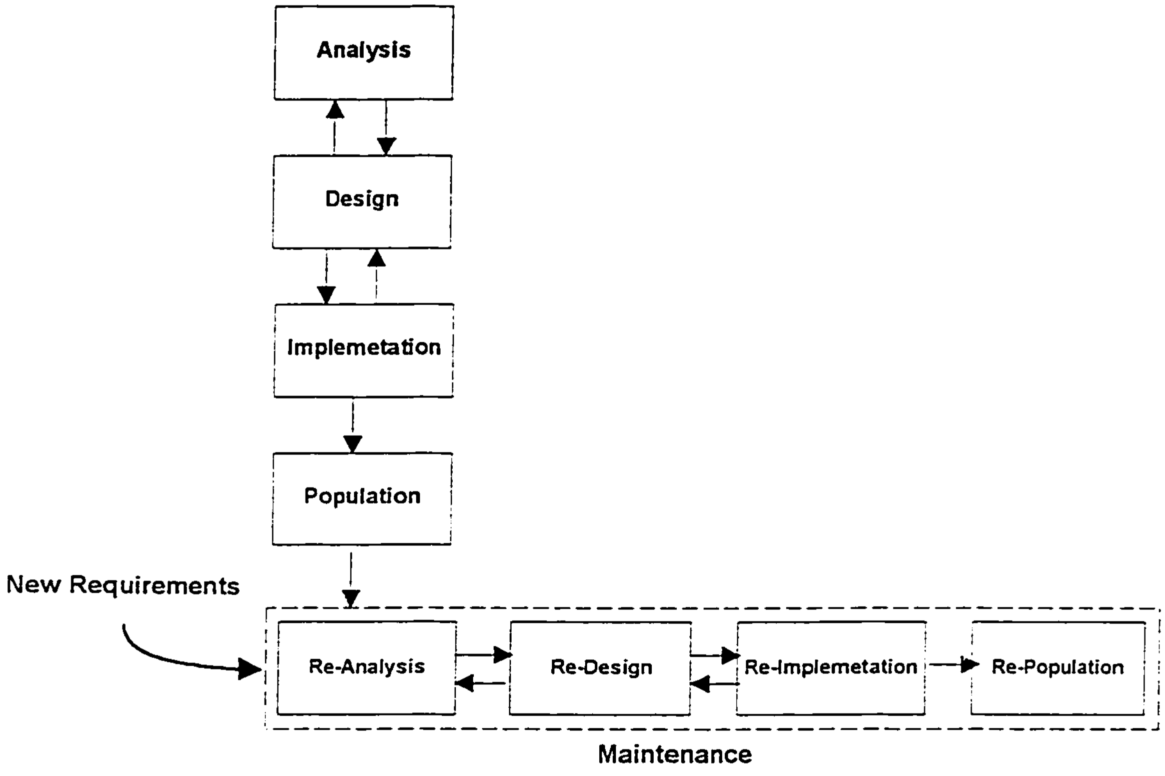


Figure 3.6, the SDLC diagram

The following table summarizes the output of the re-design step:

	RequestMgr	*InventoryItem*	*AccessPermission*
Attributes	productNo *(string)*	itemId *(string)*	**requesterId** *(string)*
	productQuantity *(integer)*	itemDescription *(string)*	**requesterAccessPermis sion** *(string)*
	requesterId *(string)*	quantityOnHand *(integer)*	
		standardCost *(float)*	
		safetyLevel *(integer)*	
		vendorId *(string)*	
		leadTime *(integer)*	
		location *(string)*	

		itemAccessCode *(string)*	
		handlingProcedure *(string)*	

Methods	*processProductionPlan*	*checkInventory*	***getAccessPermisiion***
		pullItem	***checkAccessPermisiion***
		checkSafetyLevel	
		printPurchaseOrder	
		prinHandlingProcedure	

Table 3.4, the modified classes, their attributes and methods

As shown (in bold) in the table above, the required changes are:

1. Creating a new class: *AccessPermission.*
2. Adding two attributes (requesterId & requesterAccessPerm) to class *AccessPermission.*
3. Adding two methods (*getAccessPermisiion & checkAccessPermisiion*) to class *AccessPermission.*
4. Creating objects for class *AccessPermission.*
5. Adding two attributes (itemAccessCode & handlingProcedure) to class *InventoryItem.*
6. Modifying all the objects of class *InventoryItem* to incorporate the two newly added attributes.
7. Adding a method (*prinHandlingProcedure*) to a class *InventoryItem.*
8. Modifying the implementation of a method (*processProductionPlan)* of class *RequestMgr.*

3.5. Maintenance Utilizing the Proposed Architecture vs. the Other Approaches

The maintenance phase is where most approaches will take the results of the re-design step and proceed to re-implementation by going through the traditional cycle of [Edit-Compile-Link]. An additional code would then be required for re-population. In other words, to accomplish all the changes listed in the previous section using any of the approaches discussed earlier, the developer would be required to go, at least once, through the following cycle:

1. Locating where the change must be introduced in the source code.
2. Developing some source code to accomplish the required change as well as for re-population.
3. Compiling (evaluating), linking, the source code.

The re-implementation step using the architecture, on the other hand, will take the same results from the re-design step and will require:

Selecting the "Maintenance option" of the architecture main menu, then:

- selecting the "Creating New Class" option to add the AccessPermission class.
- selecting the "Adding an Attribute to a Class" options to add the two attributes (requetsId & requestAccessPermission) to class AccessPermission, and two attributes (itemAccessCode & handlingProcedure) to class InventoryItem. This option will automatically propagate the changes to the underlying objects in both classes.
- Selecting the "Creating an Object" option to instantiate a number of objects from class AccessPermission.
- Selecting the "Adding a Method to a Class" option to compose the two methods: (getAccessPermission, checkAccessPermission) to class AccessPermission, and (printHandlingProcedure) to class InventoryItem.

All the re-implementation and re-population steps will be carried out automatically and transparently at run-time with out the need to modify an existing source code or develop a new one. Adding an attribute to a class, for example, will be automatically and transparently propagated to all the underlying objects of the target class.

This illustrative example reflects the ease of maintenance of a manufacturing software using the architecture. All the required changes can be performed by merely navigating through the various architecture menus and screens and providing the desired data.

Chapter 4

THE PROPOSED ARCHITECTURE AND THE SDLC

Section 4.1 of this chapter explains the solution methodology for this research problem, and section 4.2 introduces an overview of the proposed architecture. Section 4.3 and section 4.4 explain the different phases of the SDLC of a software system using our proposed architecture. Section 4.5 explains the modification scheme adopted in this research.

4.1. The Proposed Solution Methodology

The solution methodology is built on the concept of software reuse. The proposed architecture, the **Object Assembly System (OAS)**, also incorporate concepts from the Template-Parametric Modeling approach and will be built on top of an object-oriented run-time environment.

The OAS concept rests on two basic ideas:

1. Using a library of pre-compiled objects, and
2. Utilizing pointers to connect the pre-compiled objects in order to assemble classes and chunks.

In order to identify the OAS among the various software systems, Figure 4.1 below shows a generic non-exhaustive software classification scheme. As shown in the figure, the OAS is an application builder which develop an application software by assembling and packaging pre-compiled atomic building blocks.

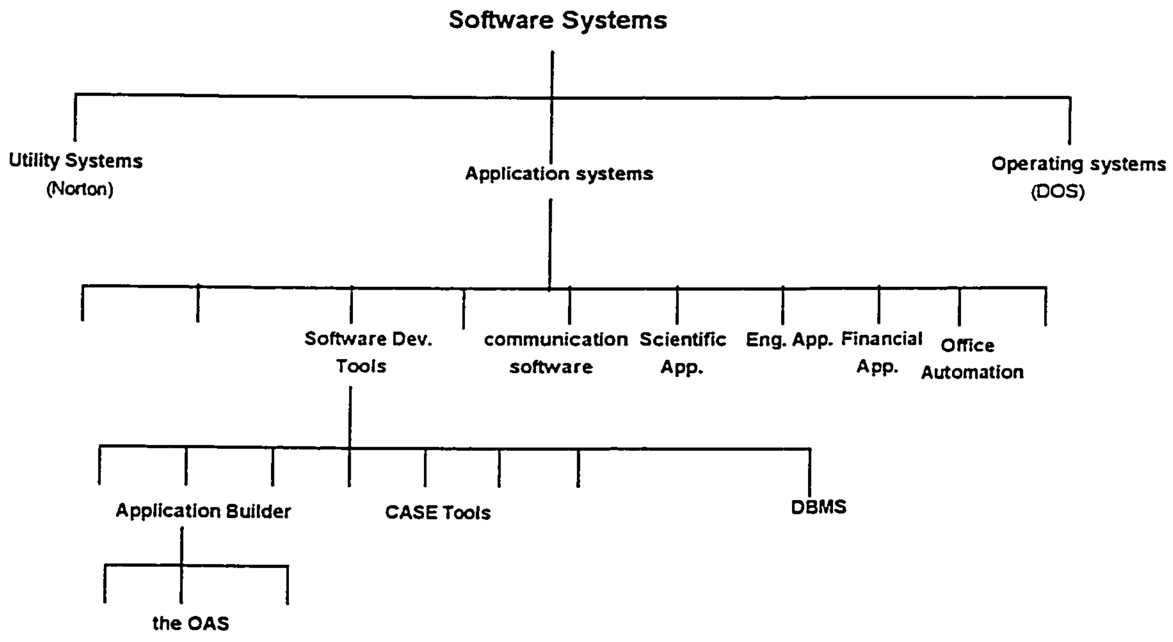


Figure 4.1, A generic Software systems classification scheme

One important contribution proposed by the OAS is the use of chunks. A *Chunk* is a logical grouping of objects, from different classes, having a pre-defined set of relationships [Mejabi 1994] and [Lim 1996]. Each *chunk template* is identified by a unique name and has a set of parameters, which consists of: the number of objects from each class, the names of their corresponding classes, and the relationship between classes. Chunking is utilized as a leverage to perform a wide range of software maintenance tasks with minimal effort. This approach provides the flexibility to automatically manipulate all the objects of the chunk instead of manipulating each object individually. The Bill-of-Material (BOM) module of MRP is a good example to demonstrate the power of chunking in manufacturing software. As shown in Figure 4.2, below, a final finished product "Table" could be thought of as a chunk. The finished "Table" chunk consists of one top and one leg assembly. The leg assembly, in turn, consists of four legs, two short rails, and two long rails. The number of all the tops, legs, and rails can be easily modified collectively by manipulating the chunk parameters as opposed to manipulate each item type individually.

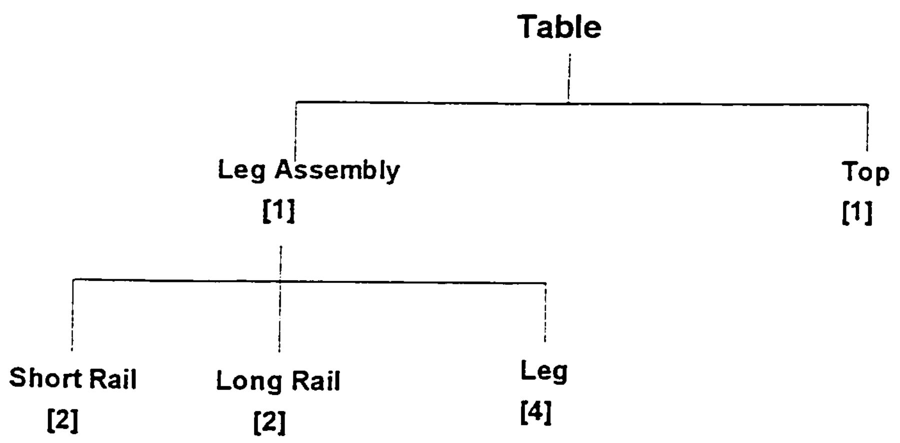


Figure 4.2 A Chunk representing a BOM of a finished product "Table"

4.2. Overview of the OAS Architecture

The architecture of the OAS, shown in Figure 4.3, consists of two main components :

1. a library of pre-compiled objects and

2. a number of functional modules to manipulate the library objects.

The next subsection will describe each component in details.

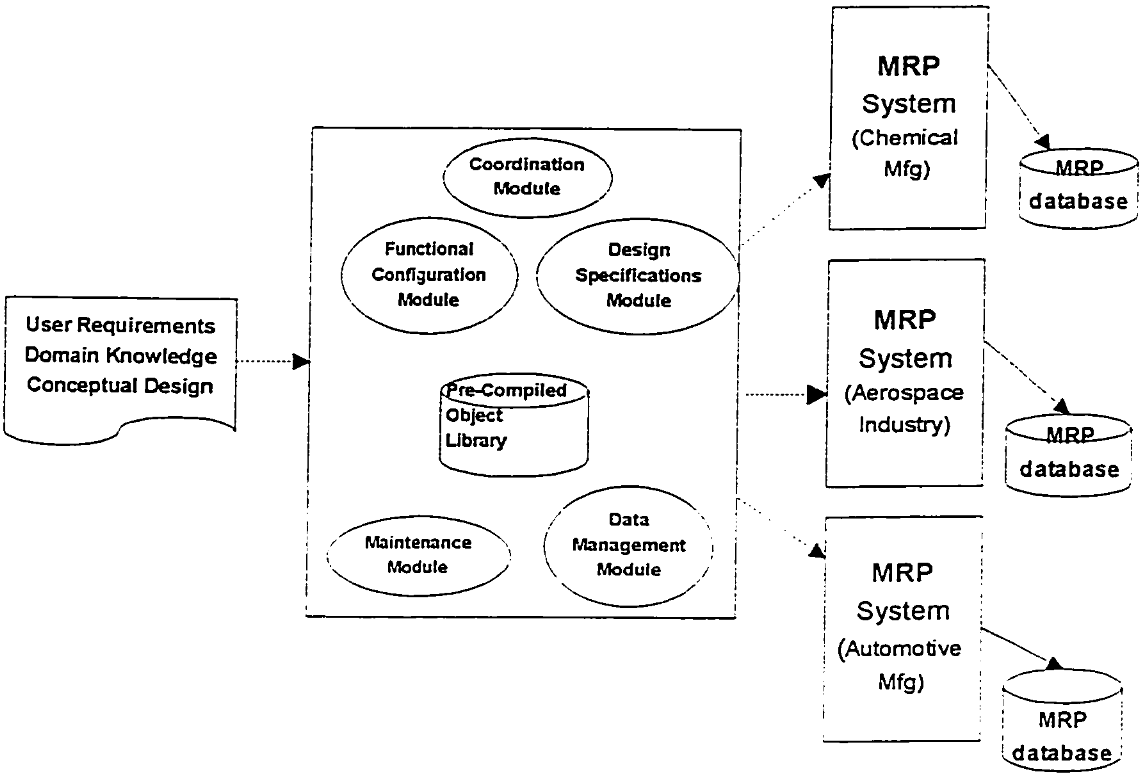

User Requirements
Domain Knowledge
Conceptual Design
Coordination Module
Functional Configuration Module
Design Specifications Module
Pre-Compiled Object Library
Maintenance Module
Data Management Module
MRP System (Chemical Mfg)
MRP database
MRP System (Aerospace Industry)
MRP database
MRP System (Automotive Mfg)
MRP database

Figure 4.3 An overview of the Object Assembly System

1. The *Pre-Compiled Object Library* which is an expandable library of primitive, as well as some specialized, pre-compiled classes and objects directed towards MRP applications. The *Pre-Compiled Object Library* (The OAS Application Kit) contains:

   a) Classes

      1. *Stub*: A class **Stub** (which is the root class in the proposed OAS) used to create all other classes. The definition of class ***Stub***, shown in Figure 4.4 below, proposed in this research will have the following structure:

         1. *Class-name:* which is a unique class identifier.
         2. *Superclass-id:* a pointer to the superclass of the target class.
         3. *Superclass-name:* the name of the superclass of the target class.
         4. *Attribute-list:* An open-ended list of pointers to other objects representing the

various class attributes, including relationship attributes.

5. *Method-list:* An open-ended list of pointers to other objects representing class behaviors.

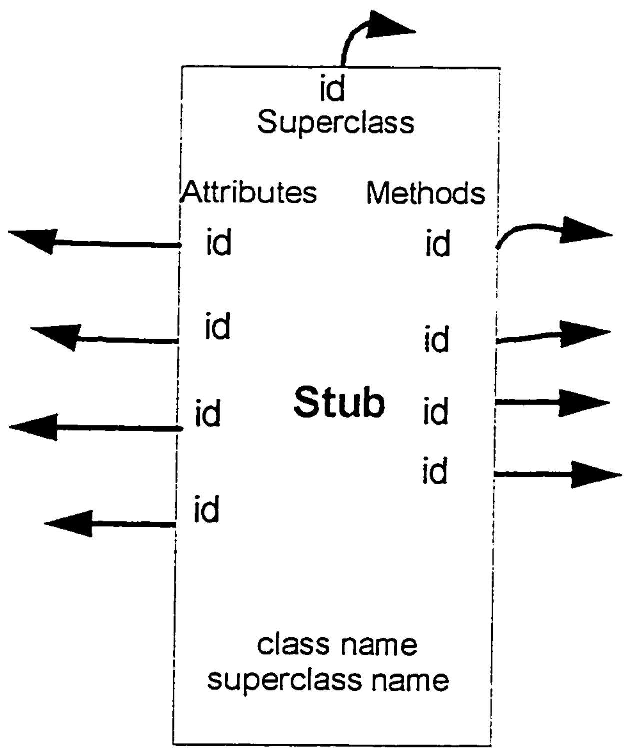


Figure 4.4 Class ***Stub***

2. Specialized Classes: A number of extendible classes designed specifically for MRP requirements. Below is a list of some specialized classes in the library:

   2.1. *InventoryItem,* A class containing information about each item in the MRP system such as: item Id, description, quantity on hand, price, location, lead time, and safety level. This class has the following three subclasses which are used to distinguish the type of item. Class SubAssemblyItem, for example, indicates a sub-assembly part and will have attributes such as componentList .Class FinishedProductItem, on the other hand, is a final finished product and will have attributes such as: toolsAndMachines required, and suggested profit margin.

2.1.1. *RawMaterialItem,*

2.2.2. *SubAssemblyItem*

2.2.3. *FinishedProductItem*

2.2. *Supplier,* A class containing information about suppliers such: supplier Id, address, phone, contact, lead time, discount, and rating.

2.3. *RequestMgr,* A class to describe a request using data such as :requester Id. This class is also capable of exploding a FinishedProductItem.

2.4. *PurchasingMgr,* A class used to hold information about purchase orders such as purchase order number, date, item Id's and their quantities, and vendor Id's. It is responsible for priority management of orders, and timely release and print of purchase orders.

2.5. *WarehouseMgr,* A class used to hold information about a warehouse such as: warehouse Id, address, phone, capacity, cycle counting, and special facilities. This class is highly important for MRP application in case of multiple warehouses due to its capability of assigning items to the proper warehouse.

2.6. *SchedulingMgr,* This class describes the Master Production Schedule by identifying the planned production quantities of finished products by date, week or period as well as the required resources. It is capable of modifying the schedule upon any change in demand.

2.7. *GeneralLedger,* This class enables a company to setup and maintain a record of codes containing summarized totals of debits and credits for specified periods. It contains information such as: company code, code for each division, department and customer of the company.

2.8. *ForecastingMgr*, This class utilizes forecasting logic to anticipate customer order demands, it holds information such as sales for previous periods. This class is important since forecasting is the driving force behind Master Production Schedule.

2.9. *InventoryTransactionMgr*, The function of this class is to record and report inventory changes such as item procurements, inventory transfers, and scrap percentage and quantity, it then updates the inventory accordingly.

2.10. *CapacityReqPlanner*, The function of this class is to monitor and report imbalances between the load versus capacity, as well as the capacity for labor/machine/resource load for a work center during a specified period of time. The class stores the following information about each work center in the company: work center Id, labor capacity, machine capacity, and resource capacity.

2.11. *FinancialMgr*, This class is designed to generate (using the information from the General Ledger) balance sheets, profit and loss accounts, and budget versus actual comparisons.

2.12. *SalesAnalysisMgr*, Presents a detailed analysis of sales history and allows sales orders to be to be matched against forecasts and driven directly into the Master Production Schedule. The class stores information such as: itemId, customer Id, salesPerson Id, quantity sold, total sales price, and ship-to-address.

2.13. *BOM*, This function of this class is to explode a finished product, it holds the Bill-of-Material of that product.

3. *MethodCreator*: A class designed to be used as a container to hold the steps of a target method.

b) Chunks:

1. *Chunk*: A logical grouping of objects.

c) Attribute Building Blocks, A number of classes representing fundamental data types designed to define the various class attributes and relationships. The following is a list of some attribute building blocks in the OAS library:

1. *Integer*: A class representing a fundamental data type "integer".
2. *Float*: A class representing a fundamental data type "float".
3. *Double*: A class representing a fundamental data type "double".
4. *Character*: A class representing a fundamental data type "character".
5. *String*: A class representing a data type "string".
6. *Boolean*: A class representing a fundamental data type "Boolean".
7. *Id*: A class represents a data type "pointer".

Each attribute building block will have the following structure:

1. an attribute name.
2. attribute value.
3. attribute default value and
4. attribute domain.

d) Behavior Building Blocks:

1. *OASMethodLib:* A class containing some basic low-level behaviors, such as:

   (a) **multiply**

   (b) **subtract**

   (c) **retreiveThis Attribute**

   (d) **unArchive ThisObject**

   The *OASMethodLib* also contains some specialized complex behaviors directed toward MRP needs such as:

   (a) **explodeThisProduct**

   (b) **updateQtyOnHand**

   (c) **printPurchaseOrder**

   A class method can be composed using both types of building blocks. The *RequestMgr* class, for example, has a method called: "getRequiredItems" which consists of the following two building blocks:

   1. **explodeThisProduct**, and
   2. **multiply**

   The *OASMethodLib* class can be further extended and new behaviors can be added as needed.

The richness and diversity of the *Pre-Compiled Object Library* (will be referred to as: the *Object Library* for the rest of this research) determines the range of applications that can be built from the library. The library can always be expanded by adding new types of objects and behaviors as needed in an off-line mode.

2. The Functional Modules which are used to manipulate the library objects and classes, they are:

a) The *Coordination Module*, invokes the system main menu and allows the developer to navigate and invoke the various functions of the OAS as well as coordinating the execution of the other modules of the system, such as the *Design Specification* module, to accomplish the desired task.

b) The *Design Specifications Module*, provides an interface to the Pre-Compiled Object Library allowing the developer to design classes, their attributes and behaviors as well as designing chunks during the design phase. The *Design Specifications Module* validates design specifications by enforcing rules such that no two attributes, in a class and its superclass, have the same name. Following is a list of the validation rules:

   a. Each class and chunk in the application software has a unique name.
   b. Each attribute, method, and relationship (in a class) has a unique name.
   c. Each attribute value must be within the domain of the attribute as defined in the class definition.
   d. A superclass of a target class cannot be a subclass of any of the subclasses of the target class.

The *Design Specifications Module* creates and uses the following files for its operations:

(1) the Class Specs file, contains data about each class in the application software such as:

(a) *Class-name:* which is a unique class identification.

(b) *Superclass-name:* Name of its superclass

(c) *Method-list:* An open-ended list of the names of methods along with the names of its building blocks as shown below.

[method name-1, Blg Blk-1,Blg Blk-2,..], [.............], [method name-N, ...]

(d) *Attribute-list:* An open-ended list of attribute definitions. Each attribute definition has the following format:

[attribute name-1, its data type, its default value, domain], [......]

(e) *Relationship-list:* An open-ended list of names of relationships in the following format:

[relationship name-1], [.............], [relationship name-N]

(2) the Chunk Specs file, contains the name of each chunk in the MRP system and its parameter-list.

(a) *Chunk name*: which is a unique name to identify each chunk in the application software.

(b) *Parameters-list*: This list contains the number of objects from various classes, the names of their corresponding classes, and their relationships.

(3) the Attribute-Method table file, which stores all the names of attributes and methods in the application software and indicates, in a tabular format, which attribute is implemented by which method. This dependency relationship is required during the maintenance phase to account for the ripple effect of adding or removing an attribute.

	M1	**M2**	**M3**	**M4**	**M5**
A1	1	1	1	1	0
A2	0	0	1	1	1

A3	1	1	1	1	0
A4	1	0	0	1	1
A5	0	1	0	0	1
A6	0	1	0	0	0

Table 4.1, A sample Attribute-Method table file

The sample table 4.1 above indicates that attribute A4 is implemented by methods M1, M4, and M5, it also shows that deleting attribute A6 will only affect method M2.

(4) the Library ClassNames file, to store the names of all the classes of the Pre-Compiled Object Library in order to be displayed later when needed.

c) The *Functional Configuration Module*, maps and assembles objects, and chunks from the Object Library to the logical design in order to generate the various class prototypes and chunks defined during the design phase.

d) The *Data Management Module* is responsible for populating the system with data objects as well as propagating the changes made during maintenance to the underlying objects.

e) The *Maintenance Module* which allows the developer to select one or more maintenance option such as adding or deleting a class. The *Maintenance Module* will then invoke the appropriate module to execute the required change.

Inheritance in the OAS

As has been shown in Figure 4.4 in the beginning of this section, this research implements ***single inheritance***, where a class inherits its behaviors and characteristics from a single superclass via an IS-A relationship as opposed to multiple inheritance, where a class inherits its behavior and characteristics from more than one superclass as shown in Figure 4.5 below. The justification for this decision is based on the following reasons:

1. *Conflict*: When methods with the same name are inherited from A and B, it is not clear from which class the methods should actually be inherited. Such conflicts are inevitable when classes A and B have a common (direct or indirect) superclass C. In this case, an additional mechanism has to be provided to resolve the ambiguity [Blascheck 1994].
2. *Duplication of instance variables*: When A and B have a common superclass C, class N would inherit attributes from both A and B. If objects of class C have an attribute x, objects from class A and B will also have attribute x. When N inherits from A and B, the problem arises whether x should be duplicated in N objects or not. An additional mechanism is needed to allow the developer to make such decision [Blascheck 1994].
3. *Complexity*: In single inheritance, class hierarchy always consists of one or more trees. Trees are easy to understand, because there is only a single line of ancestors for every class. When multiple inheritance is implemented, the resulting class hierarchy takes the form of a directed acylic graph (DAG). The superclasses of an individual class also form a DAG, which makes it hard to determine statically which methods are inherited from which superclass. This would necessarily result in additional complexity to software maintenance [Blascheck 1994].

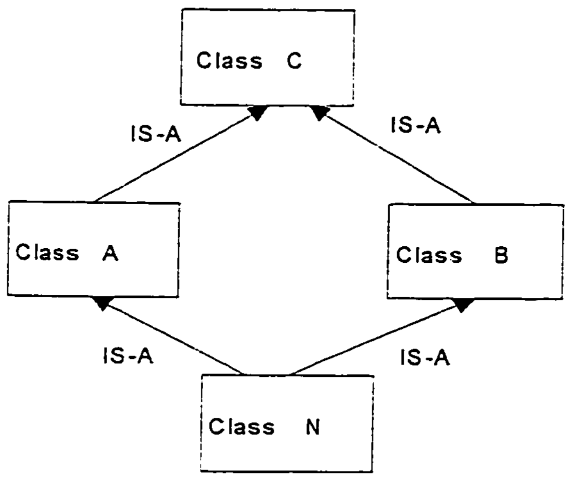


Figure 4.5 Multiple inheritance

4. *Efficiency:* Because of the additional complexity, compilers for languages with multiple inheritance are difficult to implement and consumes more memory and longer run-time [Blascheck 1994].

4.3. Overview of the OAS Approach to the SDLC

Most of the researchers and practitioners in the field of software engineering consider the software development life cycle to consist of: Analysis, Design, Implementation, and Maintenance. *Data Population,* which is the process of instatiating a software system with data so it can be used to fulfill its purpose, is typically not recognized as a separate phase despite its considerable importance. This research. however, will highlight data population since most of the maintenance efforts will relate to propagating changes to the underlying data objects.

Figure 4.6 shows a high level overview of the SDLC using the OAS approach, and Figure 4.7 outlines the SDLC using traditional software development methods. This section presents an overview of the different phases of the SDLC of a software system using the OAS approach. The discussion, in this section as well as the following sections, will be presented for each phase in terms of:

1. The input and output for the phase or activity.
2. The various functions in each phase, its corresponding set of inputs, outputs and tools
3. The proposed OAS architectural elements are explained, where applicable, to provide a high level understanding of this research.

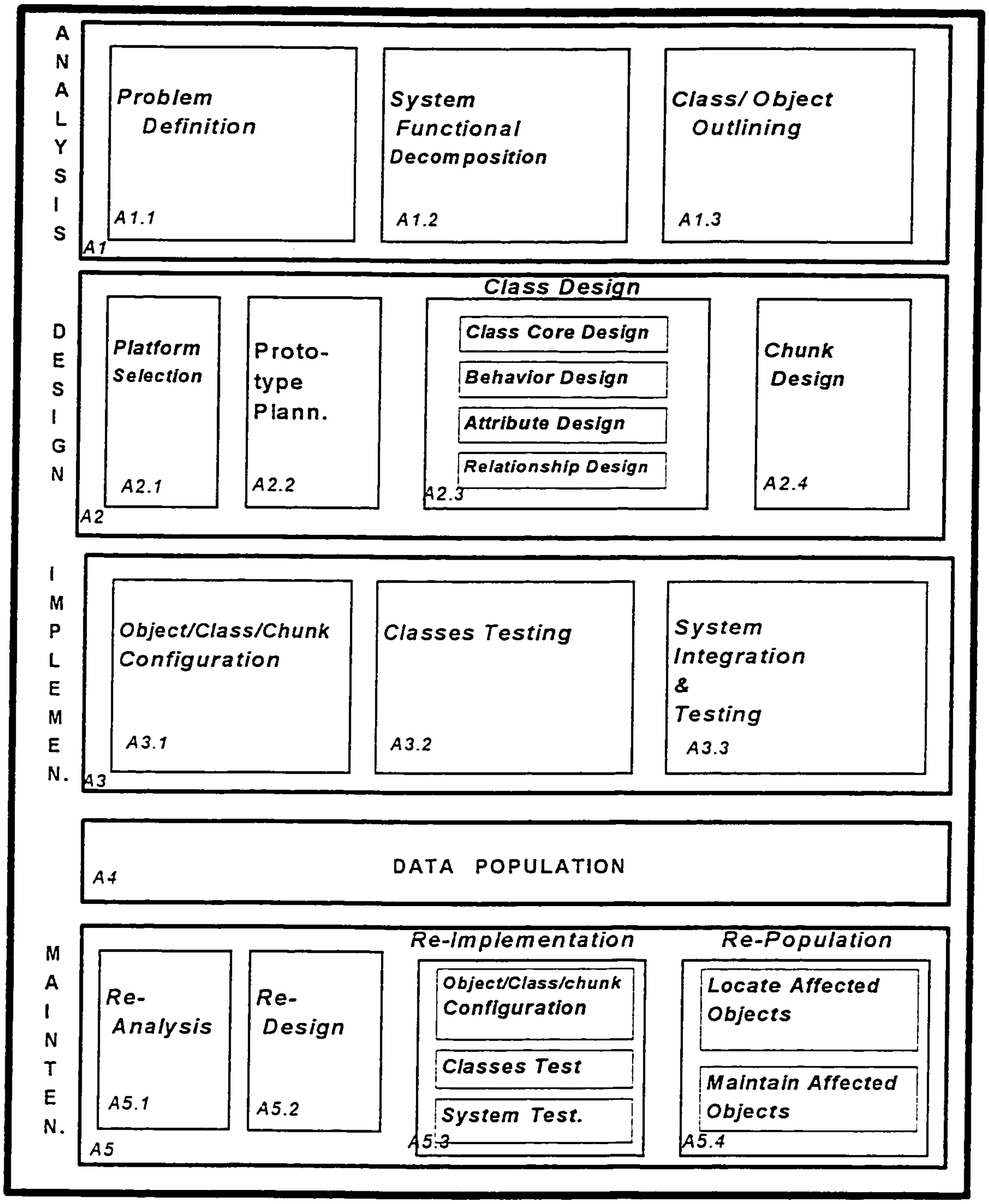


Figure 4.6, The OAS approach to the SDLC

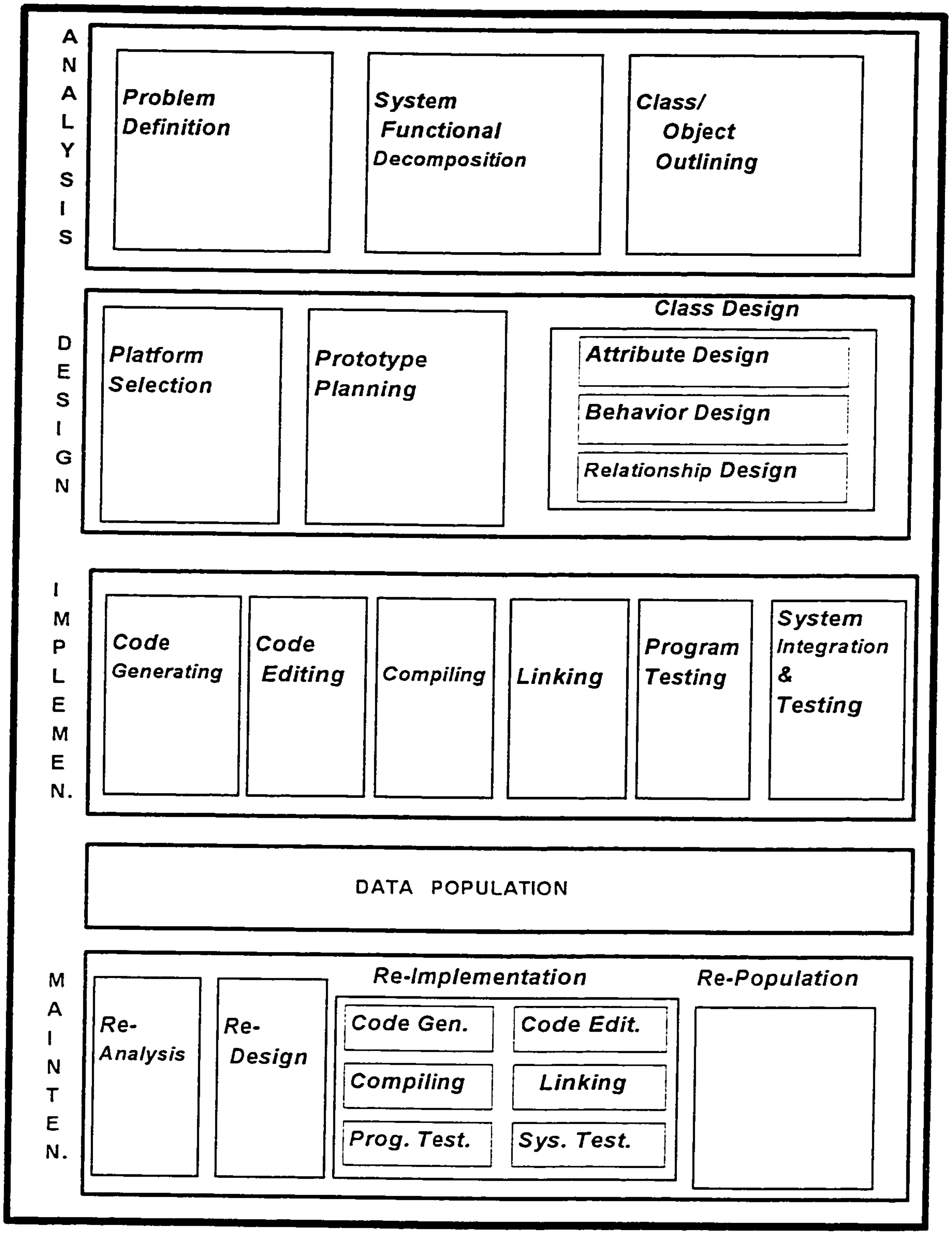


Figure 4.7 Traditional application of the SDLC

The high level view of the SDLC using the OAS, as shown in Figure 4.8, will have the following set of inputs and outputs:

<u>Input</u>

1. User requirements.

2. Knowledge of the domain that the software system will support.

3. Key practices and policies of the enterprise that the software system must support.

<u>Output</u>

1. Functional application software.

2. System data.

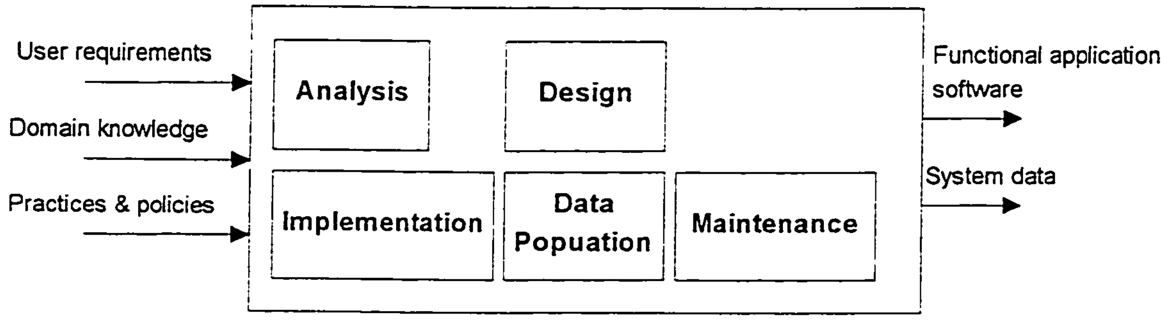


Figure 4.8 Overview of the SDLC

Details of the various sets of functions, inputs, outputs, and their corresponding tools, for each phase, are outlined below:

Function	Input	Output	Tools
Analysis	User requirements, knowledge, policies, and practices of the enterprise that the software must support.	Statement of the problem and description of the system main functional modules. Description of key objects, chunks, classes, their characteristics, key behaviors and relationships. Object to class mapping.	Text editor, data entry forms, and drawing tools to describe the system functional modules, objects, classes, and their characteristics, behaviors and relationships.
Function	**Input**	**Output**	**Tools**
Design	Description of the system main functional modules.	The Class Specs file, the Chunk Specs file, and the Attribute-	The *Design Specification Module* to access the

	Description of key objects, chunks, classes, their characteristics, key behaviors and relationships. Object to class mapping. The Object Library and the ClassNames file.	Method table file. Platform selection and prototype planning.	Object Library allowing the developer to design classes, and chunks as well validating the design specifications.

Function	**Input**	**Output**	**Tools**
Implementation	The Class Specs file, the Chunk Specs file, and the Object Library. Platform selection and prototype planning.	Class and chunk prototypes.	The *Functional Configuration Module* to map and assemble objects from the Object Library to the logical design in order to generate the various class and chunk prototypes defined during the design phase.

Function	**Input**	**Output**	**Tools**
Data Population	Prototype for each class and chunk in the application software.	Persistent data objects.	The *Data Management Module.* to create required objects.

Function	**Input**	**Output**	**Tools**
Maintenance	New user requirements and enhancements, existing class and chunk prototypes as well the persistent data objects.	Updated analysis results, updated design files, updated class and chunk prototypes, and updated data objects.	All the tools listed above in order to change the logical design, class and chunk prototypes. The data management Module to carry out the changes to the underlying data objects.

4.4. Details of the SDLC Phases

The following subsections 4.4.1 through 4.4.5 explain, in details, the OAS approach to the different phases of the SDLC..

4.4.1 Analysis

The analysis phase as shown in Figure 4.9 below consists of three major tasks. The inputs, outputs, and tools are explained below:

Input

1. User requirements in a textual format.
2. Knowledge of the domain that the software system will support.
3. Key practices and policies of the enterprise.

Output

1. Statement defining the problem and objectives of the software system.
2. Textual and diagrammatic description of the main functional modules of the system.
3. Textual and diagrammatic description of key objects, classes, their characteristics, key behaviors, and relationships.
4. Object to class mapping.

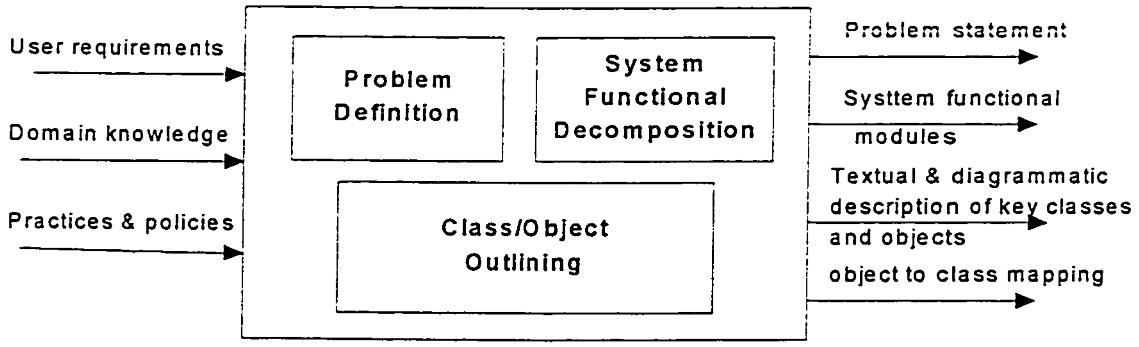


Figure 4.9 The Analysis phase

Function	Input	Output	Tools
Problem definition	User requirements.	Statement of the problem and the system objectives	Text editor and data entry tools.

System functional decomposition	system objectives, domain knowledge. policies, and practices of the enterprise.	semantic and diagramming description of the main functional modules of the application software system	Text editor and drawing tools to show the main functional modules of the system.
Class/object outlining	System functional modules.	Textual and diagrammatic description of. key classes, objects, their characteristics, key behaviors, and relationships. Object to class mapping.	Text editor, data entry and drawing tools to describe the key classes, objects. their characteristics. behaviors. and relationships.

The OAS does not introduce any changes to the analysis phase when compared to the analysis phase in the traditional software development approaches such as CASE.

4.4.2. Design

The design phase of the OAS is shown in Figure 4.10 with the chunk design function added. The inputs, outputs, tools and functions of the design phase are listed below:

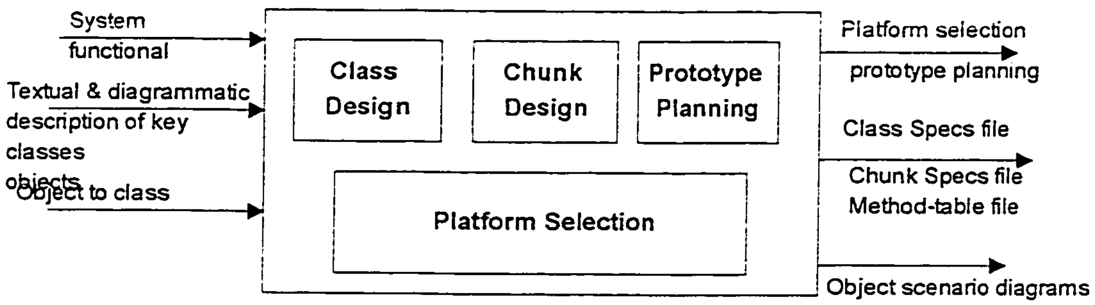


Figure 4.10 The Design phase

Input

1. Textual description of the main functional modules of the system.

2. The Object Library files and the ClasssNames file.

3. Textual description of key objects, chunks, classes, their characteristics, key behaviors, and relationships.
4. Object to class mapping.

Output

1. The Class Specs file, the Chunk Specs file, and the Attribute-Method table file.
2. Platform selection where hardware, operating system, and implementation environments are specified.
3. Prototype planning, which is a plan to develop and test several prototypes in order to verify user requirements and iterate over the analysis and design phases as necessary. The plan contains (1) the number of prototype models needed, (2) conditions for satisfactory completion of prototyping, and (3) which modules should be included in each prototype.

Function	Input	Output	Tools
Class design	Description of objects, classes, their key behaviors, characteristics, and relationships. Objects to class mapping. The Object Library and the ClassNames file.	The Class Specs file and the Attribute-Method table file.	Text editor and the *Design Specifications Module.*
Chunk design	System main functional modules.	The Chunk Specs file.	Text editor and the *Design Specifications Module* for providing the user-interface to help the developer design chunks and provide their corresponding parameters.
Platform selection	User requirements. Enterprise practices and policies.	Choice of the hardware and software environment.	Text editor.

Prototype planning	User requirements.	The number of prototypes, conditions, and the modules to be included in each prototype.	Text editor..

4.4.2.1. Class Design

Class design can be further divided into four design tasks as shown in Figure 4.11, class core, attribute, behavior, and relationship design. These four functions will be explained in detail in the next subsections.

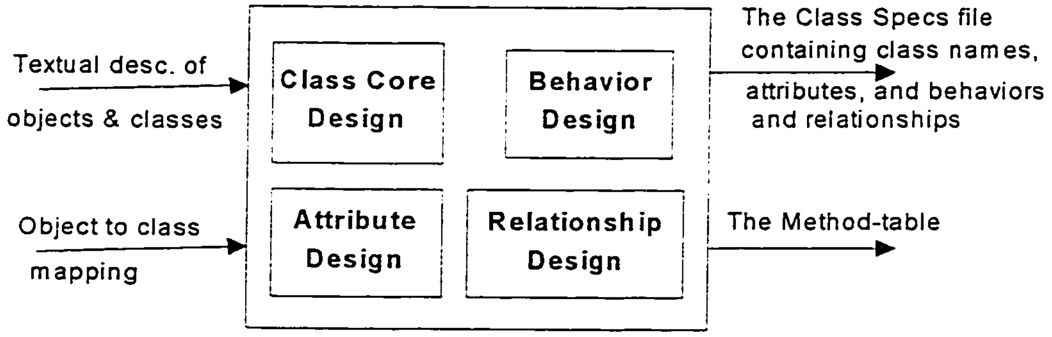

Textual desc. of objects & classes
Object to class mapping
Class Core Design
Behavior Design
Attribute Design
Relationship Design
The Class Specs file containing class names, attributes, and behaviors and relationships
The Method-table

Figure 4.11 Class Design

The inputs and outputs of class design function of the OAS are:

Input

1. Description of the key objects, classes, their key behaviors, characteristics, and relationships.
2. Object to class mapping.
3. The Object Library and the ClassNames file.

Output

1. The Class Specs file, and the Attribute-Method table file.

Class Core Design

The purpose of this step is to define new classes in the MRP system. The functions, inputs, outputs and tools in the class core design step are:

Function	**Input**	**Output**	**Tools**
Class Core design	Description of classes and the ClassNames file.	The Class Specs file.	The *Design Specifications Module* and data entry tools.

Behavior Design

In the OAS, the functions, inputs, outputs and tools in the behavior design step are:

Function	**Input**	**Output**	**Tools**
Behavior design	Description of the various behaviors of the class. The Object Library	The updated Class Specs file and the Attribute-Method table file	The *Design Specifications Module* and data entry tools.

It is important to highlight the fact that the OAS starts with behavior design as opposed to starting with designing class characteristics. This approach will result in producing an flexible software system since no attribute will be defined unless it definitely contributes to supporting class behavior. During the behavior design step, the *Design Specifications Module* updates the Class Specs File by adding the names of the methods and their building blocks to the corresponding class in the file. It also updates the Attribute-Method table file by creating a column representing the new method storing (1) in all the cells of intersection of the target method column and its supporting arguments rows as was previously shown in Table 4.1 of section 4.2. The Attribute-Method table file identifies which attribute is implemented by which method in order to resolve dependencies when deleting an attribute.

Attribute Design

This step is to design the attributes to support the required behavior of a class. The inputs, outputs, functions and tools, for this step are:

Function	Input	Output	Tools
Attribute design	Description of the characteristics of the class. The Object Library.	The updated Class Specs file and the Attribute-Method table file.	The *Design Specifications Module* and data entry tools.

In the design step, the *Design Specifications Module* updates the Class Specs File by adding the names of the attributes, provided by the developer, to the corresponding class in the file. It also updates the Attribute-Method table file by adding an entry (a row) for each attribute added and indicating which method implements which attribute.

Relationship Design

The OAS approach to relationship design of a class will have the following set of functions, inputs, outputs and tools:

Function	Input	Output	Tools
Relationship design	Description of the various relationships that a class has with other classes. The Object Library	The updated Class Specs file.	The *Design Specifications Module* and data entry tools.

Class relationships are considered as attributes, called relationship attributes. The *Design Specifications Module* updates the Class Specs File by adding the names of the relationship attributes to the corresponding class in the file.

4.4.2. 2. Chunk Design

The OAS introduces the concept of chunking to manufacturing software. This concept of grouping objects in a chunk is not addressed in traditional software development tools. The functions, its input, output, and tools implemented in this step will be:

Function	Input	Output	Tools
Chunk design	Description of the various chunks in the application software system.	The Chunk Specs file.	The *Design Specifications Module* to define. chunks and provide their corresponding parameters.

4.4.3. Implementation

The implementation phase has the following inputs and outputs:

Input

1. The Class Specs file.

2. The Chunk Specs file.

3. The Object Library files.

Output

1. Class prototype for each class and configured chunks.

The implementation phase, as shown in Figure 4.12, consists of three functions.

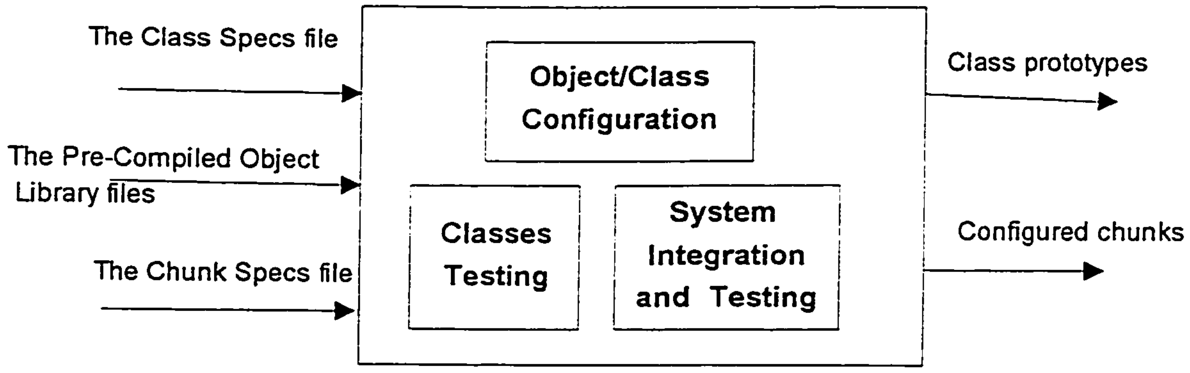

The Class Specs file
The Pre-Compiled Object Library files
The Chunk Specs file
Object/Class Configuration
Classes Testing
System Integration and Testing
Class prototypes
Configured chunks

Figure 4.12 Implementation phase in the OAS approach

Following is an overview of these functions, their inputs, outputs and tools required:

Function	Input	Output	Tools
Object/class configuration	The Class Specs file, the Chunk Specs file, and the Object Library.	Class prototypes and configured chunks.	The *Functional Configuration Module* to map and assemble object from the Object Library to the logical design
Classes testing	Class prototypes and configured chunks.	The expected output of the individual classes.	Testing systems and debuggers of the implementation environment.
System integration and testing	Class prototypes and configured chunks.	Fully functional MRP system which fulfills the user requirements.	Support modules dedicated to integrating the individual programs.

The Implementation phase is where an important extension to traditional software development approaches is proposed in this research. This research uses the concept of pre-compiled objects and open-ended lists of pointers. The Implementation phase in the traditional software development methods takes the output of the design phase and produces a set of hard-coded computer-oriented structures defining classes, their characteristics and behaviors. The OAS, on the other hand, assembles and packages pre-compiled building block to produce class prototypes.

Class attributes and behaviors are represented by open-ended lists containing pointers to pre-compiled building objects selected from the Object Library. This approach adds a high degree of flexibility to the application software since there is no need to edit, compile and link source-code upon making changes such as changing a class hierarchy, attributes or behavior.

Upon completing the implementation phase the MRP application software system will be fully functional and ready for execution.

4.4.4. <u>Data Population</u>

Data population is the process of creating persistent data before or during system use. The input, output, and tools of this step are tabulated below:

Function	**Input**	**Output**	**Tools**
Data Population	Class and chunk prototypes, and user supplied data.	Persistent data objects.	The *Data Management Module.*

4.4.5. <u>Maintenance</u>

System maintenance is defined as any change to the software after delivery to the customer. Most traditional software development approaches such as CASE are capable of only handling changes to the logical design. This requires updating the source code, compiling, and linking. Moreover, in the traditional approach, propagating these changes to the underlying data objects, or re-population, has to be done manually.

As shown in Figure 4.13 ,system maintenance phase consists of four functions, namely: Re-Analysis, Re-Design, Re-Implementation and Re-Population.

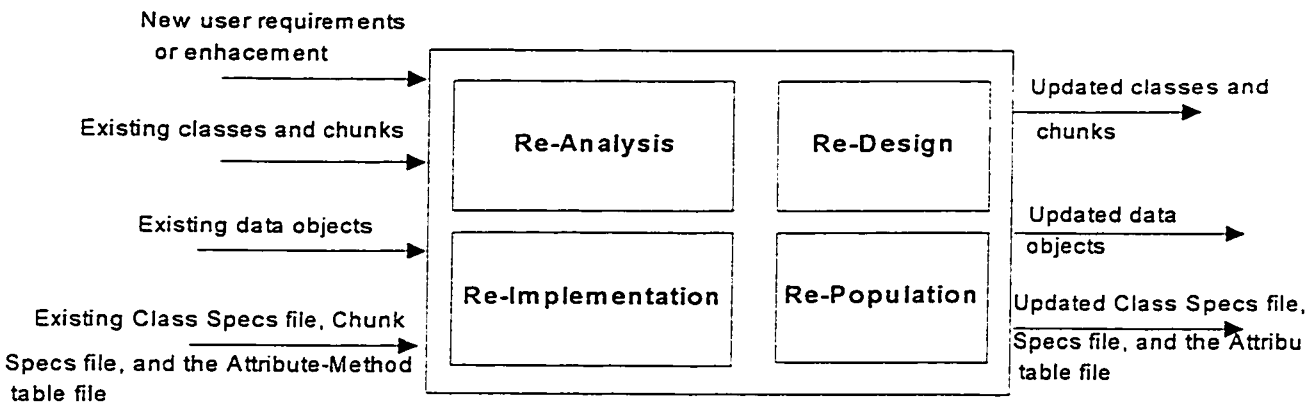

New user requirements or enhacement
Existing classes and chunks
Existing data objects
Existing Class Specs file, Chunk Specs file, and the Attribute-Method table file
Re-Analysis
Re-Design
Re-Implementation
Re-Population
Updated classes and chunks
Updated data objects
Updated Class Specs file, Specs file, and the Attribu table file

Figure 4.13 The Maintenance phase

In the OAS, the maintenance phase will have the following set of inputs and outputs:

<u>Input</u>

1. New user requirements or enhancements.

2. Existing Class specs file, Chunk Specs file, and the Attribute-Method table file.

3. Existing class and chunk prototypes.

4. Existing data object files.

<u>Output</u>

1. Updated Class specs file, Chunk Specs file, and the Attribute-Method table file.

2. Updated class and chunk prototypes.

3. Updated data object files.

The four functions, in turn, will have the following input, output, and tools associated with them:

Function	Input	Output	Tools
Re-analysis	New user requirements or enhancements, and existing analysis results.	Updated analysis results.	Text editor, data entry forms, and drawing tools, in edit mode, to describe the system functional modules, objects, classes, their characteristics, behaviors, and

			relationships.
Re-design	New user requirements or enhancements, updated analysis results, and existing Class Specs file, Chunk Specs file, and the Attribute-Method table file.	Updated Class Specs file, updated Chunk Specs file, and updated Attribute-Method table file.	Text editor, data entry forms, and drawing tools, in edit mode, to define classes and chunks. The *Design Specifications Module* for designing classes and chunks as well as validating the design specifications.
Re-implementation	Updated Class Specs file, and Chunk Specs file.	Updated class and chunk prototypes.	*Functional Configuration Module* to select objects from the *Pre-Compiled Object Library* to assemble the various class and chunk prototypes described during the design phase.
Re- Population	Updated class and chunk prototypes. Existing data object files.	Updated data object files.	The *Data Management Module.*

The various types of maintenance tasks that the OAS is capable of handling are listed below:

- Maintenance of Classes

1. Maintenance of class inheritance

    Creating a new class

    Deleting a class

    Dynamic Inheritance: (Changing a superclass)

2. Maintenance of class behavior

    Adding a method to a class

    Deleting a method from a class

    Modifying a method's implementation

3. Maintenance of class structure

Adding an attribute to a Class

Deleting an attribute from a Class

Changing an attribute's domain

Changing an attribute's default value

Adding a relationship to a class

Deleting a relationship from a class

Maintenance of Objects

Creating a new object

Deleting an object

Creating a relationship between two objects

Removing a relationship between two objects

Changing the value of an object attribute

Object Migration: (Changing an object-class membership)

- Maintenance of Chunks

Creating a chunk

Deleting a chunk

Assembling a chunk

It should be mentioned that maintaining the Object Library, such as adding a new behavior building block, is an off-line activity that may require some source coding with compiling and linking.

4.4.5.1. Re-Analysis

Re-Analysis is the first submodule to be executed during system maintenance. its inputs, outputs, and tools are:

Input

1. New user requirements or enhancements.

2. Existing analysis results.

Output

1. Updated analysis results.

Tools

Text editor, data entry forms, and drawing tools, in edit mode, to describe the system functional modules, key classes, their characteristics, behaviors, and relationships.

4.4.5.2. Re-Design

In this step the inputs, outputs, and tools are:

Input:

1. New user requirements or enhancements.

2. Updated analysis results.

3. Existing Class Specs file, Chunk Specs file, and the Attribute-Method-table file.

Output:

1. Updated Class Specs file, Chunk Specs file, and the Attribute-Method-table file.

Tools

Text editor, data entry forms, and drawing tools, in edit mode, to define system functional modules, classes and chunks. The *Design Specifications Module* for designing classes and chunks as well as validating the design specifications.

4.4.5.3. Re-Implementation

In the OAS, re-implementation is very much similar to implementation.

Input:

1. Updated Class Specs file, Chunk Specs file, and the Attribute-Method-table file.

Output:

1. Updated class and chunk prototypes.

Tools

Functional Configuration Module to select objects from the *Object Library* to assemble the various class and chunk prototypes described during the design phase.

4.4.5.4. Re-Population

The OAS, unlike traditional development approaches, is capable of automatically propagating all changes made to the logical design to the affected underlying data objects. The *Data Management Module* will be invoked to accomplish this task.

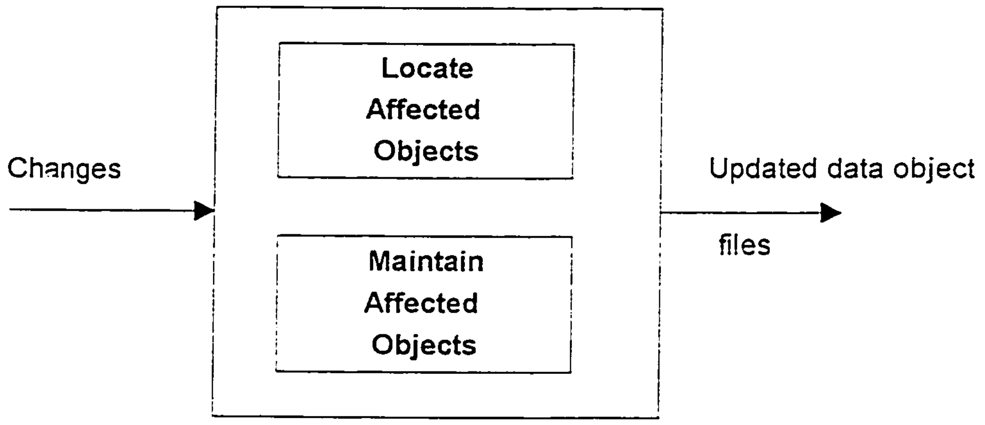


Figure 4.14 Re-Population

As shown in Figure 4.14, the inputs, outputs, and tools to re-population are:

Function	Input	Output	Tools
Locate affected objects	Changes introduced by the developer.	Classes, objects and chunks to be changed are located by the *Data Management Module.*	The *Data Management Module.*
Update affected objects	Classes, objects, or chunks to be changed	Modified classes, objects, or	The *Data Management*

	and type of change.	chunks.	*Module.*

4.5. The Modification Scheme

The modification schemes, implemented in order to modify the underlying data objects, found in the literature are:

1. The *Versionning* approach described in [Zdonic 1990] involves using a mechanism of class versions, generating a new version of a given class each time the class is modified. Another mechanism is required to connect the different versions of a class to each other and present the user with the desired version.
2. The *Deferred* or *Screening* approach used in ORION described in [Banerjee et al. 1987] where modifications to objects are deferred (possibly indefinitely) until they are used again. Objects are not actually modified but rather screened in order to be presented to the user.
3. The *Conversion* approach implemented in Gemstone described in [Penny et al. 1987]. In this approach all the objects are modified immediately to conform to the new class definition.

The first approach requires two mechanisms and versioning is not a desired option in MRP systems since the end-users usually need to see an accurate snapshot of their inventory at any given time. The screening approach causes a degregation of performance due to screening references to or from objects of a modified class. It also requires a permanent mechanism throughout the system life time. It does not appear possible to determine when the system can stop screening objects of modified classes. It also requires a permanent mechanism throughout the system life time.

The modification scheme adopted in this research is similar to the *Conversion* approach implemented in the Gemstone data model. The decision was made based on the following reasons:

1. Performance and speed of execution are highly desired options in MRP systems.
2. Consistency of data is guaranteed when implementing the conversion approach.
3. No overhead mechanisms are needed for versioning or screening.

Chapter 5

THE DETAILED FUNCTIONAL BUILDING BLOCKS AND OPERATING PROCEDURES OF THE OAS

This chapter describes to the functional modules and the operational procedures of the proposed OAS. Section 5.1 lists all the functional_modules, their building blocks, and a brief description of the task of each module. Section 5.2 explains, in details, the processes performed by each functional building block in the architecture. Section 5.3, on the other hand, presents the Object Oriented class design of the OAS. Section 5.4 describes the procedures for operating the system.

5.1. Functional Modules and their Building Blocks of the OAS

The functional modules of the proposed OAS, as shown in Figure 5.1, consist of the following components:

1. Coordination Manager
2. Design Specifications Module
3. Functional Configuration Module
4. Data Management Module
5. Maintenance Module

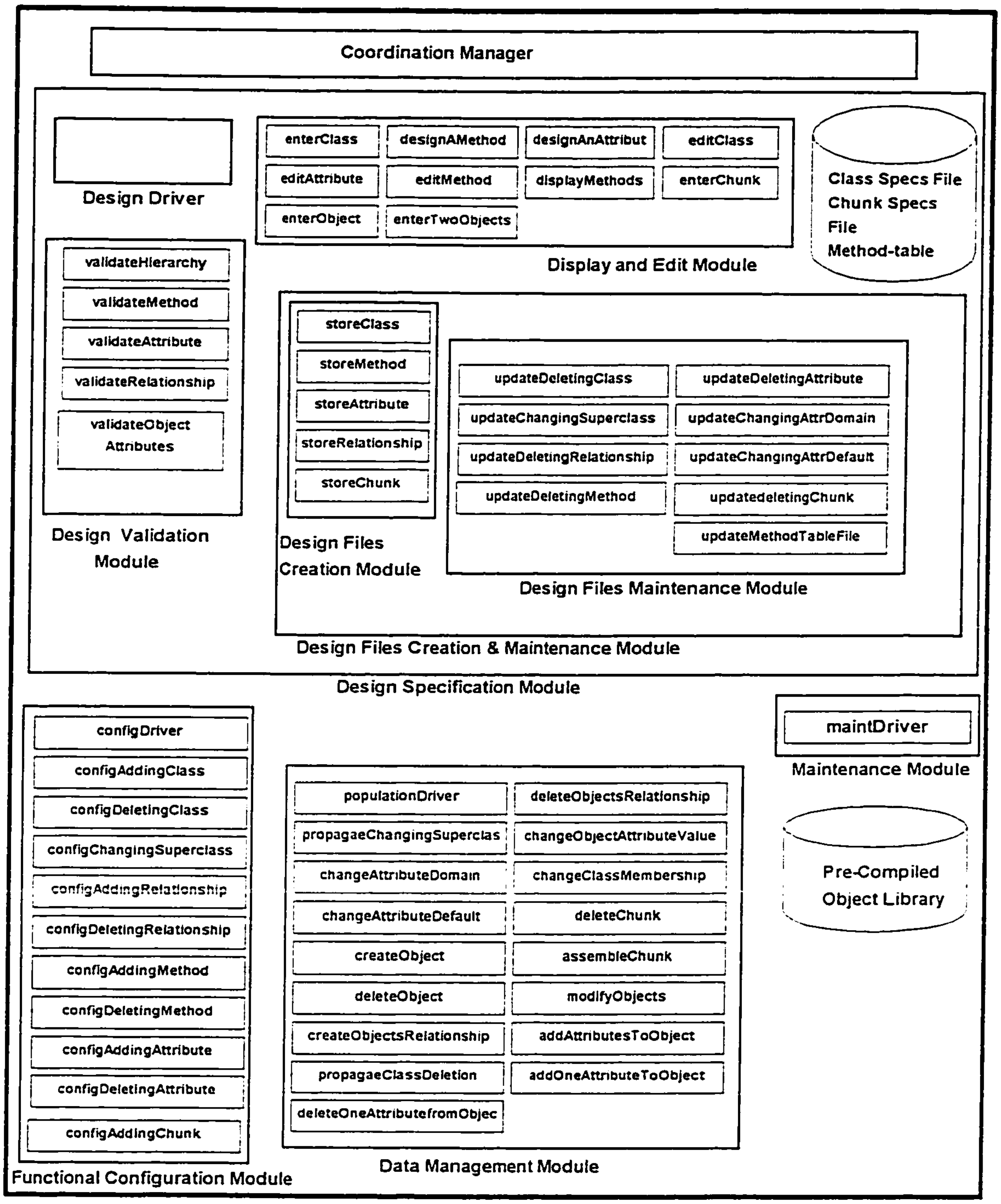


Figure 5.1, Building blocks of the OAS

5.1.1. Coordination Manager

The Coordination Manager is the key module responsible for invoking all other modules of the OAS. The main menu of the Coordination Manager allows the developer to use the different functionalities of the system such as analysis, design, or implementation.

5.1.2. Design Specifications Module

The *Design Specifications Module* is responsible for performing design tasks, design validation, and creation of the design files. The *Design Specifications Module* consists of the following submodules:

1. Design Driver
2. Display and Edit Module
3. Design Validation Module
4. Design Files Creation Module.
5. Design Files Maintenance Module.

The names of the building blocks of each submodule and a brief description of its task are listed below:

Design Driver

The Design Driver is responsible for invoking other building blocks in order to perform the various design tasks such as class core design, class behavior, class structure, and chunk design.

Display and Edit Module

This module displays the various design specification items such as, a class with its attributes and methods, in order to allow the developer to edit and modify them. The Display and Edit Module consists of the following building blocks:

enterClass: A user-interface to allow the developer to enter a class name and its superclass.

designAMethod: A user-interface to display the behavior building blocks of the Pre-Compiled Object Library and allow the developer to select, drag and drop behavior blocks to compose a method.

designAnAttribute: A user-interface to display a group of icons representing the various attributes of the Object Library allow the developer to select an attribute and provide a name, domain and default value for it.

editClass: A user-interface to display names of a class, its superclass, attributes, methods, relationships and its objects and allow the developer to select any of them.

editAttribute: A user-interface to display an attribute name, domain, and default value and allow the developer to edit the default value or domain. The *editAttribute* then prompt the developer, , in case of changing the attribute domain, to provide an algorithm in order to change the value of the attribute in the underlying objects of the target class..

enterChunk: A user-interface to display all the chunk names in the application software and allow the developer to specify a chunk name and enter its corresponding parameters.

enterObject: A user-interface to allow the developer to enter an object name and provide values for its attributes.

editMethod: A user-interface to allow the developer to edit a method implementation.

enterTwoObjects: A user-interface to create a relationship between two objects by allowing the developer to enter two object names and the name their target relationship.

displayMethods: A user-interface to display all the method names which use a target attribute.

Design Validation Module

The function of this module is to validate all design specifications. Following is a list of the building blocks of the Design Validation Module, and a brief description of their corresponding functions:

validateHierarchy: Checks whether the name of a class and its superclass provided by the developer makes a valid class/superclass relationship.

validateMethod: Checks if a target method name exists in a certain class.

validateAttribute: Checks if a target attribute name exists in a certain class.

validateRelationship: Checks if a target relationship name exists in a certain class.

validateObjectAttriibutes: Checks if the data type and value of each attribute of a target object conform to the attributes data type, domain, and default value declared in the class definition.

Design Files Creation Module

This module is responsible for creating the design files: the Class Specs file, Chunk Specs file, Attribute-Method table file, and the ClassNames file. This module has the following submodules:

storeClass: Creates a record in the Class Specs file for each class in the MRP system and saves the class name and its superclass.

storeMethod: Saves a method in the "Method-list" of the target class record of the Class Specs file. It also creates a column for the method in the Attribute-Method table file.

storeAttribute: Saves an attribute in the "Attribute-list" of the target class record of the Class Specs file. It also creates a row, for the attribute, in the Attribute-Method table file.

storeRelationship: Saves a relationship in the "Relationship-list" of the target class record of the Class Specs file.

storeChunk: Saves a chunk name along with its parameters in the Chunk Specs file.

<u>**Design Files Maintenance Module**</u>

This module updates the three design files, the Class Specs file, the Chunk Specs file, and Attribute-Method table file, upon any change initiated by the developer during the maintenance phase. The building blocks of the Design Files Maintenance Module are:

<u>*updateDeletingClass*</u>: Deletes a class record from the Class Specs file.

<u>*updateChangingSuperclass*</u>: Changes the superclass of a class in the Class Specs file.

<u>*updateDeletingRelationship*</u>: Deletes a relationship from a class in the Class Specs file.

<u>*updateDeletingMethod*</u>: Deletes a method from a class in the Class Specs file.

<u>*updateDeletingAttribute*</u>: Deletes an attribute from a class in the Class Specs file.

<u>*updateChangingAttrDomain*</u>: Changes the domain of a target attribute of a class in the Class Specs file.

<u>*updateChangingAttrDefault*</u>: Changes the default value of a target attribute of a class in the Class Specs file.

<u>*updateDeletingChunk*</u>: Deletes a chunk record from the Chunk Specs file.

<u>*updateMethodTableFile*</u>: Deletes a method or an attribute from the Attribute-Method table file

5.1.3. <u>Functional Configuration Module</u>

The task of the Functional Configuration module is to select objects from the Object Library in order to assemble and package the class and prototypes designed during the design phase. The building blocks of this module are:

<u>*configDriver*</u>: Coordinates the execution of all the configuration tasks.

<u>*configAddingClass*</u>: Creates a class core prototype.

<u>*configDeletingClass*</u>: Deletes a class prototype.

configChangingSuperclass: Changes the superclass of a class.

configAddingRelationship: Adds a relationship object to a class prototype.

configDeletingRelationship: Deletes a relationship object from a class prototype.

configAddingMethod: Adds a method object to a class prototype.

configDeletingMethod: Deletes a method object from a class prototype.

configAddingAttribute: Adds an attribute object to a class prototype.

configDeletingAttribute: Deletes an attribute object from a class prototype.

configAddingChunk: Creates a chunk object.

5.1.4. Data Management Module

The Data Management Module is responsible for performing the task of populating the MRP system with data as well as propagating changes to the physical underlying data objects during maintenance. The Data Management Module consists of the following submodules:

populationDriver: Directs and coordinates all the activities to carry out all the changes to the underlying data objects.

modifyObjects: Modifies all the objects in a target class and its subclasses upon a adding or deleting attributes or relationships.

changeAttributeDomain: Changes the domain of an attribute in all the objects of a target class.

changeAttributeDefault: Changes the default value of an attribute in all the objects of a target class.

propagateClassDeletion: Modifies the structure of all the objects in a target class and its subclasses when deleting the class *propagateChangingSuperclass*: Modifies the structure of all the objects in a target class and its subclasses when changing the superclass of the target class..

addAttributesToObject: Adds attributes to an object.

addOneAttributeToObject: Adds only one attribute to an object.

deleteOneAttributeFromObject: Removes only one attribute from an object.

createObject: Creates an object.

deleteObject: Deletes an object.

createObjectsRelationship: Creates a relationship between two objects.

deleteObjectsRelationship: Deletes a relationship between two objects.

changeObjectAttributeValue: Changes the value of an object attribute.

changeClassMembership: Changes the membership of an object in a class.

deleteChunk: Deletes a chunk.

assembleChunk: Populates a chunk with objects.

5.1.5. Maintenance Module

The task of this module is to coordinate the maintenance tasks, it has one building block.

maintDriver: Directs and coordinates the execution of the appropriate building blocks to perform the various maintenance tasks.

5.2. Processes of the Functional Building Blocks of the OAS

This section will explain in details all the processes performed by each building block of the OAS functional modules.

5.2.1. Coordination Manager

1. Display a screen with five options: Analysis, Design, Implementation, Population,

Maintenance, and Exit the system.

2. If option is Design, then the *designDriver* will be invoked.
3. If option is Implementation, then the *configDriver* will be invoked.
4. If option is Population, then the *populationDriver* will be invoked.
5. If option is Maintenance, then the *maintDriver* will be invoked.
6. If option is to Exit the system, then stop.

5.2.2. Design Specifications Module

designDriver

The *designDriver* will permit the developer to select one of the following with six options: Class Core Design, Behavior Design, Attribute Design, Relationship Design, Chunk Design, and going back to the OAS main menu.

Display and Edit Module

enterClass

1. Access the Class Specs file and the Library ClassNames file.
2. Display a screen showing a list of the class names in both the Class Specs file the Pre-Compiled Object Library files.
3. Prompt the developer to provide (or select) a class name and its superclass.

designAMethod

1. Access the Class Specs file and display a list of the class names in the Class Specs file and prompt the developer to select a class name.
2. Display the attribute names of the target class.
3. Access the Object Library files and display a list of its behavior building blocks.

4. Allow the developer to drag and drop any behavior building block from the library, provide the required arguments and assemble the blocks in a sequence that accomplishes the desired behavior.
5. Prompt the developer to provide a method name.

<u>designAnAttribute</u>

1. Access the Class Specs file and display a list of the class names in the Class Specs file and prompt the developer to select a class name.
2. Access the Object Library files to display all the data types of the Pre-Compiled Object Library and allow the developer to select one.
3. Prompt the developer to provide a name for the target attribute, a domain and a default value when applicable.

<u>editClass</u>

1. Access the Class Specs file and display a list of the class names in the Class Specs file and prompt the developer to select a class name.
2. Upon selecting a class name, display a screen to display the name of the target class, its superclass, attributes names along with their data types, domains and defaults, method names, and objects names.
3. Allow the developer to select the superclass name, any attribute or method name.

<u>editAttribute</u>

1. Display a screen showing the target attribute name, its data type, default, domain.
2. Allow the developer to modify the domain or the default value and provide an algorithm in order to change the attribute values of existing objects, when applicable.

editMethod

1. Access the Object Library files and display a list of its behavior building blocks.
2. Display the attribute names of the target class.
3. Display the building blocks of the target method and allow the developer to delete any block or change its arguments. Also allow the developer to drag and drop any building block from the Object Library, supply the required arguments and assemble the blocks in a sequence that accomplishes the desired behavior.

enterObject

1. Display the attribute names of the target class and prompt the developer to provide values for them.
2. Prompt the developer to provide an object name.

enterTwoObjects

1. Access the Class Specs file and display a list of the class names in the Class Specs file and prompt the developer to select one or more class name.
2. Upon selecting class name(s) display a list of relationship names of the class selected (or the first class if more that one class are selected) and prompt the developer to select a relationship name.
3. Upon selecting a relationship name, display a screen prompting the developer to provide two object names (a source and a destination).

displayMethods

1. Access the Class Specs file and display a list of the class names in the Class Specs file and prompt the developer to select a class name.

2. Upon selecting a class name, display a screen to display the name of the target class, its superclass, attributes names along with their data types, domains and defaults, method names, and objects names.
3. Allow the developer to select a target attribute.
4. Search the Attribute-Method table file for the target attribute.
5. Display a list of all method names which use the target attribute.

<u>enterChunk</u>

1. Access the Chunk Specs file.
2. Display a list of all the chunk names in the Chunk Specs file and prompt the developer to provide a chunk name.
3. Upon selecting a chunk, prompt the developer to provide the chunk parameters-list.

<u>Design Validation Module</u>

<u>validateHierarchy</u>

1. Search the Class Specs file for the superclass name (provided by the developer) of the target class and verify that it is not a subclass of the target class. If it is, then return FALSE to indicate invalid hierarchy else return TRUE.
2. Search the Class Specs file for the subclasses of the target class and verify that none of them is a superclass of the superclass (provided by the developer). If this test is true then return TRUE to indicate a valid hierarchy else return FALSE.
3. If both step 1 and 2 returns TRUE then this submodule will return TRUE to indicate a valid hierarchy else return FALSE.

<u>validateMethod</u>

1. Search the Class Specs file for the target class record.

2. If the target method name is found in the target class then return TRUE else return FALSE

validateAttribute

1. Search the Class Specs file for the target class name.
2. If the target attribute name is found in the target class then return TRUE else return FALSE.

validateRelationship

1. Search the Class Specs file for the target class name.
2. If the target relationship name is found in the target class then return TRUE else return FALSE.

validateObjectAttributes

1. For each attribute in the target object
    1.1. If its data type matches the data type of that attribute declared in the class definition in the Class Specs file then return TRUE
    else return FALSE.
    1.2. If the value is not less and not greater that the attribute domain declared in the class definition in the Class Specs file then return TRUE
    else return FALSE.
2. If the return value if TRUE in both 1.1, and 1.2 then return TRUE else return FALSE.

Design Files Creation Module

storeClass

1. Access the Class Specs file and create a record for the target class.
2. Save the name of the target class and its superclass in the newly created record.

3. Search the Class Specs file for the subclasses of the target class.

4 For each subclass record:

    store the name of the target class in the (*superclass-name*)

    save the subclass record.

<u>storeMethod</u>

1. Search the Class Specs file for the target class record.

2. Add the name of the target method, as well as the list of its building blocks, to the (*Method-list*) of the target class in the Class Specs file, and save the target class record.

3. Access the Attribute-Method table file and create a column for the target method.

4. For each supporting argument of the target method:

    Search the rows of the Attribute-Method table file for the supporting attribute, and store (1) in the cells where the target method column intersects with the row of its supporting attribute to indicate the dependency relationship.

5. Save the Attribute-Method table file.

<u>storeAttribute</u>

1. Search the Class Specs file for the target class record.

2. Add the target attribute name to the (*Attribute-list*) of the target class in the Class Specs file, and save the target class record.

3. Access the Attribute-Method table file and create a row for the target attribute.

4. Search the columns of the Attribute-Method table file for the methods that implement the target attribute, and store (1) in the cells where the target attribute row intersects with the columns of methods which implement it. to indicate the dependency relationship.

5. Save the Attribute-Method table file

<u>storeRelationship</u>

1. Search the Class Specs file for the target class record.
2. Add the target relationship name to the (*Relationship-list*) of the target class in the Class Specs file and save the target class record.

<u>storeChunk</u>

1. Access the Chunk Specs file and create a record for the target chunk.
2. Store the target chunk name along with its parameters-list in the newly created record.

<u>Design Files Maintenance Module</u>

<u>updateDeletingClass</u>

1. Search the Class Specs file for the records of the superclass and subclasses of the target class.
2. For each subclass:

    Remove the target class name from the (*Superclass-name*)

    Store the name of the superclass of the target class in the (*Superclass-name*)

    Save the subclass record.

3. Delete the target class record from the Class Specs file.

<u>updateChangingSuperclass</u>

1. Search the Class Specs file for the target class record.
2. Store the name of the new superclass in the *(superclass-name)* of the target class in the Class Specs file and save the target class record.

<u>updateDeletingRelationship</u>

1. Search the Class Specs file for the target class record.

2. Remove the name of the target relationship from the (*Relationship-list)* of the target class and save the target class record.

updateDeletingMethod

1. Search the Class Specs file for the target class record.
2. Remove the name of the target method from the (*Method-list*) of the target class and save the target class record.

updateDeletingAttribute

1. Search the Class Specs file for the target class record.
2. Remove the name of the target attribute from the (*Attribute-list*) field of the target class and save the target class record.

updateChangingAttrDomain

1. Search the Class Specs file for the target class record.
2. Replace the domain of the target attribute in the (*Attribute-list*) of the Class Specs file with the new Domain provided by the developer and save the target class record.

updateChangingAttrDefault

1. Search the Class Specs file for the target class record.
2. Replace the default value of the target attribute in the (*Attribute-list*) of the Class Specs file with the new Default provided by the developer and save the target class record.

updateDeletingChunk

1. Search the Chunk Specs file for the record of the target chunk.
2. Remove the record of the target chunk from the Chunk Specs file.

updateMethodTableFile

1. If *maintOption* is Deleting a method from a class then:

    Search the Attribute-Method table file for the target method column.

    Remove the column of the target method from the Attribute-Method table file.

2. If *maintOption* is Deleting an attribute from a class then:

    Search the Attribute-Method table file for the target attribute row.

    Remove the row of the target attribute from the table.

3. Save the Attribute-Method table file.

5.2.3. Functional Configuration Module

configDriver

1. If Creating a new class then call *configAddingClass* submodule.
2. If Deleting class then call *configDeletingClass* submodule.
3. If Changing a superclass then call *configChangingSuperclass* submodule.
4. If Adding a relationship to a class then call *configAddingRelationship* submodule.
5. If Deleting a relationship from a class then call *configDeletingRelationship* submodule.
6. If Adding a method to a class then call *configAddingMethod* submodule.
7. If Deleting a method from a class then call *configDeletingMethod* submodule.
8. If Adding an attribute to a class then call *configAddingAttribute* submodule.
9. If Deleting an attribute from a class then call *configDeletingAttribute* submodule.
10. If Creating a chunk then call *configAddingChunk* submodule.

configAddingClass

1. Search Object Library or the MRP system for the superclass of the

    target class.

2. Create a class prototype for the target class as a subclass of the superclass (provided by the developer) by:

   Creating a proxy of the superclass prototype

   Storing both ID of the superclass in the (*Superclass-id)* of the target class and the name of the superclass in the (*Superclass-name)* of the target class.

3. Save the target class prototype..

<u>configDeletingClass</u>

1. Search the class files of the MRP system for the target class prototype.
2. Search for the class prototype of the subclasses of the target class.
3. For each subclass:

   Remove the ID and the name of the target class from the(*Superclass-id),* and the (*Superclass-name)* respectively

   Store both the ID and name of the superclass of the target class in the(*Superclass-id),* and the (*Superclass-name)* respectively

4. Save the subclasses of the target class.
5. Delete the target class file.

<u>configChanging-Superclass</u>

1. Search the class files of the MRP system for the target class prototype.
2. Search for the prototype of the new superclass of the target class.
3. Remove the ID and name of the old superclass from the (*Superclass-id)* and (*Superclass-name)* of the target class and store in it the name and ID of the new superclass instead..
4. Save the target class prototype.

<u>configAddingRelationship</u>

1. Search the class files of the MRP system for the target class prototype.
2. Search the Object Library for the class representing the relationship attribute.
3. Create an instance using the target relationship name and store its ID in the *(Attribute-list)* of the target class prototype.
4. Save the target class prototype.

<u>configDeletingRelationship</u>

1. Search the class files of the MRP system for the target class prototype.
2. Search for the target relationship in the *(Attribute-list)* of the target class prototype.
3. Remove the ID of the target relationship from the *(Attribute -list)* of the target class prototype.
4. Save the target class prototype.

<u>configAddingMethod</u>

1. Search the class files of the MRP system for the target class prototype.
2. Create an instance of class MethodCreator, using the target method name provided by the developer, to store the method building blocks listed in the Class Specs file.
3. Store the ID of the target method object in the *(Method-list)* of the target class prototype.
4. Save the target class prototype.

<u>configDeletingMethod</u>

1. Search the class files of the MRP system for the target class prototype.
2. Remove the ID of the target method from the *(Method-list)* of the target class

prototype.

3. Save the target class prototype.

ConfigAddingAttribute

1. Search the class files of the MRP system for the target class prototype.
2. Search the Object Library for the class representing the target attribute as described in the Class Specs file.
3. Create an instance using the target attribute name and store its ID in the *(Attribute-list)* of the target class prototype.
4. Save the target class prototype.

configDeletingAttribute

1. Search the class files of the MRP system for the target class prototype.
2. Remove the ID of the target attribute from the *(Attribute-list)* of the target class prototype.
3. Save the target class prototype.

configAddingChunk

1. Search the Object Library for class ***Chunk.***
2. Create an instance of class ***Chunk*** using the chunk name in the Chunk Specs file and store the chunk parameters in it.
3. Save the target chunk.

5.2.4. Data Management Module

populationDriver

1. If maintOption is Deleting a class then *propagateClassDeletion* will be invoked to propagate the appropriate changes.

2. If maintOption is Changing a superclass then *propagateChangingSuperclass* will be invoked to propagate the appropriate changes.
3. If maintOption is Adding a relationship to a class, Deleting a relationship from a class, Adding an attribute to a class, or Deleting an attribute from a class, then *modifyObjects* will be invoked to propagate the appropriate changes to the affected objects.
4. If maintOption is Changing an attribute domain then call *changeAttributeDomain.*
5. If maintOption is Changing an attribute default value then call *changeAttributeDefault.*
6. If maintOption is Deleting a chunk then call *deleteChunk* submodule.

<u>propagateClassDeletion</u>

```
1. Search the class files of the MRP system for the target class prototype.
2. Store the attributes of the target class in a temporary area.
3. Search the class files of the MRP system for the subclasses of the target class.
4. Search the MRP data files for the object under each class in step 1 and 3.
        For each object:
                for each attribute of the target class ( temporarily held in step 2)
                        call deleteOneAttributeFromObject
                        Save the object.
5. If the objects of the target class is to "migrate to another class" then search the class
   files of the MRP system for the "migrate to class" and its superclasses.
6. Search the MRP data files for the object under each class in step 5.
        For each object:
                call addAttributesToObject
                Save the object.
```

<u>propagateChangingSuperclass</u>

```
1. Search the class files of the MRP system for the target class prototype.
2. Search the class files of the MRP system for the old superclass of the target class.
3. Store the attributes of the old superclass in a temporary area.
4. Search the class files of the MRP system for the subclasses of the target class.
5. Search the MRP data files for the objects under each class in step 1 and 4.
        For each object:
                for each attribute of the old superclass ( temporarily held in step 3)
                        call deleteOneAttributeFromObject
                        Save the object.
6. Search the class files of the MRP system for the new superclass of the target class.
7. Store the attributes of the new superclass in a temporary area.
8. Search the MRP data files for the objects under each class in step 1 and 4.
        For each object:
                for all attribute of the new superclass ( temporarily held in step 7)
                        call addAttributesToObject
                        Save the object.
```

<u>addAttributesToObject</u>

```
1. Compare the attribute set of the new class (Held in a temporary area) with the attribute
   set of the target object.
2. For each attribute that exists in the new class but not in the target object
        call addOneAttributeToObject.
```

<u>addOneAttributeToObject</u>

1. Search the Object Library for the class that matches the data type of the target attribute.
2. Create its instance object using the name and default value defined in the class definition.
3. Store the ID of the object created in step 2 in the target object.

<u>removeOneAttributeFromObject</u>

1. Remove the ID of the target attribute from the attribute-list of the target object.

<u>modifyObjects</u>

```
1. Search the MRP class files for the target class prototype.
2. Identify the target relationship or attribute to be added or deleted.
3. Search the MRP class files for the subclasses of the target class.
4. Search the MRP data files for the object under each class in step 1 and 3.
        For each object:
                If Deleting an Attribute or Relationship
                        call deleteOneAttributeFromObject
                If Adding an Attribute or Relationship
                        call addOneAttributeToObject
                save the object.
```

<u>changeAttributeDomain</u>

1. Search the class files of the MRP system for the target class prototype.
2. Search the MRP data files for the objects under the target class.
3. For each object:

Change the value of the target attribute be scanning and executing the "algorithm" provided by the developer (the algorithm will take the following form:

change factor, operator such as *, +, /)

and the new domain will take the following value:

new domain value = old domain multiplied or added (or any other operation) to the change factor

Save the object.

<u>changeAttributeDefault</u>

1. Search the class files of the MRP system for the target class prototype.
2. Search the MRP data files for the objects under the target class.
3. For each object:

    Change the default value of the target attribute to the value provided by the developer.

    Save the object.

<u>createObject</u>

1. Search the class files of the MRP system for the target class prototype.
2. Create the desired object as an instance of the target class prototype.
3. Call *validateObjectAttributes*.
4. If validation is TRUE (all attribute values provided are valid) then

    for each attribute defined in the class prototype

        call *addOneAttributeToObject* using the attribute values provided by the developer.

5. Save the object.

<u>deleteObject</u>

1. Search the MRP data files for the target object.
2. Delete the target object.

<u>createObjectsRelationship</u>

1. Search the MRP data files for the target objects.
2. Store the ID of the destination object in the *(Attribute-list)* of the source object along with the name of the target relationship.
3. Save the destination object.

<u>deleteObjectsRelationship</u>

1. Search the MRP data files for the target objects.
2. Remove the ID of the destination object from the *(Attribute-list)* of the source object.
3. Save the destination object.

<u>changeObjectAttributeValue</u>

1. Search the MRP data files for the target objects.
2. Display the target attribute in edit mode and allow the developer to modify the attribute value.
3. Search the class files of the MRP system for the target class to which the object belongs.
4. Validate that the attribute value provided by the developer is a valid one (not less or greater that the target attribute domain defined in the class prototype).
5. If valid attribute value then save the object in the corresponding data object file else

display an error message such as: "attribute value does not match attribute domain".

changeClassMembership

```
1. Search the class files of the MRP system for the target "migrate-to-class" prototype.
2. Search the class files of the MRP system for the superclasses of the target "migrate to
   class".
3. For each class in step 1 and 2
       Save the attributes in a temporary holding area.
       Search the MRP data files for the objects under each class in step 1 and 2.
       For each object:
           call addAttributesToObject
           Save the object.
```

assembleChunk

1. Search the MRP data files for the target chunk prototype.
2. Access the parameters list of the target chunk to identify the number of objects required of each class.
3. Search the MRP data files for the for the required objects.
4. Store the ID's of the target objects in a temporary area to make it available to the developer.

deleteChunk

1. Search the MRP data files for the target chunk prototype.
2. Delete the target chunk.

5.2.5. Maintenance Module

maintDriver

1. Display a screen showing the following maintenance options:

    Creating a class, Deleting a class, Changing a superclass, Adding a relationship to a class, Deleting a relationship from a class, Adding a method to a class, Deleting a method from a class, Modifying a method implementation, Adding an attribute to a class, Deleting an attribute from a class, Changing an attribute domain, Changing an attribute default value, Creating an object, Deleting an object, Creating a relationship between two objects, Removing a relationship between two objects, Changing the value of an object attribute, Changing an object-class membership, Creating a chunk, Deleting a chunk, Assembling a chunk.

```
2. If maintOption is Creating a class then
            call enterClass
            call validateHierarchy
            If valid hierarchy then call storeClass
                    call configDriver
3. If maintOption is Deleting a class then
            call enterClass
            call updateDeletingClass
            call configDriver
            call propagateClassDeletion
4. If maintOption is Changing a superclass then
            call enterClass
```

```
        call editClass
        call validateHierarchy
        If valid hierarchy call updateChangingSuperclass
            call configDriver
            call propagateChangingSuperclass
    5. If maintOption is Adding a relationship to a class then
        call enterClass
        call designAnAttribute
        call validateRelationship
        If valid relationship name then call storeRelationship
            call configDriver
            call modifyObjects
    6. If maintOption is Deleting a relationship from a class then
        call enterClass
        call editClass
        call updateDeletingRelationship
        call configDriver
        call modifyObjects
    7. If maintOption is Adding a method to a class then
        call enterClass
        call designAMethod
        call validateMethod
        If valid method name call storeMethod
        call configDriver
8. If maintOption is deleting a method from a class then
        call enterClass
        call editClass
        call updateDeletingMethod
        call updateMethodTableFile
        call configDriver
9. If maintOption is Modifying a method implementation then
        call enterClass
        call editClass
        call editMethod
        call storeMethod
10. If maintOption is Adding an attribute to a class then
        call enterClass
        call designAnAttribute
        call validateAttribute
        If valid attribute name call storeAttribute
            call updateMethodTableFile
            call configDriver
            call modifyObjects
11. If maintOption is Deleting an attribute from a class then
        call enterClass
        call editClass
        call displayMethods
        call updateDeletingAttribute
        call updateMethodTableFile
        call configDriver
        call modifyObjects
    12. If maintOption is changing an attribute domain then
        call enterClass
        call editClass
        call editAttribute
        call updateChangingAttrDomain
        call changeAttributeDomain
    13. If maintOption is changing an attribute default value then
        call enterClass
        call editClass
        call editAttribute
        call updateChangingAttrDefault
        call changeAttributeDefault
    14. If maintOption is Creating an object then
        call enterClass
        call enterObject
        call createObject
    15. If maintOption is Deleting an object then
        call enterClass
        call enterObject
        call deleteObject
16. If maintOption is Creating a relationship between two objects then
        call enterTwoObjects
        call createObjectsRelationship
17. If maintOption is Deleting a relationship between two objects then
        call enterTwoObjects
        call deleteObjectsRelationship
18. If maintOption is Changing the value of an attribute then
        call enterClass
        call enterObject
        call changeObjectAttributeValue
19. If maintOption is Changing an object-class membership then
        call enterClass
        call enterObject
        call changeClassMembership
20. If maintOption is Creating a chunk  then
        call enterChunk
        call storeChunk
        call  configDriver
21. If maintOption is Deleting a chunk then
        call enterChunk
        call updateDeletingChunk
```

```
            call deleteChunk
22. If maintOption is Assembling a chunk then
            call enterChunk
            call assembleChunk
```

5.3. <u>Object Oriented Classes of the OAS</u>

As indicated in section 4.2 of chapter 4, classes of the OAS can be arranged into two categories:

1. The OAS Application Kit, or the Object Library classes such as: Integer, or InventoryItem.
2. The functional classes which are utilized to manipulate the Application Kit classes.

In order to identify the potential functional classes of the OAS, the functional building blocks have to be grouped based on similarity of their functions. Observing the function of each building block in the detailed architecture of the OAS, the building blocks can be categorized into the following functional groups:

1. Coordination
2. Maintenance
3. Display and Edit
4. Validation
5. Files Updating
6. Configuring
7. Data Management

The functional groups and their corresponding building blocks are shown in Figure 5.2 below.

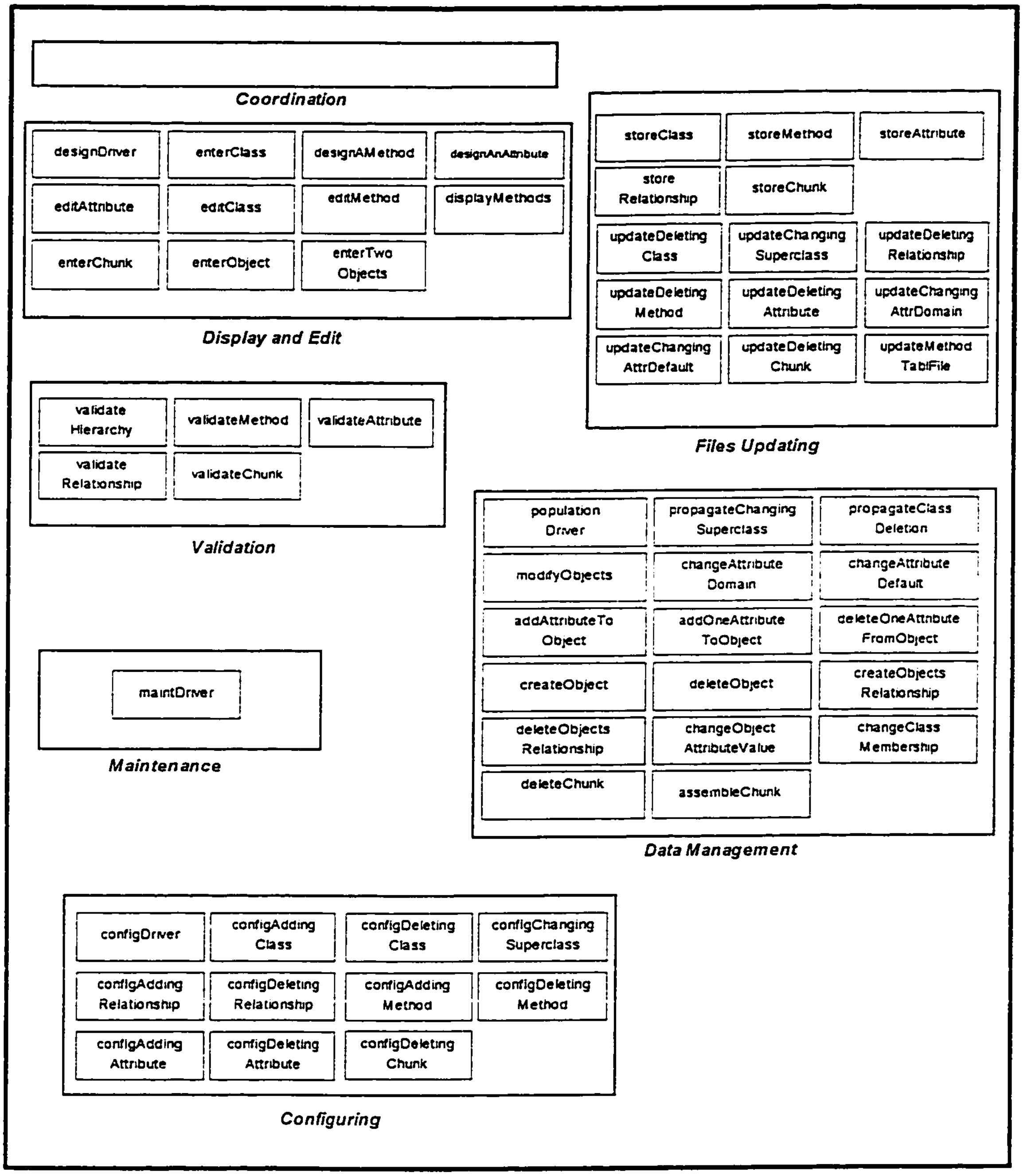


Figure 5.2, Functional Grouping

Based on a functional grouping, the following are potential candidate functional classes:

1. MainDriver

2. DesignMgr

3. ValidationMgr

4. ConfigurationMgr

5. PopulationMgr

6. MaintenanceMgr

Figure 5.3 below shows the functional candidate classes with their corresponding building blocks.

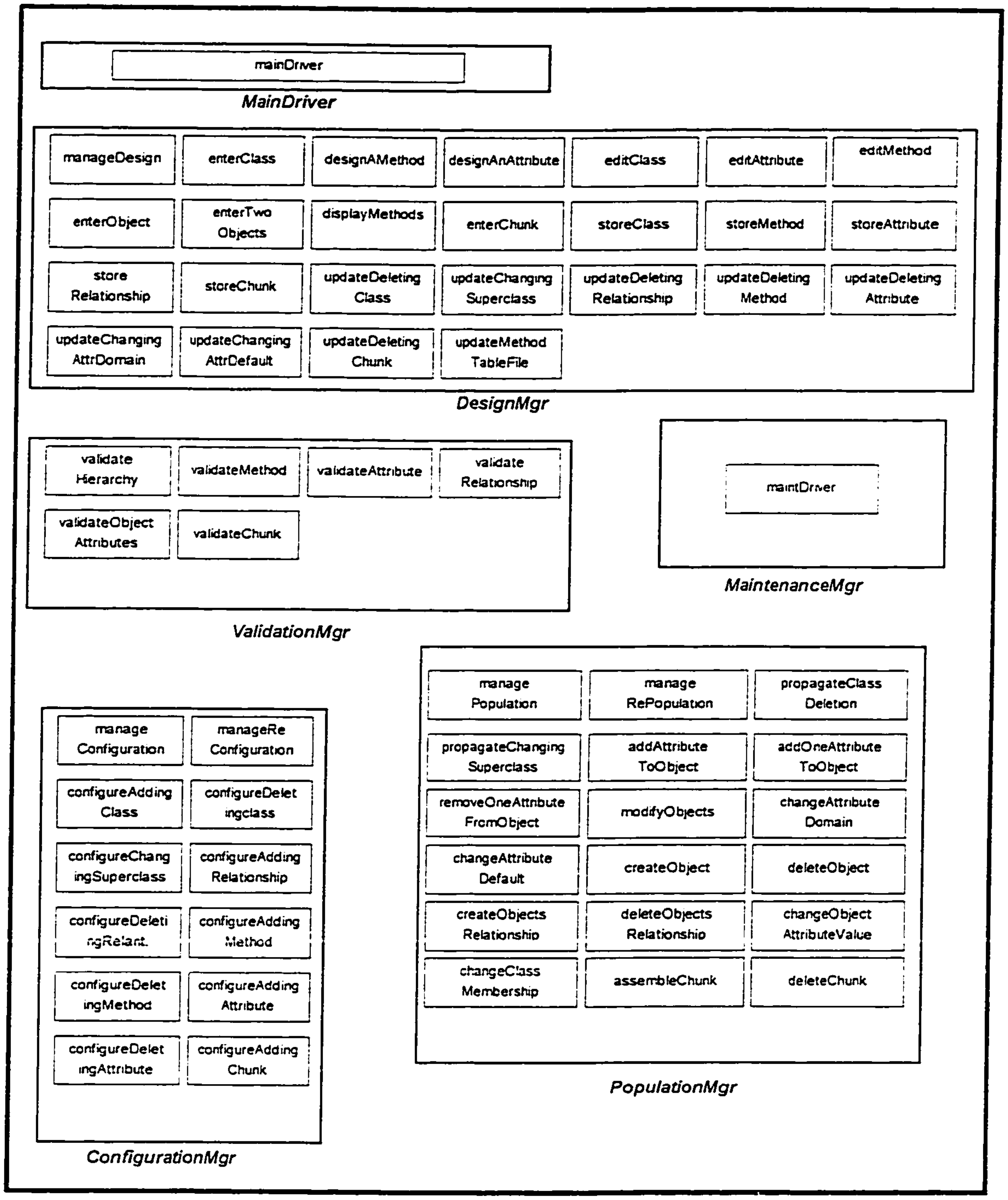


Figure 5.3, Functional class of the OAS

Figure 5.4 below shows the full view of the Object Oriented class hierarchy of the OAS including its Application Kit as well as the functional classes.

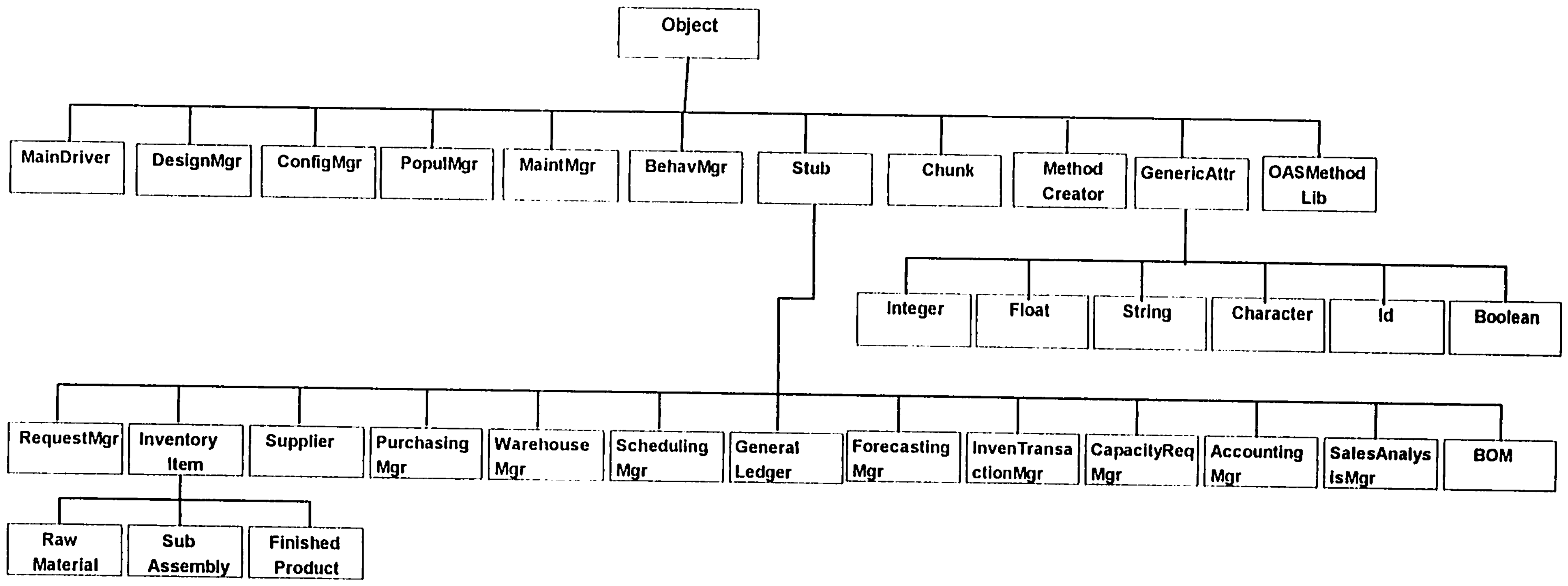


Figure 5.4, the O.O. Class Hierarchy of the OAS

5.4. Detailed Operating Procedures of the OAS

This section presents the steps required to operate the OAS and invoke its various functions. However, in this research, the analysis phase is not considered to be a key factor in providing flexibility to software. Accordingly, this research is not introducing any new concepts or suggestions to the analysis and will not discuss this phase. All the other phases will be explained in detail in the next four subsections, 5.4.1 through 5.4.4.

System operation, as shown in Figure 5.5, starts with selecting an option from the OAS Main Menu such as design, implementation, population, or maintenance.

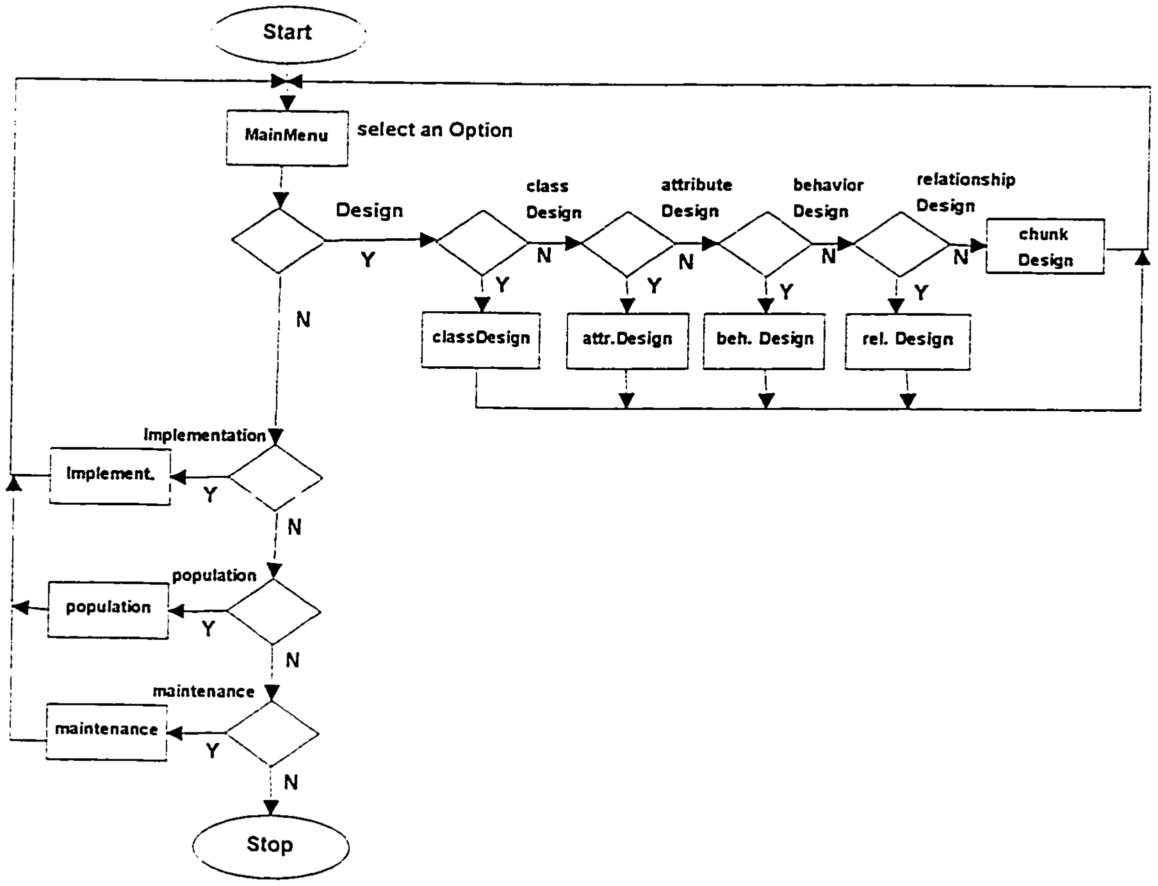


Figure 5.5, Operating Procedure of the OAS

5.4.1. Design

Upon selecting the Design option from the Main Menu, another menu will be displayed. The developer will be given the choice to select: Class Core design, Behavior design, Attribute design, Relationship design, or Chunk design. The processes performed in each option will be discussed below:

Class Core Design

If the developer selects the class core design option, then:

1. The *enterClass* submodule will display a user-interface similar to Figure 5.6 where existing classes in both the Class Specs file and the Object Library are shown. The developer will be prompted to provide a class name, and select a superclass from the classes displayed on the screen.

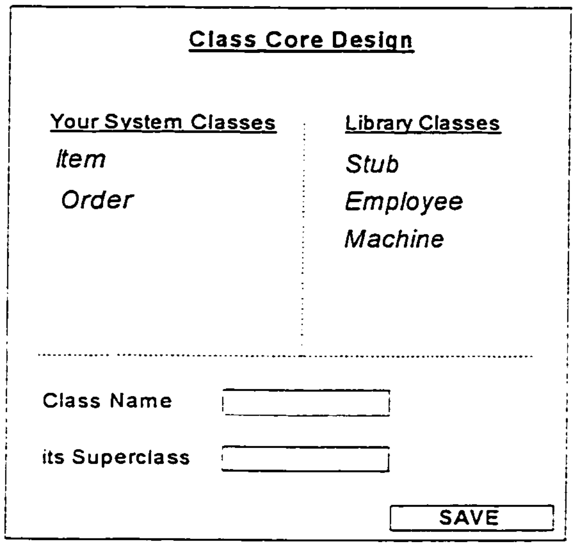


Figure 5.6, Class Core Design user-interface

2. Upon selecting the “save” option, the *validateHierarchy* submodule will be triggered to validate whether the superclass name provided by the developer results in a valid hierarchical relationship.

3. If the hierarchy is valid, the name of the target class, along with its superclass will be stored in the Class Specs file by calling the *storeClass* submodule. Else an

error message (Invalid class/superclass relationship) will be displayed and the developer will prompted to provide a new superclass..

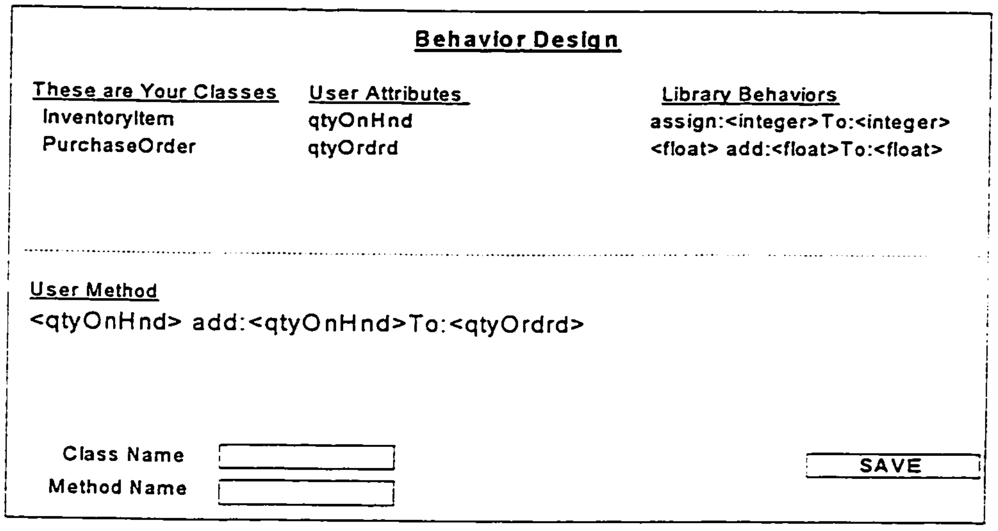
Behavior Design

If the developer selects the class behavior design option, then the following procedures will be performed:

1. The *enterClass* submodule will display a screen similar to Figure 5.7, where classes from the Class Specs file are shown and the developer will be allowed to select a class.
2. Upon selecting a class, the *designAMethod* submodule will display the name of the selected class and a list of its attribute names as well as a list of all the behavior building blocks of the Object Library in the format shown in the following example:

   <float> add: <float> To: <float>

   The line above shows a low-level behavior building block, from the Pre-Compiled Object Library, called "**add**" which requires two arguments of float data type and returns a float.
3. The developer will be allowed to drag & drop any number of blocks, supply the required arguments and assemble the blocks in the sequence which accomplishes the desired behavior.
4. Next, the developer will be prompted to provide a method name.
5. Upon selecting the "save" option, the *validateMethod* will be invoked to validate whether the target method name exists in the target class.


Behavior Design

These are Your Classes	User Attributes	Library Behaviors
InventoryItem	qtyOnHnd	assign:<integer>To:<integer>
PurchaseOrder	qtyOrdrd	<float> add:<float>To:<float>

User Method
<qtyOnHnd> add:<qtyOnHnd>To:<qtyOrdrd>

Figure 5.7, Behavior Design user-interface

6. If the method name does not exist in the target class, the *storeMethod* will store the method in both the Class Specs file and Attribute-Method table file, else an error message, such as “Method name already exists in the class”, will be displayed and the developer will be prompted to provide a new method name.

Attribute Design

If the developer selects the class attribute design option, then:

1. The *enterClass* submodule will display a screen similar to the one shown in Figure 5.8, where classes from the Class Specs file are shown and the developer is prompted to select a class name.
2. Upon selecting a target class, the *designAnAttribute* submodule will display the name of the selected class and a number of icons, to represent the various data types available in the Object Library such as: integer, double, string, or id. The developer will be able to select a data type then provide a name, default value, and domain when applicable.

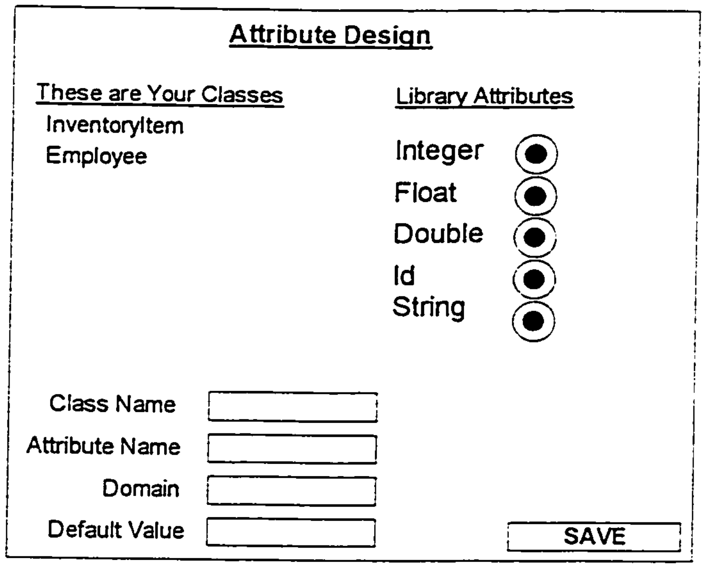


Figure 5.8 Attribute Design user-interface

3. Upon choosing the "save" option, the *validateAttribute* will be invoked to validate whether the target attribute name exists in the target class.

4. If attribute name does not exist in the target class, it will be stored in both the Class Specs file and Attribute-Method table file by calling the *storeAttribute* submodule. Else an error message (Attribute name already exists in the class) and the developer will be prompted to provide a new attribute name.

Relationship Design

If the developer selects the class relationship design option, then these procedures will be performed:

1. The *enterClass* submodule will display a screen similar to Figure 5.8, where classes from the Class Specs file are shown, and the developer is prompted to select a class name.

2. Upon selecting a target class, the *designAnAttribut* submodule will display the name of the selected class and a number of icons to represent the various data types available

in the Object Library. The developer will then be prompted to select the relationship data type (**id**) and provide a name for it.

3. Upon selecting the "save" option selection, the *validateRelationship* will be invoked to check whether or not the target relationship name exists in the target class.
4. If the relationship name does not exist in the target class, the target relationship will be stored in the target class in the Class Specs file by calling the *storeRelationship* submodule else an error message(Relationship name already exists in the class) will be displayed.

Chunk Design

When the developer selects the chunk design option, then:

1. The *enterChunk* submodule will display a user-interface similar to Figure 5.9 where all chunk names in the Chunk Specs file will be shown. The developer is prompted to enter a chunk-name along with its parameter-list.
2. Upon selecting the "save" option, the *storeChunk* submodule will store the chunk in the Chunk Specs file.

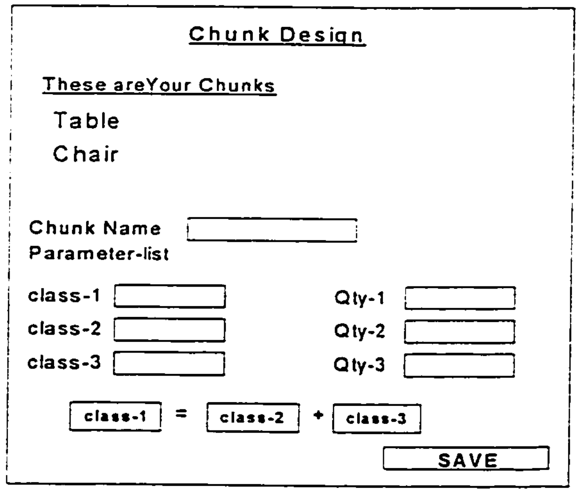


Figure 5.9 Chunk Design user-interface

5.4.2. Implementation

When selecting the Implementation option from the Main Menu, the Coordination Manager gives control to the *configDriver* in order to generate the class and chunk prototypes. The *configDriver* will perform the following processes, for each class and chunk in the Class Specs file and the Chunk Specs file respectively, to accomplish its task:

1. For each class defined in the Class Specs file:

   1.1. The *configAddingClass* submodule will search the Pre-Compiled Object Library for Class Stub ,or the superclass of the target class, and create a subclass of it. This newly created class will be the target class prototype. The *configAddingClass* will then will store the ID of the superclass in the target class prototype.

   1.2. For each method defined in the target class, the *configAddingMethod* submodule will search the Pre-Compiled Object for class MethodCreator. Next it will create an object from MethodCreator and use it as a container to store the building blocks of the target method (from the Class Specs file). The ID of the container object which represent the target method will then be stored in the target class prototype.

   1.3. For each attribute defined in the target class, the *configAddingAttribute* submodule searches the Object Library for the attribute whose data type matches the target attribute definition and creates an object of it. The ID of the target attribute object will then be stored in the target class prototype.

   1.4. For each relationship defined in the target class, the *configAdding Relationship* submodule will search the Object Library for the relationship attribute "**id**" and create an object representing the target relationship then store its ID in the target class prototype.

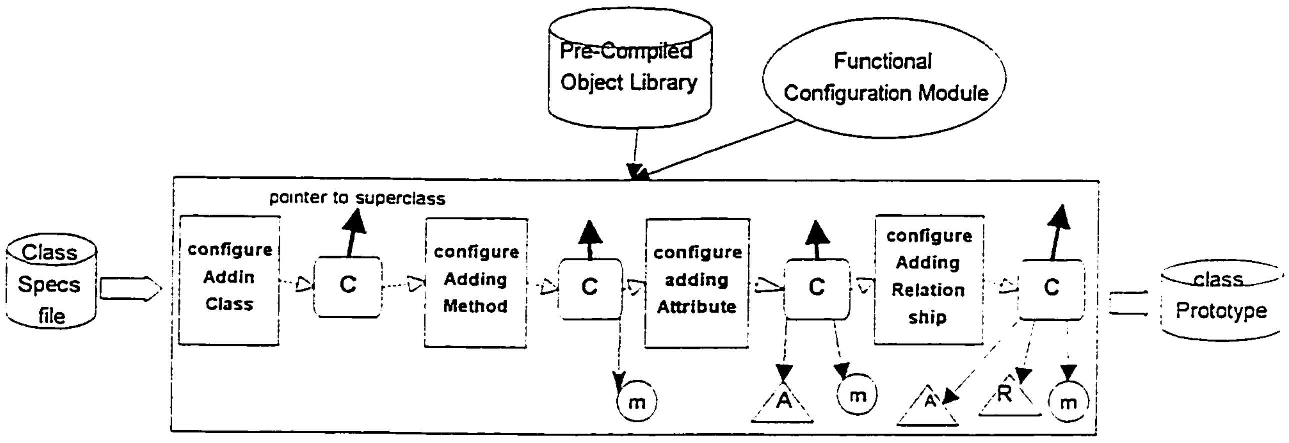

Pre-Compiled Object Library
Functional Configuration Module
pointer to superclass
Class Specs file
configure Addin Class
C
configure Adding Method
C
m
configure adding Attribute
C
A
m
configure Adding Relation ship
C
A
R
m
class Prototype

Figure 5.10 The Configuration Process

Figure 5.10 shows a schematic representation of the sequence of events performed in order to assemble a class during configuration where m stands for method, c for class and A for an attribute.

2. For each chunk defined in the Chunk Specs file, the *configAddingChunk* submodule will search the Object Library for the for class Chunk and creates its proxy, then store the chunk parameters in it.

5.4.3. Data Population

When selecting the data population option from the Main Menu, the Coordination Manager will invoke the *populationDriver* submodule in order to populate the MRP system with data. A menu will be displayed with options to: create an object, or assemble a chunk. The *populationDriver* will perform the following procedures to accomplish the various tasks for each option.

1. If the developer selects creating an object then the *enterObject* submodule will be called to display all class names in the MRP system and allow the developer to select

one. The *enterObject* will then display attribute names of the target class and prompt the developer to provide an object name and values for its attributes. Upon "save" option, the *createObject* submodule will create and save the desired object.

2. To assemble a chunk from existing objects the *populationDriver* would call *display Chunks* submodule to display a list of the chunk names in the Chunk Specs file and allow the developer to select a target chunk. Next, the *assembleChunk* submodule will be invoked to identify the required number of objects from the right classes based on the chunk Parameter-list specified in the Chunk Specs file. The ID's of these objects will then be available to the developer to take the desired action.

5.4.4. Maintenance

When selecting the maintenance option from the Main Menu, the Coordination Driver gives control to the *maintDiver* submodule where a menu will be displayed allowing the developer to choose any of the options listed in Table 5.1 below:

Maintenance Type
Creating a New Class
Deleting a Class
Changing a Superclass
Adding a Method to a Class
Deleting a Method from a Class
Modifying a Method Implementation
Adding an Attribute to a Class
Deleting an Attribute from a Class
Changing an Attribute Domain
Changing an Attribute Default Value
Adding a Relationship to a Class
Deleting a Relationship from a Class
Creating a Chunk
Deleting a Chunk
Creating an Object
Deleting an Object

Creating a Relationship between two objects
Removing a Relationship between two Objects
Changing the value of an Object Attribute
Changing an Object-Class Membership
Assembling a Chunk

Table 5.1 Maintenance types

Some of the changes listed in Table 5.1, require re-design and re-implementation, some require re-design, re-implementation, and re-population while others require population only. Table 5.2 below, categorizes the maintenance types by the required action.

Maintenance Type	**Action(s) Required**
Creating an Object	These changes require **Re-Population** only
Deleting an Object	
Creating a Relationship between two objects	
Removing a Relationship between two Objects	
Changing the value of an Object Attribute	
Changing an Object-Class Membership	
Assembling a Chunk	
Changing an Attribute Domain	These changes require **Re-Design** and **Re-Population**
Changing an Attribute Default Value	
Deleting a Chunk	
Creating a New Class	These changes require **Re-Design** and **Re-Implementation**
Adding a Method to a Class	
Deleting a Method from a Class	
Creating a Chunk	
Deleting a Class	These changes require **Re-Design**, **Re-Implementation**, and **Re-Population**
Changing a Superclass	
Adding an Attribute to a Class	
Deleting an Attribute from a Class	
Adding a Relationship to a Class	
Deleting a Relationship from a Class	

Table 5.2, Maintenance types sorted by action required

It is very important to emphasize that these changes will be immediately executed in a real-time fashion as they initiated by the developer. In other words, upon any change initiated by the developer, all the appropriate re-design, re-implementation, and re-population procedures will be transparently invoked without further developer intervention.

The following subsections will present , in details, all the processes performed for each maintenance option in the sequence used in Table 5.1.

Creating a New Class

1. The *enterClass* submodule will display a screen where classes from both the Class Specs file and the Object Library are shown and the developer is prompted to provide a class name and select a superclass.
2. Upon selecting a "save" option the *validateHierarchy* will validate if the developer provided a valid hierarchy.
3. If valid hierarchy then *storeClass* submodule will be update the Class Specs file by adding the target class to it else display an error message "invalid superclass"..
4. The *configAddingClass* submodule will be called to create the target class prototype.

Deleting a Class

1. The *enterClass* submodule will display a screen where classes from the Class Specs file are shown and the developer is prompted to select a class name.
2. Next, the developer will be prompted to confirm deleting the target class. If deletion is confirmed, then the developer will be asked to specify whether or not to delete all the objects of the target class or migrate them to another class.
3. If migration is selected then the developer will be asked to provide the "migrate-to-class".

4. Upon selecting a "delete" option the *updateDeletingClass* submodule will update the Class Specs file by deleting the target class from it.
5. Next, the *configDeletingClass* will be called to remove the target class and re-align the hierarchical relationships.
6. The *propagateClassDeletion* will be invoked to propagate the appropriate changes.

<u>Changing a Superclass</u>

1. The *enterClass* submodule will display a screen where classes from the Class Specs file are shown and the developer is prompted to select a class name.
2. Upon selecting a class name the *editClass* submodule will be called and the class superclass will be displayed on the screen in edit mode and the developer would be allowed to modify it.
3. The validity of the new hierarchy will then be checked by calling the *validate Hierarchy* submodule. If valid then *updateChangingSuperclass* will change the inheritance relationship of the target classes in the Class Specs file. Else display an error message "invalid superclass" and the developer will be prompted to provide another superclass.
4. The *configChangingSuperclass* will be invoked next..
5. The change will be propagated to the affected objects by calling the *propagateChangingSuperclass.*

<u>Adding a Method to a Class</u>

1. The *enterClass* submodule will display a screen where the classes from the Class Specs file are shown and the developer is prompted to select a class name.
2. Upon selecting a target class, all the attribute names of the target class will be shown

and the *designAMethod* submodule will display all the behavior building blocks of the Object Library.

3. The developer will be allowed to drag & drop any number of blocks, supply the required attributes and assemble the blocks in the sequence which accomplishes the desired behavior.
4. Next the *designAMethod* submodule will prompt the developer to provide a method name.
5. Upon selecting a "save" option, the *validateMethod* submodule will validate if the target method name exists in the target class.
6. If method name does not exists in the target class, the *storeMethod* submodule will add the target method to the target class in the Class Specs file and the Attribute-Method table file. Else a "Duplicate method name" message will be displayed and the developer will be prompted to provide a new method name.
7. The *configAddingMethod* submodule will be invoked to create an object containing the method building blocks and store its ID in the target class prototype.

Modifying a Method Implementation

1. The *enterClass* submodule will display a screen where the classes from the Class Specs file are shown, and the developer is prompted to select a class name.
2. Upon selecting a target class, the *editClass* submodule will be called to display a list of all the method names in the target class and allow the developer to select a target method.
3. Upon selecting a target method, the *editMethod* submodule will display all the building blocks of the Object Library, the attribute names of the target class as

well as the building blocks of the target method allowing the developer to (a) add or delete a building block, (b) modify the arguments of a building block.

4. Upon selecting a "save" option, the *storeMethod* submodule will save the target method in the target class in the Class Specs file and the Attribute-Method table file.

Deleting a Method from a Class

1. The *enterClass* submodule will display a screen where the classes from the Class Specs file are shown and the developer is prompted to select a class name
2. Upon selecting a target class, the *editClass* submodule will be called to display a list of all the method names in the target class and allow the developer to select a target method for deletion.
3. Upon confirming the deletion, the *updateDeletingMethod* and *Update-Method-TableFile* submodules will delete the target method from both the Class Specs file and Attribute-Method table file.
4. The *configDeletingMethod* submodule will remove the ID of the target object method from the class prototype.

Adding an Attribute to a Class

1. The *enterClass* submodule will display a screen where classes from the Class Specs file are shown and the developer is prompted to select a class name.
2. Upon selecting a target class, the *designAnAttribute* submodule will display a screen where all the attribute names of the Object Library will be listed and the developer will be allowed to select an attribute and provide a name, a domain and default value for it.
3. The *validateAttribute* will validate if the target attribute name exists in the target class.

4. If attribute name does not exist in the target class, the *storeAttribute* and the *updateMethodTableFile* submodules will add the target attribute to both the Class Specs file and Attribute-Method table file respectively else an error message “Duplicate attribute name” will be displayed and the developer will be prompted to provide another attribute name..
5. The *configAddingAttribute* submodule will be called to create the required attribute object and store its ID in the target class prototype.
6. Finally, the *modifyObjects* will be invoked to propagate the appropriate changes to the affected objects.

<u>Deleting an Attribute from a Class</u>

1. The *enterClass* submodule will display a screen where classes from the Class Specs file are shown and the developer is prompted to select a class name.
2. Upon selecting a target class, the *editClass* submodule will be called to display a list of all the attribute names in the target class and allow the developer to select a target attribute.
3. The target attribute will be used as a look-up key in the Attribute-Method table file to display all the method names which implement the target attribute by calling the *displayMethods* submodule. If there exists a method which implements the target attribute then the session will be terminated with a proper message, else the *updateDeletingAttribute* and *updateMethodTableFile* submodules will delete the target attribute from the Class Specs file and the Attribute-Method table file respectively.
4. The *configDeletingAttribute* submodule will be called to remove the ID of the target attribute object from the class prototype.

5. The *modifyObjects* will be invoked to propagate the appropriate changes to the affected objects.

Changing an Attribute Domain

1. The *enterClass* submodule will display a screen where classes from the Class Specs file are shown and the developer is prompted to select a class name
2. Upon selecting a target class, the *editClass* submodule will be called to display a list of all the attributes names in the target class and allow the developer to select a target attribute.
3. The *editAttribute* submodule will display the target attribute, its data type, domain and default value in edit mode and allow the developer to modify the attribute domain. The developer will then be prompted to provide an "*algorithm*", if needed, in order to change the attribute value in the existing objects . The algorithm will have the following format:

   change factor, an operator (such as * or +)
4. The *updateChangingAttrDomain* will change the target attribute domain in the Class Specs file.
5. The *changeAttributeDomain* submodule will then modify the domain of the target attribute in all the objects of the target class using the "*algorithm*" provided by the developer.

Changing an Attribute Default Value

1. The *enterClass* submodule will display a screen where the classes from the Class Specs file are shown and the developer is prompted to select a class name.
2. Upon selecting a target class, the *editClass* submodule will be called to display a list of

all the attributes names in the target class and allow the developer to select a target attribute.

3. The *editAttribute* submodule will display the target attribute, its data type, domain and default value in edit mode and allow the developer to modify the attribute default value.
4. The *updateChangingAttrDefault* will change the target attribute default value in the Class Specs file.
5. Next, the *changeAttributeDefault* submodule will modify the default of the target attribute in all the objects of the target class.

Adding a Relationship to a Class

1. The *enterClass* submodule will display a screen where the classes from the Class Specs file are shown and the developer is prompted to select a class name.
2. Upon selecting a target class, the *designAnAttribute* submodule will display all the attributes in the Object Library and prompt the developer to select the relationship attribute.
3. The *validateRelationship* submodule will be invoked to validate if the target relationship does not exist in the target class.
4. If relationship name does not exist in the target class, the *storeRelationship* submodule will add the target relationship to the target class in the Class Specs file. Else an error message "duplicate relationship name" will be displayed and the developer will be prompted to provide another relationship name.
5. The *configAddingRelationship* submodule will create the required relationship object and store its ID in the target class prototype.

6. The *modifyObjects* will then be invoked to propagate the appropriate changes to the affected objects.

Deleting a Relationship from a Class

1. The *enterClass* submodule will display a screen where the classes from the Class Specs file are shown, and the developer is prompted to select a class name.
2. Upon selecting a target class, the *editClass* submodule will be called to display a list of all the relationship names in the target class and allow the developer to select a target relationship.
3. The *updateDeletingRelationship* will the be invoked to delete the target relationship from the target class in the Class Specs file.
4. The *configDeletingRelationship* will be invoked to remove the ID of the target relationship object from the target class prototype.
5. Finally, the *modifyObjects* will be invoked to propagate the appropriate changes to the affected objects.

Creating an Object

1. The *enterClass* submodule will display a screen where the class names from the Class Specs file are shown, and the developer is prompted to select a class name.
2. Upon selecting a target class, the *enterObject* submodule will display the attribute names of the target class and prompt the developer to enter an object name and values for its attributes.
3. Upon "save" option the target object will be created by the *createObject* submodule and saved in a data object file.

Deleting an Object

1. The *enterClass* submodule will display a screen where classes from the Class Specs file are shown, and the developer is prompted to select a class name.
2. Upon selecting a target class, the *enterObject* submodule will display the object names of the target class and the developer will be prompted to select an object name for deletion.
3. Upon the developer acceptance, the *deleteObject* submodule will delete the target object from its corresponding data object file.

Creating a Relationship between two Objects

1. The *enterTwoObjects* submodule will display all class names and allow the developer to select one or two class names, then display their corresponding object names.
2. The *createObjectsRelationship* will prompt the developer to select any two object names.
3. Upon a "save" option, the ID of the second object along with the relationship name will be stored in the first object.

Removing a Relationship between two Objects

1. The *enterTwoObjects* submodule will display all class names and allow the developer to select one or two class names, then show their corresponding objects names.
2. The *deleteObjectsRelationship* will prompt the developer to select any two object names.
3. Upon a "delete" option the ID of the second object along with the relationship name will be removed from the first object.

Changing the Value of an Object Attribute

1. The *enterClass* submodule will display a screen where classes from the Class Specs file are shown and the developer is prompted to select a class name.
2. Upon selecting a target class, the *enterObject* submodule will display the object names of the target class and the developer will be prompted to select an object name and a target attribute for change.
3. The *changeObjectAttributeValue* will display the target attribute and allow the developer to modify the value of the target attribute and save the target object in its data object.

Changing an Object-Class Membership

1. The *enterClass* submodule will display a screen where classes from the Class Specs file are shown and the developer is prompted to select a class name.
2. Upon selecting a target class, the *enterObject* submodule will display the object names of the target class and the developer will be prompted to select an object name and select its new class (migrate-to-class).
3. The *changeClassMembership* will modify the object structure to make it compatible with its new class.

Creating a Chunk

1. The *enterChunk* submodule will display a list of all the chunk names in the Chunk Specs file and the developer is prompted to enter a chunk name along with its parameter-list.
2. Upon selecting a "save" option, the *storeChunk* will store the target chunk in the Chunk Specs file.

3. Next, the *configAddingChunk* submodule will create an object representing the target chunk and store in it the chunk parameter-list..

Deleting a Chunk

1. The *enterChunk* submodule will display a list of all the chunk names in the Chunk Specs file and the developer is prompted to select one.
2. Upon selecting a chunk name and confirming its deletion, the *updateDeletingChunk* will delete the target chunk from the Chunk Specs file.
3. The *deleteChunk* submodule will then delete the target chunk.

Assembling a Chunk

1. The *enterChunk* submodule will display all the chunk names in the Chunk Specs file and prompt the developer to select one.
2. Upon selecting a target chunk name, the *assembleChunk* submodule will assemble the target chunk using existing object.

Maintenance of the Object Library

One of the important features of OAS is the capability for expanding its Object Library. As shown in Figure 5.11 below, the Library could be easily maintained by adding new behavior building blocks, classes, or attribute building blocks. The input will be the old library along with the new additions.

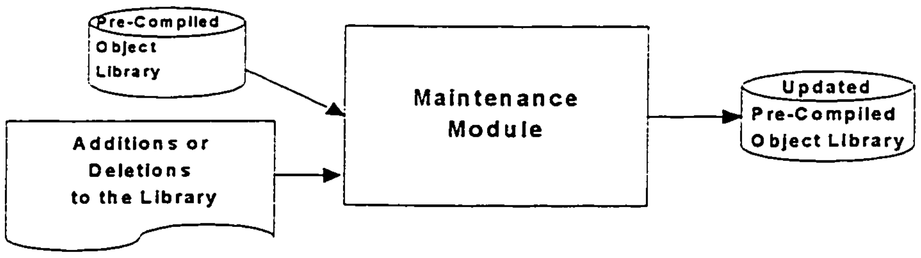

Pre-Compiled Object Library
Additions or Deletions to the Library
Maintenance Module
Updated Pre-Compiled Object Library

Figure 5.11 Maintenance of the Object Library

The Object Library can be maintained by developing the required module using Objective C source code going through the cycle of edit-compile-link-test before appending the target module to the library.

Chapter 6

RESEARCH CONTRIBUTION AND FUTURE RESEARCH

6.1. The Contribution of the Research

The research is building on existing research such as software reuse, chunking [Lim 1996], and CASE, in an object-oriented framework in order to develop and maintain manufacturing software systems. The research, in this regard, is a software development tool capitalizing on existing ideas and approaches. However, the research will have the following contribution to the existing body of knowledge:

- Taking the software reuse approach one step further by introducing the reuse of pre-compiled objects with higher level of granularity such as *integer*, *float* and *id*. The OAS also reuses specialized pre-compiled objects designed to meet the MRP needs.
- Utilizing the idea of chunking in manufacturing software system.
- Providing another alternative to the traditional approach of [edit-compile-link] to software development and maintenance such as CASE. This alternative of assembling and packaging pre-compiled objects, as has been demonstrated in the case study, requires a minimal effort and software development skills.

The research will have the following benefits in the area of manufacturing software engineering:

1. Ease of development and maintenance of highly flexible manufacturing planning software systems due to its capability to assemble and de-assemble classes from pre-compiled building blocks.
2. Use of chunking and MRP-oriented application kit of specialized classes.

3. Capability, as an Object-Oriented. run-time application building tool, to reduce the effort and cost of software maintenance where all changes would be carried out at run-time.
4. Automated propagation of all changes to the underlying data objects at run-time.

The OAS is a good tool for manufacturing software due to the following requirements and features that distinguishes manufacturing from other domains:

1. Frequent changes due to continuous improvements, new product developments and subsequent changes in operations.
2. The high cost of maintenance.
3. The need for run-time maintenance than compile-time in many cases.

6.2. Future Research

The research could be extended to:

1. Develop a set of variables to quantify software flexibly. As a general guideline, ease and cost of maintenance are good tools to measure software flexibility. Following are some variables which could be used to measure the ease and cost of maintenance:

   (a) The time required to carry out the following processes:

   - Locating where the change should be introduced in the target software.
   - Developing a code or any other mean required to accomplish the change.
   - Processing and executing change.
   - Propagating the change to the persistent data.

   (b) Skills needed to carry out the change.

   (c) Complexity of the process of locating, developing, and executing the change.

Ultimately, all these times, skills, and complexity could then be weighted and factored as cost.

2. Develop a real-life MRP case study and use the variables listed above in order to bench-mark the flexibility of the OAS versus any other comparable software maintenance tool.
3. Develop software systems for domains other than manufacturing planning domain. A good candidate domain would be characterized by:
    - High frequency of change.
    - Run-time maintenance requirement.
    - Wide ripple effect upon introducing a change.
    - Larger volume of data.

   Applying the OAS for other domains would be accomplished by expanding its Pre-Compiled Object Library through:

   (a) adding other attribute building blocks.

   (b) adding specialized behavior building blocks to the OASMethodLib to meet the requirements of the target domain.

   (c) adding more specialized classes to the library.

   Any addition to the library has to start with initiating the proper Objective-C code, compiling and saving into the library.
4. Implementing the technique of multiple inheritance to further extend its flexibility by extending the structure of the root class **Stub** so that it would have a list of pointers instead of one to its superclass. This could be accomplished by modifying the Objective-C underlying code of class Stub to reflect Superclass-id-list and superclass-name-list instead of superclass-id and superclass-name respectively. Accordingly, all the naming conflicts associated with multiple inheritance would then have to be

resolved by adopting a proper approach from the existing research body.

5. Incorporating the concepts of fuzzy logic to make the OAS applicable for even wider range of domains. According to [George et. al. 1993] a fuzzy object-oriented system is characterized by:

(a) uncertainty in the values of an object's attributes.

(b) uncertainty in the class-superclass relationship.

In other words, the value of an object attribute equals to a certain value in a certain degree of membership, similarly, a class is a subclass of another class in a certain degree of membership. Therefore, to integrate fuzziness in the OAS structure, two steps would be required:

(a) To accomplish fuzziness in an object's attribute value, another variable must be added to the definition of an attribute building block (in addition to name, value, domain, and default value), this variable would be an integer of float to hold the degree of membership.

(b) To incorporate fuzziness in the class-superclass inheritance relationship, each pointer pointing to a superclass must be associated with an integer or float variable to hold the degree of inheritance relationship.

REFERENCES

A

[Aggarwal et al. 1995] R. Aggarwal and J. Lee, CASE and TQM for Flexible Systems Information Systems Management, Fall 1995, pp. 15-19.

[Ahlsen et al. 1983] M. Ahlsen, A. Bjornerstedt, S. Britts, C. Hulten, L. Soderlund, *Making Type Changes Transparent*, Proceedings of IEEE workshop on Languages for Automation, Nov. 1983, pp. 110-117.

[Ahlsen et al. 1984] M. Ahlsen, A. Bjornerstedt, S. Britts, C. Hulten, L. Soderlund, *An Architecture for Object Management in OIS*, Readings in object-oriented databases, edited by Stanley Zdonik and David Maier, Morgan Kaufmann publishing, 1990, pp. 570-581.

[Arnold et. al. 1997] Vincent Arnold, Rebecca Bosch, Eugene Dumstroff, Paula Helfrich, Timothy Hung, Verlyn Johnson, Ronald Persik, Paul Whidden, *IBM Business Framework: San Francisco Project Technical Overview*, IBM Systems Journal, vol. 36, No. 3, 1997, pp. 437-445

B

[Banerjee et al. 1987] Jay Banerjee, Hong-Tai Chou, J. F. Garza, Won Kim, Darrel Woelk, Nat Ballou, Hyoung-Joo Kim, *Data Model Issues for Object-Oriented Applications*, ACM Transactions on Office Information Systems, Vol. 5, No. 1, Jan. 1987, pp. 3-26.

[Banerjee et al. 1987] Jay Banerjee, Won Kim, Hyoung-Joo Kim, Henry Korth, *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*, Proceedings- ACM on Management of Data,, May 1987, Publ. by ACM pp 311-322.

[Barnes et. al. 1991] Bruce Barnes, Terry Bollinger, *Making Reuse Cost-Effective*, IEEE Software, January 1991, pp. 13-24.

[Bassett 1987] Paul G. Bassett, *Frame-Based S oftware Engineering*, IEEE Software, July 1987, pp. 9-16.

[Baxter 1992] Ira Baxter, *Design Maintenance Systems*, Communications of the ACM, Vol. 35, No. 4, April 1992, pp. 73-89.

[Benson et al. 1992] Dan Benson, Greg Zick, *Spatial and Symbolic Queries for 3-D Image Data*, SPIE vol. 1662 Image Storage and Retrieval Systems 1992, pp. 134-145.

[Berlin 1990] Lucy Berlin, *When Objects Collide, Experiences with Reusing Multiple Class Hierarchies*, OOPSLA ECOOP 90, Ottawa, Canada 21-25 Oct. 1990, ACM Press, pp. 181-193.

[Bertino et al. 1993] Elisa Bertino, Lorenzo Martino, *Object-Oriented Database Systems,* Addison-Wesley Publishing Company, 1993, pp. 23.

[Booch 1991] G. Booch, *Object-Oriented Design with Applications,* Benjamin Cummings Publishing Co., Redwood City, CA (1991).

[Borchardt et al. 1995] D. Borchardt, N. Ramarapu, and M. Frolick, CASE Tools as Catalyst for Reengineering, Information Systems Management, Fall 1995, pp. 20-25.

[Brown et al. 1994] A. Brown, D. Carney, E. Morris, D. Smith, P. Zarrella, Principles of CASE Tool Integration, Oxford University Press, 1994, pp. 12-15.

[Buchmann et al. 1991] A. P. Buchmann, R. S. Carrera, M. A. Vazquez-Galindo, *Handling Constraints and their Exceptions: An Attached Constraint Handler for Object-Oriented CAD Databases*. On Object-Oriented Database Systems, K. R. Dittrich, U. Dayal, A. P. Buchmann (Eds), Springer-Verlag, 1991, pp. 65-83.

[Burch 1992] John G. Burch, *System Analysis, Design, and Implementation,* Boyd & Fraser Publishing Company, 1992, pp. 608-609.

[Burton et. al. 1987] Bruce Burton, Rhonda Weink-Aragon, Stephen Baily, Kenneth Koehler, Lauren Mayes, *The Reusable Software Library,* IEEE Software, July 1987, pp. 25-33.

C

[Chamberlain et al. 1995] W. Chamberlain, and G. Thomas, *The Future of MRP II: Headed for Scrap Heap or Rising from the Ashes,* IIE Solutions, July 1995, pp.32-35.

[Chung et al. 1992] Yunkung Chung, Gary W. Fischer, *Illustration of object-oriented databases for the structure of a bill of materials*. Computers in Industry, No. 19, 1992, pp. 257-270.

[Church et al. 1995] Terry Church, Philip Matthews, *An Evaluation of Object-Oriented CASE Tools: The Newbridge Experience*. Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering, July 1995, Publ. by IEEE, pp. 4-9.

[Clamen 1994] Stewart Clamen, *Schema Evolution and Integration*, Distributed and Parallel Databases, No. 2, 1994, pp. 101-126.

[Coleman et al. 1994] D. Coleman, D. Ash, B. Lowther, and P. Oman, Using Metrics to Evaluate Software System Maintainability, Computer No. 27, Aug. 94, pp. 44-49.

[Cox et al. 1991] Brad Cox, Andrew Novobilski, *Object-Oriented Programming, An Evolutionary Approach,* Addison-Wesley Publishing Co., 2nd edition, 1991, pp. 27.

D

[de Cima et. al. 1994] Alberto de Cima, Claudia Werner, Alessandro Cerqueira, *The Designof Object-Oriented Software With Domain Architecture Reuse,* The International Conference on Software Reuse, Advances in Software Reusability, Nov. 1-4, 1994, Rio de Janero, Brazil, Edited by William B. Frake, IEEE Computer Press Society, pp. 178-187.

[Dunn et. al. 1991] Michael Dunn, John Knight, *Software Reuse in an Industrial Setting: A Case Study,* IEEE 1991, pp. 329-338.

E

[El-Sharkawi et al. 1990] Mohamed El-Sharkawi, Yahiko Kambayashi, *Object Migration Mechanism to Support Updates in Object-Oriented,* PARBASE-90, International Conference on Databases, Parallel Architectures, and their Applications, Mar. 1990, Publ. by IEEE, pp. 378-387.

F

[Fishman et al. 1987] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Rayan, M. C. Shan, *Iris: An Object-Oriented Database Management System,* Readings in object-oriented databases, edited by Stanley Zdonik and David Maier, Morgan Kaufmann publishing, 1990, pp. 216-226.

[Frakes et. al. 1994] William Frakes, Sadahiro Isoda, *Success Factors of Systematic Reuse,* IEEE Software, July 1987, pp. 15-19.

[Frakes et. al. 1994] William Frakes, Thomas Pole, *An Empirical Study of Representation Methods for Reusable Software Components,* IEEE Transaction on Software Engineering, vol. 20, No. 8, Aug. 1994, pp. 617-630.

G

[Gargaro 1987] Anthony Gargaro, *Reusability Issues and Ada,* IEEE Software, July 1987, pp. 43-51.

[George et. al. 1993] Roy George, Bill Buckles, Fredrick Petry, *Modelling Class Hierarchies in the Object-Oriented Data Model,* Fuzzy Sets and Systems, Official Publication of the International Fuzzy Systems Association IFSA, Volume 60, Number 3, Dec. 24, 1993, pp. 259-272.

[Griss et. al. 1994] Martin L. Griss, W. (Vojtek) Kozaczynski, Anthony Wasserman, Christina Jette, Robert Troy, *Object-Oriented Reuse (Panel Discussion),* The International Conference on Software Reuse, Advances in Software Reusability, Nov. 1-4, 1994, Rio de Janero, Brazil, Edited by William B. Frake, IEEE Computer Press Society, pp. 209-213.

[Griss 1994] Martin L. Griss, *Architechting Kits for Reuse*, The International Conference on Software Reuse, Advances in Software Reusability, Nov. 1-4, 1994, Rio de Janero, Brazil, Edited by William B. Frake, IEEE Computer Press Society, pp. 216-217.

[Gupta et al. 1990] Rajiv Gupta, Melvin A. Breuer, *An Extensible User Interface for an Object-Oriented VLSI CAD* Framework, Proceedings of the first International Conference on Systems Integration ICSI, Apr. 1990, Publ. by IEEE pp. 559-568.

H

[Hudson et al. 1989] Scott Hudson, Roger King, *Cactis: A self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management Systems*, ACM Transactions on Database Systems, Vol. 14, No. 3, Sep. 1989, pp. 291-321.

J

[Jarke 1987] M. Jarke, R. Venken, *Database Software Development as Knowledge base Evolution in ESPRIT 87: Achievements and Impact*. Amsterdam, North-Holland, pp. 204-414.

[Joos. 1994] Rebbecca Joos, *Software Reuse at Motorola*, IEEE Software, September 1994, pp. 42-46.

K

[Kaiser et. al. 1987] Gail E. Kaiser, David Garlan, *Melding Software Systems from Reusable Building Blocks*, IEEE Software, July 1987, pp. 17-24.

[Katz et al. 1990] R. H. Katz, E. Chang, *Making Changes in a Computer-Aided Design Database*, Readings in object-oriented databases, edited by Stanley Zdonik and David Maier, Morgan Kaufmann publishing, 1990, pp. 400-407.

[Keene 1989] Sonya E. Keene, *Object-Oriented Programming in Common LISP, A Programmer's Guide to CLOS*, Addison-Wesley Publishing Company, 1989, pp. 139-151.

[Kim et al. 1987] Won Kim, Nat Ballou, Jay Banerjee, Hong-Tai Chou, J. F. Garza, Darrel Woelk, *Features of the ORION Object-Oriented Database System*,. Proceedings of the 13th International VLDB Conference 1987, published by the IEEE, pp. 154-187.

[Kim et al. 1987] Won Kim, Jay Banerjee, Hong-Tai Chou, J. F. Garza, Darrel Woelk, *Composite Object Support in an Object-Oriented Database System*, OOPSLA 87, Sigplan Notices, Vol. 22, No. 12, Dec. 1987, pp. 1118-125.

[Kim 1990] Won Kim, *A New Database for New Times*. Datamation, Jan. 1990, pp. 35-42.

[Konomi et al. 1993] S. Konomo, T. Furukawa, and Y. Kambayashi, *Super-Key Classes* *for Udating Materialized Derived Classes in Object Bases,* Lecture Notes in Computer Science, S. Ceri, K. Tanaka, and S. Tsur (eds.), Springer-Verlag, 1993, pp.310-326.

L

[Lee & Henry 1993] Wei Li, Sallie Henry, *Object-Oriented Metrics that Predicts Maintainability,* Journal of Systems Software, 1993, 23, pp. 111-122.

[Lenz et. al. 1987] Manfred Lenz,Hans Albrecht Schmid, Peter Wolf, *Software Reuse Through Building Blocks,* IEEE Software, July 1987, pp. 34-42.

[Lewis et. al. 1991] John Lewis, Sallies Henry, Dennis Kafura, Robert Schlman, *An Empirical Study of the Object-Oriented Paradigm and Software Reuse,* OOPSLA91 SIGPLAN Notices, vol. 26, No. 11, Nov. 1991, pp. 184-196.

[Lim 1994] Wayne Lim, *Effects of Reuse on Quality, Productivity, and Economics,* IEEE Software, September 1994, pp. 23-29.

[Lim 1996] Ik Sung Lim, *Design of a Flexible Unified Object-Oriented Simulation System Architecture,* A Ph. D. Dissertation Submitted at Wayne State University, Detroit, Michigan, U.S. ,1993, pp.79-108.

[Liu et al. 1993] K. Liu, D. Spooner, *Object-Oriented Databases Views for Supporting Multidisciplinary Concurrent Engineering,* Proceedings of the 17th Annual International Computer Software and Applications Conference, Publ. by IEEE, 1993, pp.19-25.

M

[Manfred et. al. 1987] Manfred Lenz, Hans Albrecht Schmid, and Peter F. Wolf, *Software Reuse through Building Blocks,* IEEE Software, July 1987, pp. 34-42.

[Mejabi 1993] O.O. Mejabi, *Object-Oriented Simulation Using Model Builder,* Proceedings of the 1993 Winter Simulation Conference, G. W. Evans, M. Mollaghasemi, E. C. Russel (eds.), pp. 303-307.

[Meyer 1987] Bertnard Meyer, *Reusability: The Case for Object-Oriented Design,* IEEE Software, March 1987, pp. 50-63.

[Moore 1988] D.J. Moore, P.A. Drewe, M.s. Ganti, R.j. Nassif, S. Podar, D.H. Taenzer, *Vishnu: An Object-Oriented Database Management System Supporting Software Engineering,* Proceedings of the Tewlfth Annual International Computer Software & Applications Conference, Oct. 1988, Publ. by IEEE pp. 186-192.

[Morsi et al. 1991] Magdi Morsi, Shamkant Navathe, Hyoung-Joo Kim, *A Schema Management and Prototyping Interface for an Object-Oriented Database Environment,* Object Oriented Approach in Information Systems, F. Van Assche,

B. Moulin, C. Rolland (eds.) North Holland, 1991 IFIP, pp. 157-179.

[Morsi et al. 1992] Magdi Morsi, Shamkant Navathe, Hyoung-Joo Kim, *An Extensible Object-Oriented Database Testbed*, Proceedings-International Conference on Data Engineering, Feb. 1992, Publ. by IEEE pp. 150-157.

N

[Narayanaswamy et al. 1988] K. Narayanaswamy , K. V. Bapa Rao, *An Incremental Mechanism for Schema Evolution in Engineering Domains*, Proceedings of the International Conference on Data Engineering 1988, published by the IEEE, pp. 294-301.

[Nguyen et al. 1987] G. T. Nguyen, D. Rieu, *Expert Database Support for Consistent Dynamic Objects*, Proceedings of the 13th VLDB Conference, Brighton Sep. 1987, pp. 493-500.

[Nguyen et al. 1989] G. T. Nguyen, D. Rieu, *Schema evolution in object-oriented database systems*, Data & Knowledge Engineering, 4, (1989), North Holland, pp. 43-67.

[Ning et. al. 1994] Jim O. Ning, Kanth Miriyala, W. (Vojtek) Kozaczynski, *An Architecture-driven, Business-specific, and Component-based Approach to Software Engineering*, The International Conference on Software Reuse, Advances in Software Reusability, Nov. 1-4, 1994, Rio de Janero, Brazil, Edited by William B. Frake, IEEE Computer Press Society, pp. 84-93.

O

[O'Connor et. al. 1994] James O'Connor, Catherine Mansour, Jerri Turner-Harris, Grady Campbell, Jr., *Reuse in Command-Control Systems*, EEE Software, September 1994, pp. 70-79.

P

[Parker 1997] Kevin Parker, *Big, Bigger, Best Manufacturing Systems*, Chilton's Manufacturinf Systems, July 1997, pp. 20-27.

[Penny et al. 1987] D. Jason Penny, Jacob Stein, *Class Modification in the Gemstone Object-Oriented DBMS*, OOPSLA 87, Sigplan Notices, Vol. 22, No. 12, Dec. 1987, pp. 111-117.

R

[R. Olivia et al. 1992] Olivia R., Liu Sheng, Chih-Ping Wei, *Object-Oriented Modeling and Design of Coupled Knowledge-base/Database Systems*, Proceedings-International Conference on Data Engineering, Feb. 1992, Publ. by IEEE pp. 98-105.

[Rajlich et. al. 1996] Vaclav Rajlich, Joao Silva, *Evolution and Reuse of Orthogonal Architecture*, IEEE Transactions on Software Engineering, vol. 22, No. 2, Feb.

19961996, pp. 153-157.

[Ramamoorthy et al. 1984] C. V. Ramamoorthy, A.Prakash, W. T. Tsai, and Y. Usuda, *Software Engineering: Problems and perspectives*, IEEE Computer, Vol. 17, No. 10, Oct. 1984, pp. 191-209.

[Ratcliffe 1987] M. Ratcliffe, *Report on a Workshop on Software Reuse*, Hereford, UK on 1-2 May 1986, ACM SIGSOFT Software Engineering Notes, vol. 12, No. 1, Jan. 1987, pp. 42-47.

[Riet 1989] R. P. van de Riet, *MOKUM: An object-oriented active knowledge base system*. Data and Knowledge Engineering, v4, n1, July 1989 pp. 21-42.

[Rine 1991] David Rine, *A Short Overview of a History of Software Maintenance: As it Pretains to Reuse*, ACMSIGSOFT Software Engineering Notes, vol. 16, No. 4, Oct. 1991, pp. 60-63.

[Rovira et al. 1993] M. Rovira, D. Spooner, J. Haddock, *The Concepts of Views in Simulation*, Proceedings of the 1993 Winter Simulation Conference, G.W. Evans, M. Mollaghasemi, E. C. Russel, W. E. Biles (edt.), 1993, pp. 552-559.

[Rundensteiner 1993] Elke Rundensteiner, *Tools for View Generation in Object-Oriented Databases*, Proceedings of the 2nd International Conference on Information and Knowledge Management, 1993, Publ. by ACM, N.Y., pp. 635-644.

[Rundensteiner 1993] Elke Rundensteiner, *Design Tool Integration Using Object-Oriented DatabasesViews*, Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Publ. by IEEE, N.J., pp. 104-107.

S

[Schapiro 1989] David Schapiro, *A continuous Approach to Object-Oriented Software Analysis*. Fourth Israeli conference on Computer Systems and Software Engineering, June 1989, Publ. by IEEE pp. 120-127.

[Sciore 1989] Edward Sciore, *Object Specialization*, ACM Transactions on Information Systems, Vol. 7, No. 2, April 1989, pp. 103-122.

[Seidewitz 1989] E. Seidewitz, *General Object-Oriented Software Development: Background and Experience*. The Journal of Systems and Software, No. 9, 1989, pp. 95-108.

[Senn et al. 1995] J. Senn and J. Wynekoop, The Other Side of CASE Implementation Information Systems Management, Fall 1995, pp. 7-14.

[Shriver 1987] Bruce Shriver, *Reuse Revisited*. IEEE Software, January 1987, pp. 5.

[Steele 1990] Guy Steele, Common LISP, The Language, second edition, Digital Press, 1990.

[S.Y.S. et al. 1991] S. Y. S, Rahim Yaseen, H. Lam, *An Extensible kernel object management system*, Proceedings of OOPSLA 91, pp. 247-263.

T

[Tracz 1987] Will Tracz, *Reusability Comes of Age*, IEEE Software, July 1987, pp. 6-8.

[Turbide 1993] David Turbide, *MRP + The Adaptation, Enhancement, and Application of MRP II*, Industrial Press, 1995, pp. 1-12.

[Turbide 1995] David Turbide, *MRP II Still Number one!*, IIE Solutions, May 1995, pp. 28-31.

V

[Verlage et al. 1992] Martin Verlage, Peter Knauber, *Just-in-Time Initialization of Objects Representing Software Processes*, Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing SAC, Mar. 1992, Publ. by ACM pp. 706-710.

[VerDuin 1995] William VerDuin, *Better Products Faster, A Practical Guide to Knowledge-Based Systems for Manufacturing*, IRWIN Publ., 1995.

W

[Wiederhold 1991] Gio Wiederhold, *Views, Objects, and Databases*, Topics in Information systems, K. R. Dittrich, U. Dayal, and A. P. Buchmann (eds.), Springer-Verlag, 1991, pp.29-43.

[Weide et. al. 1994] Bruce Weide, William Ogden, Murali Sitaraman, *Recasting Algorithms to Encourage Reuse*, IEEE Software, September 1994, pp. 80-88.

Z

[Zdonik 1983] Stanley Zdonik, *An Object Management System for Office Applications*, Management and Information Systems, Languages for Automation, Edited by Shi-Kuo Chang, 1985, pp. 197-222.

ABSTRACT

A FLEXIBLE ARCHITECTURE FOR MANUFACTURING PLANNING SOFTWARE MAINTENANCE

by

YOUSIF A. MUSTAFA

May 1998

Advisor Dr. O. Mejabi

Major: Industrial Engineering

Degree: Doctor of Philosophy

Computer software systems took on a new role in manufacturing planning with the introduction of Material Requirement Planning (MRP) system in 1965. The MRP system generates material requirement lists in response to given production requirements. In this way, inventory management, purchasing, and shipping activities are linked to manufacturing. In 1979, Manufacturing Resource Planning (MRP II) systems were introduced [VerDuin 1995]. MRP II typically includes planning applications, customer order entry, finished goods inventory, forecasting, sales analysis, production control, purchasing, inventory control, product data management, cost accounting, general ledger processing, payables, receivables, and payroll [Turbide 1995]. An emerging market is developing for software systems that expand the scope of MRP II farther to encompass activities for the entire organization. Among these systems are Enterprise Resource Planning (ERP), Customer-Oriented Manufacturing Management System (COMMS), and Manufacturing Execution Systems (MES). These systems integrate marketing,

manufacturing, sales, finance, and distribution to move beyond optimizing production alone, to optimizing the organization's multiple objectives of low cost, rapid delivery, high quality, and customer satisfaction [VerDuin 1995]. MRP II is still the dominant solution for manufacturing in tens of thousands of companies. These companies range in size from less than a million dollars in sales right up to the top of Fortune 500 companies. However, this is a market penetration of only 11% which clearly shows the size and potential of the opportunity for MRP II development. Yet, despite the commonality of needs across the scope of manufacturing, there are distinct differences when comparing plant to plant, company to company, and industry to industry. Often MRP II has to be modified to adapt to a particular industry [Turbide 1993]. This modification often pushes the cost even higher and makes MRP II more out-of-reach for many companies.

Therefore, it would be highly beneficial for the overall scope of manufacturing if a highly flexible low-cost MRP II system can be developed.

This research presents a flexible architecture for development and maintenance of manufacturing planning software, especially MRP II. The architecture uses the concept of software reuse and is built on top of run-time object-oriented framework.

AUTOBIOGRAPHICAL STATEMENT

YOUSIF A. MUSTAFA

PERSONAL DATA:	Born on October 17, 1947 in Iraq.
EDUCATION:	B.S. in Physics September 1970, College of Science, University of Baghdad, Iraq.
	M.S. in Operations Research May 1993, College of Engineering, Wayne State University, USA.
	Ph. D. in Industrial Engineering December 1997, College of Engineering, Wayne State University, USA.
EXPERIENCES:	1995-Date, Associate Professor Madonna University, Livonia, Michigan, USA.
	1990-1995, College Instructor Lewis College of Business, Detroit, Michigan, USA
	1988-1990, College Instructor City International College of Business, Sydney, Australia.
	1970-1987, Computer Programmer/Analyst, Iraq, Syria, and the USA.
	1970-1977, High School Physics Teacher, Baghdad, Iraq.
INTERESTS:	Academic research and teaching
HOBBIES:	Reading and following political events and news.