THE DESIGN AND IMPLEMENTATION OF A LARGE SCIENTIFIC CODE USING FORTRAN 90

CENTRE FOR NEWFOUNDLAND STUDIES

TOTAL OF 10 PAGES ONLY
MAY BE XEROXED

(Without Author's Permission)

DARRYL REID

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisisitons et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-89666-8
Our file Notre référence
ISBN: 0-612-89666-8

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

THE DESIGN AND IMPLEMENTATION OF A LARGE SCIENTIFIC CODE USING FORTRAN 90

by

© Darryl Reid, B.Sc.

A thesis submitted to the School of Graduate

Studies in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Chemistry

Memorial University of Newfoundland

St. John's Newfoundland, Canada

February 2003

St. John's, Newfoundland Canada

Abstract

Computational science has become an integral part of scientific research over the past couple of decades. From the beginnings of computing, scientists have written codes to help them solve problems. The language of choice for most scientific computing for the past 20+ years has been FORTRAN 77. However, modern advancements in programming languages, such as the idea of object-oriented programming, and other features such as dynamic memory allocation, have caused many scientific programmers to look for an alternative to FORTRAN 77. This work aims to show that Fortran 90/95 is a viable option for these scientific programmers, and although it is not fully object-oriented many of the desired features of an object-oriented language can be implemented in Fortran 90/95. This work sets out a series of design protocols and an overall programming scheme which makes writing large scientific codes more manageable. A series of specific programming tools and choices will be described which aid both the programmer and the user of the codes. Finally, some examples of the implementation of these ideas and practices will be included.

Acknowledgments

There are many individuals to which I would like to extend my deepest gratitude for their guidance and support while I completed this work. The road to completion was a long one and the following people stood by me throughout. Firstly, I have to thank my supervisor, Dr. Raymond Poirier, for the hours, upon hours, of work. Ray is a constant source of ideas and inspiration, without which this thesis would not have happened. I would also like to thank the Poirier research group, Michelle Shaw, Tammy Gosse, Aisha El-Sherbiny, and Sherene Bungay. Each of them helped in there own way, often just lending an ear when that was needed. Much of the ground work for this thesis was laid by Michelle. In addition I have to thank the Mun Chemistry Society, and all my friends at MUNCS for making my time at Memorial a very enjoyable one. I also want to extend my gratitude to the Chemistry Department, the staff and faculty made me feel like one of them and helped to guide many of my life decisions. Lastly I must thank my family. My family have always been there for me in more ways then they will ever know. Thanks to everyone, your kindness and love have meant the world to me.

Contents

List of Figures

List of Tables

List of Abbreviations

AM1	Austin Model 1
AMBER	Assisted Model Building with Energy Refinement
B3LYP	Becke-Lee-Yang-Parr
B3P86	Becke's 3 - Perdew 86
BSSE	Basis Set Superposition Error
CASSCF	Complete Active Space Self-Consistent Field
CC	Coupled Cluster
CFF	Consistent Force Field
Charmm22	Chemistry at Harvard Macro-molecular Mechanics 22
CI	Configuration Interaction
CNDO	Complete Neglect of Differential Overlap
COMPASS	Condensed-phase Optimized Molecular Potentials for Atomistic Simulation Studies

CVFF	Consistent Valence Force Field
DFT	Density Functional Theory
G961LYP	Gill 96-Lee-Yang-Parr
GVB	Generalized Valence Bond
HF	Hartree-Fock
HPF	High Performance Fortran
INDO	Intermediate Neglect of Differential Overlap
MC-SCF	MultiConfiguration Self-Consistent Field
MINDO	Modified Intermediate Neglect of Differential Overlap
MM	Molecular Mechanics
MMFF94	Merck Molecular Force Field, 1994 version
MNDO	Modified Neglect of Diatomic Overlap
MPn	Møller-Plesset theory, nth order
NDDO	Neglect of Differential Diatomic Overlap
OSIPE	Open Structured Interfaceable Programming Environment
PIC	Proper Internal Coordinates
PM3	Parametric Method number 3
RHF	Restricted Hartree-Fock
RIC	Redundant Internal Coordinates
SVWN	Slater-Vosko-Wilk-Nusair
UHF	Unrestricted Hartree-Fock

XYZ	Cartesian (x, y, z) coordinate system
ZM	Z-matrix

Glossary

Class

For the purposes of this thesis, a class is a term used to describe the organization of the code, that is, a class is a group of objects that have something in common. The term is used loosely and the "something in common" statement can be applied to suit the programmers needs. It is used as part of the total object name, which includes the *name* and *modality* as well.

Compatibility

Compatibility is the ability of subprograms to be combined easily without conflicts between each part. This is a necessity for projects which involve more than one contributor. This is important in enabling software parts to interact with one another[1].

Correctness and Robustness

Correct code does what it was intended to do for all possible known cases. This is the most important feature of the code, since if the code does not do what it is designed to then the other features are meaningless. However, one cannot possibly know all cases that need to be run. In these cases, should the program fail, it should do so in a clean manner with the proper error message to alert the user or programmer of the problem. This feature is known as *robustness*[1].

Ease of Use

The software should be *easy to use* (operate, prepare input, analyze output, handle errors) and should come with documentation to instruct the user on how to deal with problems when encountered[1].

Efficiency

The program should be *efficient,* that is it should make optimal use of the hardware and software components of the system it runs on[1.]

Extendibility

Extendible programs are easy to modify or extend, for example when a new feature is needed. For small programs this is not an issue but for large complex programs it is essential. In order to make code more extendible, two things can be done[1]:

1. *Simplify the code:* the program should be designed in a simple manner, with a simple architecture.

2. *Divide program into smaller parts*: The program can be divided into smaller independent subunits of the program.

Fortran, FORTRAN 77, and Fortran 90/95

In the mid 1950's and IBM development team, headed by John Backus, developed a new language which made programming much easier. One achievement of this new language was that it provided a much more intuitive way to code mathematical formulas, and so was named FORTRAN, taken from the first few letters of "***For**mula **Tran**slation*". Over the years this language was revised and in 1977 a new standard, developed by American National Standards Institute (ANSI), was released, known as FORTRAN 77. This version of Fortran was very popular, and is still used extensively in scientific programming today. Over the next decade or more the ANSI committee discontinued the traditional all capital letters in the official name of the language and started using "Fortran" with only the first letter capitalized. Developments continued and in 1990 the new standard Fortran 90 was introduced[2]. Minor updates and fixes, with some minor extensions led to the release of the newest Fortran standard Fortran 95. The differences in Fortran 90 and Fortran 95 are so small that in this thesis the term Fortran 90/95 will be used to indicate either can be used.

Integrity

The program should have *integrity,* that is the program components should not be able to corrupt one another. Utilities can be designed to handle security within the program and this should be an essential part of the software design[1].

Modality

Modality is another characteristic of an object. It is used to represent a level of theory or a coordinate system that the calculations are performed in or at. An object can "inherit" the modality portion of its name from a parent call.

Object

In this thesis an object is something that can be printed or used by other portions of the code. An object usually represents something, that something in computational chemistry often comes from theory.

Reusability

Reusability of the code is also an important feature. The more code is reused, the less code needs to be rewritten and this reduces the cost of development. What parts can be reused is determined by finding parts of the code that are the same or share a common piece[1].

Portability

Portability, the capability of the program to run on a few different systems, is an important feature. In ensuring this feature, any machine-specific parts should be clearly defined in the documentation[1].

Verifiability

The programmer should be able to prepare test data and procedures to determine if there are any problems with the software. This would be best accomplished if test data was included with the software package along with instructions on running test data and listing of expected output[1].

Chapter 1

Introduction

1.1 Project Description

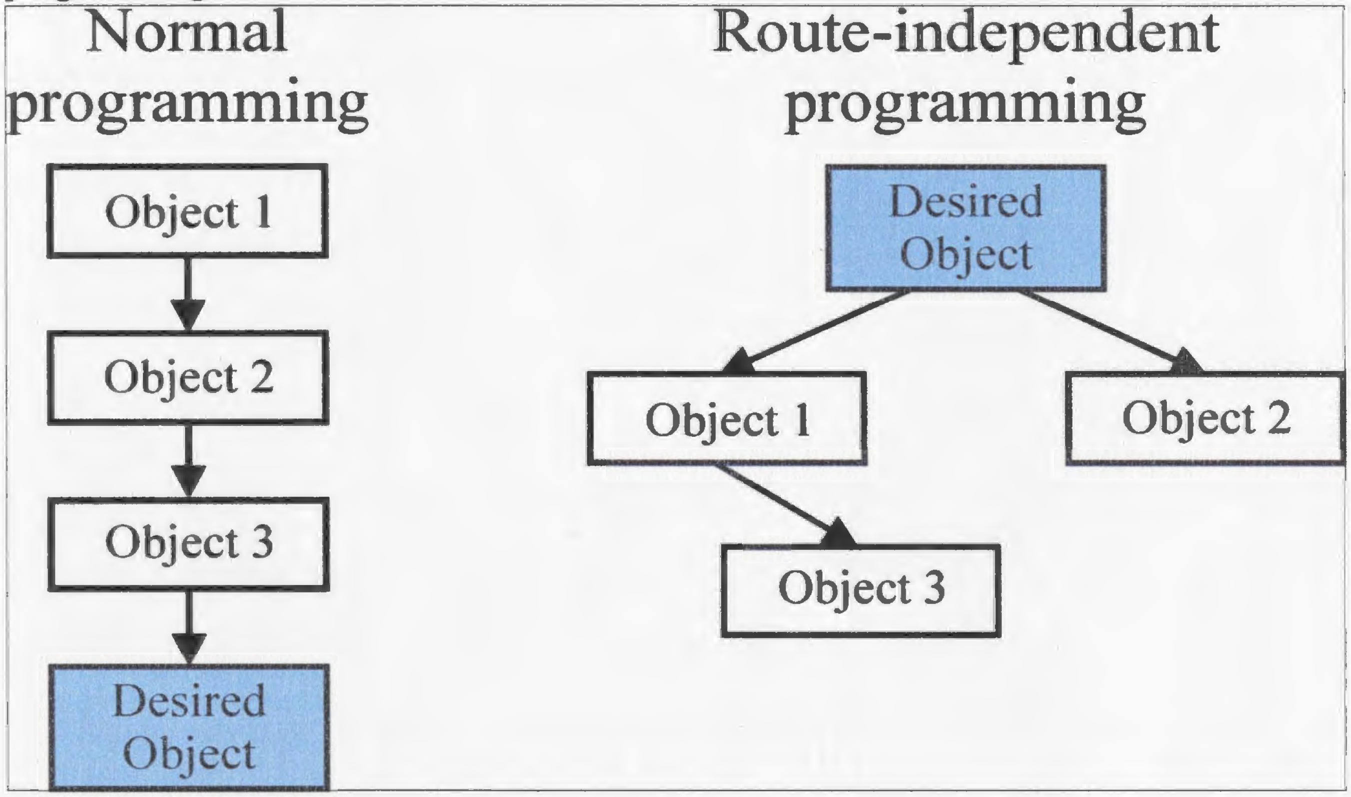
The use of computers to aid scientific research has exploded over the past 30-40 years. With the advent of higher level programming languages such as FORTRAN 77 and C, more and more scientist were able to create programs designed to perform very specific tasks, applicable to their own research. Over the years, these codes were changed and added to for slightly different applications. This process of code evolution often involves many different programmers over many years. Much of the existing scientific code is written in this manner.

The problem with this kind of evolution is that there is no clear plan guiding the process. Much of the code was written by scientists that were not trained programmers. The codes are often written with little or no comments, each programmer has their own individual style, and the spaghetti nature made the codes almost impossible to follow. These problems make maintaining and updating those codes a very difficult, if not impossible task, unless the programmer is extremely familiar with the entire code. Since these codes are often written by students completing an honours, masters, Ph.D., or some other project, they often do not have intimate knowledge of the code. For this reason a systematic approach to writing large scientific codes is required. For the purpose of this thesis, the term "large scientific code" refers to programs on the order of 10^5 to 10^6 lines of code, consisting of hundreds of files, and thousands of functions and subroutines. The ideas expressed will also be useful for much smaller programs.

1.2 Goals

In this work an approach at creating large scientific codes is presented. The approach is as general as possible and can be applied to virtually any scientific application. However, the examples used are from an ab initio quantum chemistry package, namely MUNgauss. MUNgauss is a fairly large code consisting of over 200 files, well over 1500 functions and subroutines and has been in development for over 20 years. The code was originally written in FORTRAN 77 and was fairly modular and readable. MUNgauss took advantage of OSIPE[3] tools, which gave it a degree of modularity. OSIPE (Open Structured Interfaceable Programming Environment) consists of a set of FORTRAN 77 tools which created a kind of low-level object-oriented programming environment. OSIPE made use of a large common

Figure1.1.1: Illustration of route-independent programming compared to normal programming



block to allow access to data throughout the program. OSIPE also required that an object (defined as an entity the program is able to address i.e., scalar, vector, matrix, tensor,...) can only be created by one routine which created nothing else, giving the code a structured and modular form.

OSIPE also gave MUNgauss a very useful feature, route-independence. Route-independence simply means that there is no predetermined "path of execution" of the code. Figure 1.1.1 shows an illustration of route-independent compared to normal programming. On the left of the figure it can be seen that if an object was desired (in blue) the programmer would first have to create object 1, then object 2, then object 3, to finally create the desired object. This means that the programmer has to know how to construct each object necessary to build the desired object. In route-independent programming, if the programmer needs to build an object, the programmer only needs to know the first order dependence, i.e., the objects that the desired object directly depends on. The building routine of the desired object knows that it requires object 1 and object 2 to build the desired object. The desired object then "asks" for object 1 and 2, if they exist then it uses them, if not the code builds them. Note here that object 1 depends on object 3, thus object 3 is created once the desired object "asks" for object 1. The builder of the desired object does not need to know anything about object 3, since the desired object does not depend directly on object 3.

The emergence of Fortran 90 and its new features (which will be explained in section 2.3), prompted the decision to overhaul the design of MUNgauss to create a better working environment for future developments. Many of the desirable features in the OSIPE version of MUNgauss were kept in the newly designed version, namely the modular approach, route independent character, and several others.

Generally speaking, the new design also tries to include other commonly considered "good programming" features. They include, *correctness, robustness, extendibility, reusability, compatibility, efficiency, portability, verifiability, integrity, ease of use,* and *proper* *documentation*[1]. These terms are defined in the glossary.

1.2.1 User and Programmer Interests

When developing code, it is important to remember that the user and the programmer have different needs. From the user's point of view, the program should be correct, robust, compatible, portable, efficient, easy to use, and be well documented. From the programmer's point of view the program should have all of the above features and also be readable, extendible, reusable (or have reusable parts), verifiable, and have integrity. Ideally for both the user and the programmer, the program should be optimal in every feature, but this is not always possible. As a result, there are trade-offs between these features. When designing programs one should try and balance each feature in the best way possible.

This work will present an approach to code design which addresses these desired features. In addition to these very general features, more project specific features will be added to the code; these will be discussed in chapter 2.

1.3 Outline

This work presents an approach to creating large scientific programs. Chapter 2 describes the process of problem description and reasons for choosing Fortran 90/95 as the language in which to write these codes. Chapter 3, will present a list of strict design protocols for the programmers to follow. Following these protocols will create a program that will allow others to read, understand and maintain the code more easily. A method of writing the code is contained in Chapter 4. This section lays out a program design which creates very useful features that can be used for debugging and code optimization. Chapter 5, presents the total picture, and how to add new functionality to the program. Often the new functionality is an older peice of code writen in FORTRAN 77, by following the procedure in Chapter 5, old FORTRAN 77 code can be converted to be used in the new Fortran 90 environment. An example of such a conversion will also be included.

Chapter 2

Problem Description

2.1 Defining the Problem

The first step in the creation of a well written scientific program is to know what problem is to be solved. Writing a description of the problem is a very important and often overlooked step in creating a program. This initial step allows for the creation of an overall plan to tackling the problem. It also allows for the separation of the problem into groups (classes and objects). These classes and objects will be used to organize and modularize the code.

It is important to clarify the use of the terms classes and objects. Fortran 90/95 is not an object-oriented language, however, it does contain many of the features of an object-oriented language. The terms are not used in the object-oriented sense. These terms are used in the thesis to represent an organization method. The code was analyzed in such a way as to group like-things together. An object is defined, in relation to the program description, as something that the code can create, use, and/or print. Classes are then defined as a group of objects which have something in common. This differs greatly from the use of these terms in object-oriented programming, where an object is a specific instance of a class. These terms will be further clarified throughout the problem description.

2.1.1 Program Description

To start the organization process the main purpose of the code, MUNgauss, had to be written.

> *"To create a quantum chemistry program which will have ab initio, density functional theory, molecular mechanics, and other packages."*

In this purpose statement the high level divisions of any quantum chemistry package can be seen, i.e. *ab inito*, density functional theory (DFT), molecular mechanics (MM), etc. These high level divisions are used as the basis for the code organization.

This is a top-down organization method. Researchers find it easier to visualize this organization method. A theoretical chemist, for example, will think of results in terms of what theory is used to calculate them. Molecular mechanics, *ab initio*, or DFT calculations are thought of differently by a theoretical chemist[4]. For example, the results of an *ab initio* calculation may be written as MP4/6-31G(d)//HF/6-31G(d), meaning that the energy was calculated at the MP4 level of theory, with a 6-31G(d) basis set, using a Hartree-Fock (HF) with 6-31G(d) basis set, optimized geometry. The design style should to reflect this, so the first level of division is at the theory level. When starting to write a program these different theories are "naturally" distinct, and thus are a good first level of division. Remember the programming style designed aims to be programmer friendly, since the target programmers are scientists, the style should reflect their way of thinking. For a large, complex code, such as MUNgauss, a top-down organization method provides clarity to the "readers" of the code, i.e greater readability.

The next level of division would be based on the methods of calculation. The wavefunction, in ab initio calculations distinguishes between different methods, be it Restricted Hartree-Fock (RHF), generalized valence bond theory (GVB), or unrestricted Hartree-Fock (UHF), or one of many others[4]. For molecular mechanics (MM) there are different force fields which must be treated differently, MMFF90 is the current force field which has been implemented in MUNgauss[5]. Similarly density functional theory (DFT) has many different methods which could be implemented, e.g. B3LYP or SVWN[5]. Note that not all these methods are currently available in MUNgauss but the design of the code must allow for the addition of different packages, as the field of computational chemistry is a very dynamic one. These methods are the basis of the next division of code, the term *modality* is used to describe this level of division. Table 2.1 shows a list of many of the possible modalities which could be incorporated in a quantum chemistry package.

Table 2.1.1: Possible modalities for a quantum chemistry package.

Level of Theory				Coordinates
ab initio	semi-empirical	MM	DFT	
RHF	CNDO	MMFF94	B3LYP	XYZ
UHF	INDO	CFF	B3P86	ZM
MP*n*	MINDO	Charmm22	G961LYP	PIC
CI	NDDO	AMBER	SVWN	RIC
CC	MNDO	CVFF		
MC-SCF	AM1	COMPASS		
CASSCF	PM3			

Another division which can be made at this level is that of the coordinate system. The coordinate system is not a level of theory but each coordinate system must be treated differently in the calculations, therfore the division was made. Some coordinate systems available in MUNgauss are cartesians, Z-Matrix (ZM), proper internal coordinates (PIC), or redundant internal coordinates (RIC).

The modalities above are chosen as a way to split the code along the lines of thought of the theoretical chemists.

The next step is to write a list of the individual components the code can produce. This will be a list of the objects which will be later classified into classes for further code organization.

2.1.2 Classes and Objects

Objects

What is an object? This is not an easy thing to precisely define. As mentioned, OSIPE[3] defined an object as an entity the program was able to address i.e. scalar, vector, matrix, tensor, etc., and can only be created by one routine which created nothing else. In this context an object is defined as something that can be either printed or used by other parts of the code to preform their computations. The objects are what the code actually uses, they contain all the calculations and subsequent results of the calculations preformed in the program. An object should represent something. Usually, in the case of a computational chemistry code, that something comes from the theory which defines the problem. For example, when calculating the energy of a molecule, the distances and angles between atoms in the molecule are needed. These are two objects of the program, named `ATOMIC_DISTANCES` and `ANGLE_ATOMS` respectively in MUNgauss. Below is a list of many of the objects which MUNgauss needs/uses to preform computations.

Table 2.1.2: List of Objects in MUNgauss

1E_AO	COMPONENTS	INTERNAL%ZM
1E_DIPOLE	CONN_FAILSAFE	INVERSE
1E_OVLAP_AO_A_B_BLOCK	CONN_PRUN	INVERSE_FACTORED
1E_SM1TV_AO_AB	CONN_PRUN_HOMRED_INCIDENCE	LENGTH_FULL
1E_SSM1_AO_AB_BLOCK	CONNECT	LOCALIZED
1E_TV_AO_AB	CONVAL	LOCALIZED_LIST%GVB
1E_TV_AO_MONOMER_A	COORDINATES_PIC	MATRIX%ANALYTICAL
1E_TV_AO_MONOMER_B	DEFAULTS	MATRIX%NUMERICAL
1E_V_AO_AB	DENSITY	MM_CONTRIBUTIONS
1E_V_AO_MONOMER_A	DENSITY%RHF	MO
1E_V_AO_MONOMER_B	DIPOLE_MOMENT	MO%GVB
1MATRIX	EDGE_RING_TO_RING_ASSEMBLY	NET_ATOMIC%MULLIKEN
1MATRIX%GVB	EDGE_TO_RING	NON_BONDED
1MATRIX%RHF	ENERGY	NON_BONDED_ATOMS
1MATRIX%UHF	ENERGY_CONTRIBUTIONS	NUMBER_OF_PARAMETERS
2E%COMBINATIONS	ENERGY_TOTAL	NUMERICAL
2E%RAW	ENERGY_WEIGHTED	OBJECTS_CREATED
ANALYTICAL	ENERGY_WEIGHTED%GVB	OBJECTS_STATUS
ANGLE_ATOMS	ENERGY_WEIGHTED%RHF	OOPBEND_ATOMS
AO%MULLIKEN	ENERGY_WEIGHTED%ROHF	OPT
AO_BY_AO%MULLIKEN	ENERGY_WEIGHTED%UHF	OPT_BFGS
ATOM_TYPES	FACTORED	OPT_DIIS
ATOMIC%MULLIKEN	FOCK_MATRIX	OPT_OC
ATOMIC_DISTANCES	FREQUENCIES	OPT_VA
ATOMIC_MASSES	FUNCTION	PAIR_LIST_CMO
ATOMIC_NUMERICAL	FUNDRING	PARAMETER_CONTRIBUTIONS
BMATRIX	GAMMA_FUNCTION	PARAMETERS
BMATRIX%PIC	GRADIENTS	POINT
BMATRIX%RIC	GUESS	PROGRAM
BMATRIX%ZM	HESSIAN	PRUNE_HOMRED
BOND_ATOMS	HUCKEL	RIC
BOND_ORDER	INTEGRALS	SCF
CARTESIANS	INTEGRALS_MO	STEP_SIZE
CLOSE_CONTACT	INTERNAL	THIRD_SEMI_DIAGONAL
CNCOMP	INTERNAL%PIC	TORSION_ATOMS
COEFFICIENTS	INTERNAL%RIC	VAN_DER_WAAL_BONDS
COEFFICIENTS%GVB		VERTEX_LIST
COEFFICIENTS%RHF		VERY_WEAK_BONDS
COEFFICIENTS%UHF		
COEFFICIENTS_OV		
COMCON		
COMP_ATOM_NUM		

This list shows another feature which is very important to the readability of the code, the naming of objects are meaningful to anyone reading it. A reader with knowledge of the theory of the different methods being coded, would recognize most of the objects listed above as something from the theory. As stated earlier MUNgauss is an old code that is being converted to the new programming style, the list above shows this conversion is a process, it contains both objects named in the new style and those created in the earlier versions of the code. These objects have not yet been given proper names but will be converted as the project continues. Also note that some object names contain a percent,'`%`', character followed by one of the modalities defined above. This shows that the code can create different "flavors" of these objects, i.e. it can create a matrix used in calculating density, the `1MATRIX`, for `GVB`, `UHF`, or `RHF`.

Classes

The objects must now be grouped in some logical manner which will help modularize the code. These groups are referred to as classes. The process of deciding which class an object belongs to, is often a difficult one, and one for which there is no set procedure. It is a process which will involve constant re-evaluation as coding proceeds. There are some considerations when deciding class, they include:

- Placing an object into a class should help the code organization
- All objects in a class should have something in common
- Objects within a class are built in a similar fashion

Figure 2.1.1 shows the objects given above divided into classes. As with the list of objects provided this list of classes is not finalized. This is still a work in progress and classes are redefined, created, and removed, as the code continues to develop.

Figure 2.1.1: MUNgauss OBJECTS grouped into **CLASSES**

BASIS_SET
STO_3G

COORDINATES
BMATRIX
BMATRIX%ZM
BMATRIX%PIC
BMATRIX%RIC
RIC

CHARGE_DENSITY
POINT
ATOMIC_NUMERICAL

DATA
GAMMA_FUNCTION

DEFAULTS
GRADIENTS
GUESS
HESSIAN
INTEGRALS
INTEGRALS_MO
OPT
OPT_VA
OPT_BFGS
OPT_OC
OPT_DIIS
COORDINATES_PIC
SCF
PROGRAM

DENSITY
1MATRIX
1MATRIX%RHF
1MATRIX%GVB
1MATRIX%UHF
ENERGY_WEIGHTED
ENERGY_WEIGHTED%RHF
ENERGY_WEIGHTED%UHF
ENERGY_WEIGHTED%GVB
ENERGY_WEIGHTED%ROHF

DERIVATIVES
THIRD_SEMI_DIAGONAL

ENERGY
COMPONENTS

FOCK
HUCKEL

FORCE_CONSTANT
NUMERICAL
MM_CONTRIBUTIONS

FUNCTION
ENERGY

GRADIENTS
COMPONENTS
CARTESIANS
INTERNAL
INTERNAL%ZM
INTERNAL%PIC
INTERNAL%RIC
OPT
LENGTH_FULL
MM_CONTRIBUTIONS

GUESS
DENSITY
DENSITY%RHF
MO
MO%GVB
HESSIAN

GVB
DEFAULTS
PAIR_LIST_CMO

RHF
FOCK_MATRIX

GRAPH
ANGLE_ATOMS
BOND_ATOMS
CLOSE_CONTACT
COMCON
COMP_ATOM_NUM
CNCOMP
CONNECT
CONN_FAILSAFE
CONN_PRUN
CONN_PRUN_HOMRED_INCIDENCE
CONVAL
EDGE_RING_TO_RING_ASSEMBLY
EDGE_TO_RING
FUNDRING
OOPBEND_ATOMS
PRUNE_HOMRED
TORSION_ATOMS
VERTEX_LIST
VERY_WEAK_BONDS
VAN_DER_WAAL_BONDS
NON_BONDED
NON_BONDED_ATOMS

HESSIAN
ANALYTICAL
FACTORED
INVERSE
INVERSE_FACTORED
MATRIX%NUMERICAL
MATRIX%ANALYTICAL
STEP_SIZE

INTEGRALS
1E_AO
1E_DIPOLE
1E_OVLAP_AO_A_B_BLOCK
1E_SM1TV_AO_AB
1E_SSM1_AO_AB_BLOCK
1E_TV_AO_AB
1E_TV_AO_MONOMER_A
1E_TV_AO_MONOMER_B
1E_V_AO_AB
1E_V_AO_MONOMER_A
1E_V_AO_MONOMER_B
2E%COMBINATIONS
2E%RAW

MO
HUCKEL
COEFFICIENTS
COEFFICIENTS%RHF
COEFFICIENTS%GVB
COEFFICIENTS%UHF
COEFFICIENTS_OV
LOCALIZED
LOCALIZED_LIST%GVB

MM
ATOM_TYPES
ENERGY_TOTAL
ENERGY_CONTRIBUTIONS
PARAMETER_CONTRIBUTIONS

MOLECULE
ATOMIC_DISTANCES
ATOMIC_MASSES

OPT
FUNCTION
HESSIAN
NUMBER_OF_PARAMETERS
PARAMETERS
STEP_SIZE

PARAMETERS
FREQUENCIES

POPULATION_ANALYSIS
AO_BY_AO%MULLIKEN
AO%MULLIKEN
ATOMIC%MULLIKEN
NET_ATOMIC%MULLIKEN
BOND_ORDER

PROPERTIES
DIPOLE_MOMENT

PROGRAM
OBJECTS_CREATED
OBJECTS_STATUS

The bolded names in the first row of each box is the class name and below are the objects which belong to that class.

2.1.3 Overall Code Organization

Figure 2.1.2 shows the overall code organization. It shows the 5 levels of modularity:

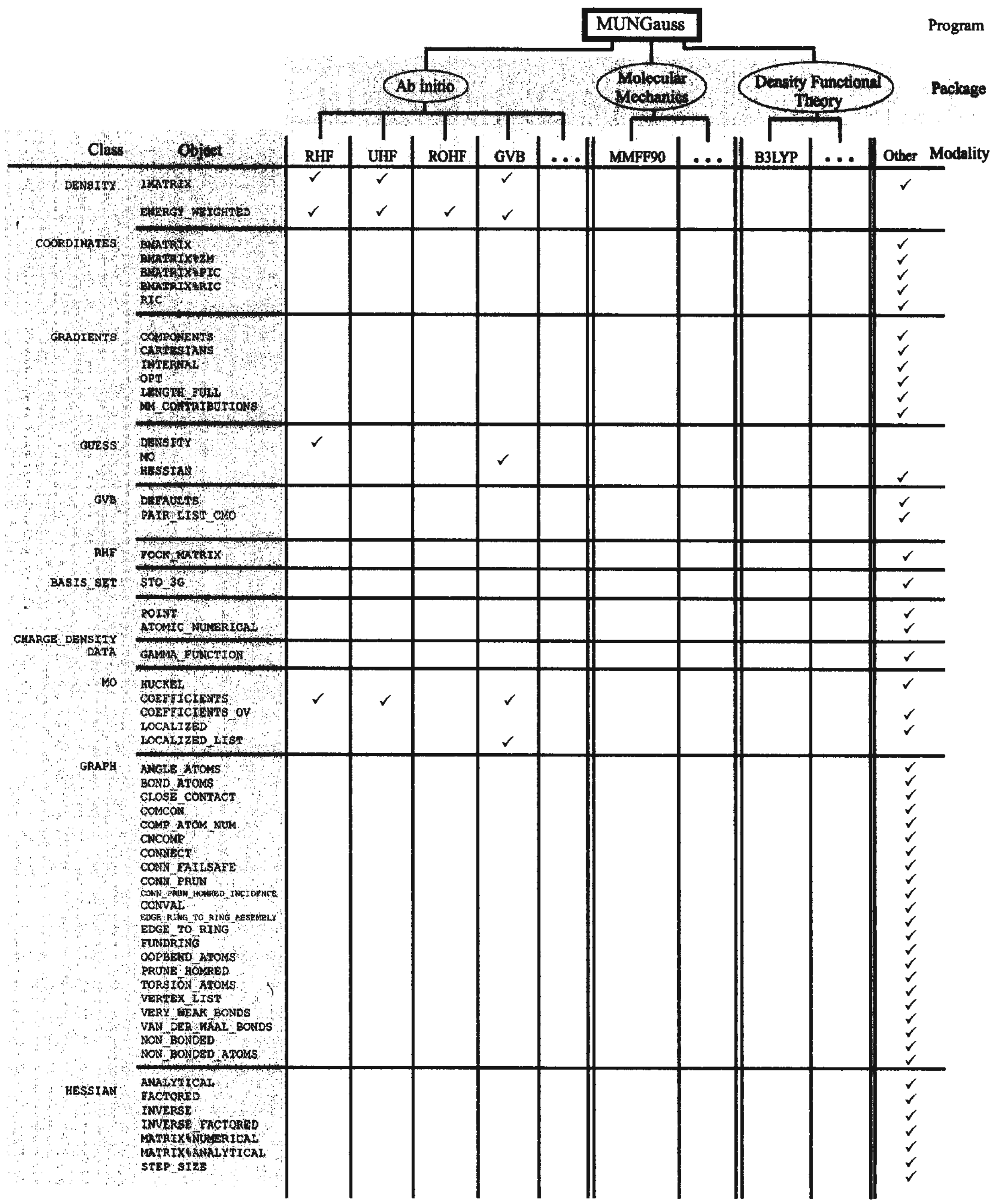
1. the program encompasses everything in the code.

2. A package is defined as a level of theory, be it ab initio, molecular mechanics.
3. The modality division is based on the next division in theory, i.e. wavefunction (RHF, UHF...), force field (MMFF90). Also at the modality level is the coordinate system, which is not shown in the figure.
4. Classes are next, the classes in the code are listed down the left side of the figure.
5. Finally the objects are listed, organized into classes. Note that the checks mean that the object is available at that particular modality.

A summary diagram such as Figure 2.1.2 shows the functionality of the program. A quick look at the figure tells the user exactly what the code can do, it can also help the programmer locate objects that they may need to use, this may become useful in very large codes where an individual programmer may need a piece of information (an object) and may not know if it has been already coded. A look at a summary table could save the re-coding of an object.

2.2 Desired Features of Code

Each project/code will have features specific to that project or code. These features include how users input information, what printing control is required, debugging and timing features, as well as other programmer tools desired. Clearly defining all of these features is vital so that the program design can reflect these requirements. Following is a list of the features desired in MUNgauss.



		MUNGauss										Program
		Ab initio					Molecular Mechanics		Density Functional Theory			Package
Class	Object	RHF	UHF	ROHF	GVB	. . .	MMFF90	. . .	B3LYP	. . .	Other	Modality
DENSITY	1MATRIX	✓	✓		✓						✓	
	ENERGY_WEIGHTED	✓	✓	✓	✓							
COORDINATES	BMATRIX										✓	
	BMATRIX%ZM										✓	
	BMATRIX%PIC										✓	
	BMATRIX&RIC										✓	
	RIC										✓	
GRADIENTS	COMPONENTS										✓	
	CARTESIANS										✓	
	INTERNAL										✓	
	OPT										✓	
	LENGTH_FULL										✓	
	MM_CONTRIBUTIONS										✓	
GUESS	DENSITY	✓										
	MO				✓							
	HESSIAN										✓	
GVB	DEFAULTS										✓	
	PAIR_LIST_CMO										✓	
RHF	FOCK_MATRIX										✓	
BASIS_SET	STO_3G										✓	
CHARGE_DENSITY DATA	POINT										✓	
	ATOMIC_NUMERICAL										✓	
	GAMMA_FUNCTION										✓	
MO	HUCKEL										✓	
	COEFFICIENTS	✓	✓		✓							
	COEFFICIENTS_OV										✓	
	LOCALIZED										✓	
	LOCALIZED_LIST				✓							
GRAPH	ANGLE_ATOMS										✓	
	BOND_ATOMS										✓	
	CLOSE_CONTACT										✓	
	COMCON										✓	
	COMP_ATOM_NUM										✓	
	CNCOMP										✓	
	CONNECT										✓	
	CONN_FAILSAFE										✓	
	CONN_PRUN										✓	
	CONN_PRUN_HOMRED_INCIDENCE										✓	
	CONVAL										✓	
	EDGE_RING_TO_RING_ASSEMBLY										✓	
	EDGE_TO_RING										✓	
	FUNDRING										✓	
	OOPBEND_ATOMS										✓	
	PRUNE_HOMRED										✓	
	TORSION_ATOMS										✓	
	VERTEX_LIST										✓	
	VERY_WEAK_BONDS										✓	
	VAN_DER_WAAL_BONDS										✓	
	NON_BONDED										✓	
	NON_BONDED_ATOMS										✓	
HESSIAN	ANALYTICAL										✓	
	FACTORED										✓	
	INVERSE										✓	
	INVERSE_FACTORED										✓	
	MATRIX%NUMERICAL										✓	
	MATRIX%ANALYTICAL										✓	
	STEP_SIZE										✓	

Figure 2.1.2: Summary table of MUNgauss functionality and organization

1. All the features of "well-written" code presented in Chapter 1 must be considered and balanced. That is: *correctness, robustness, extendibility, reusability, compatibility, efficiency, portability, verifiability, integrity, ease of use,* and *proper documentation*[1].
2. Separate packages. As mentioned MUNgauss contains several different methods for different levels of theory. The final code should be able to be separated into packages (i.e. smaller devoted executables). For example, just an ab initio code, which is compiled and ran separate from the molecular mechanics part of the code.
3. Self-debugging tools. The code should contain tools that help the programmer find errors in the code. This should be a feature which can be turned on and off as needed, and possibly removed from the code when a "number-crunching" or "user" version of the code is released.
4. Timing feature. The ability to determine the time the code takes in a particular routine would be very beneficial in optimization of the code.
5. Trace of execution. Since MUNgauss is route independent, that is it has no set path of execution, the ability to determine the path of execution is very useful in debugging the code.
6. Complete control from the menu. Control over default values and other variables the code uses should be available in the menu (input).
7. Control of output. Similarly, what the code outputs should be exactly what the user wants, with no extras.

8. Portability. The code must be portable, that is it must be able to run on different platforms.

2.3 Choice of Language

The choice of programming language is a very important one. There are many factors which can guide the decision from one language or another. There are practical considerations, which include cost of appropriate compilers/hardware, cost of retraining staff if a new language is chosen, applicability of the language to the desired code. In general one desires a language with a notation that fits the problem, simple to write and learn, powerful operations, etc. Fortran is very good with numerical computations, has many diverse and reliable libraries, and an official standard exists which helps portability[6].

However, the major factor which directed the decision of language in this project was that the old version of MUNgauss was written in FORTRAN 77. While it is possible to convert old FORTRAN 77 codes into C or C++, the process is much more difficult than converting to Fortran 90/95. Fortran 90/95 is based upon FORTRAN 77, and in fact any standard FORTRAN 77 codes should compile and execute with a Fortran 90/95 compiler[2,7,8,9]. This allows for incremental conversion, which means there can always be a working code, even during the conversion process.

Table 2.3.1: A sample of current theoretical chemistry packages, and their programming language

Package	Description	Programming Language
Aces II[10]	Coupled cluster and many body perturbation theory	FORTRAN 77
CADPAC[11]	ab initio quantum chemistry package	FORTRAN 77
GAMESS[12]	general ab initio quantum chemistry package	FORTRAN 77 (with some C for specific unix calls)
Gaussian 98[13]	quantum chemistry package	FORTRAN 77
Q-Chem[14]	ab initio electronic structure program	C++
Spartan[15]	molecular modeling program	FORTRAN 77, C, (C++ GUI)
AMBER[16]	molecular mechanic chemistry package	FORTRAN 77
CHARMM[17]	program for macromolecular simulations	FORTRAN 77
HyperChem[18]	molecular modeling software	C and C++

Most legacy scientific codes are written in FORTRAN 77 so Fortran 90 is an appropriate language to use in the project. Table 2.3.1 contains a list of current theoretical chemistry packages and their programming language. In addition to the back-compatibility issue, Fortran was designed for computation intensive computing, i.e. scientific computing[9]. There are also many other features in Fortran 90 which make it more programmer friendly. Some of these are briefly discussed below.

Fortran 90 is also a migration path to Fortran 2000 and High Performance Fortran (HPF)[19]. HPF was thought to be the "next big thing" in parallel computing in the late 1990's. Although HPF seems to have lost some of its initial steam, there are still many groups pursuing HPF. Converting to Fortran 90 is a good first step for programming in HPF, as many of the construct of HPF are used in Fortran 90. Fortran 95 has many more of the HPF features, however due to the fact that a Fortran 95 compiler was not available on all systems used for this project, Fortran 90 was chosen as the standard.

The next major release in the long history of Fortran will be Fortran 2000[9]. Scheduled for released in 2004[20], Fortran 2000 will eliminate some of the features marked obsolete in Fortran 90 (see table of obsolete features, Table 2.3.2). Most of these features are part of the FORTRAN 77 standard and who's implementation can be more eloquently performed with newer constructs. Fortran 2000 will also introduce true object-oriented programming to the Fortran world. The approach to code design presented in this work, while not object-oriented, is a good first step in that direction.

2.3.1 Drawbacks of FORTRAN 77

Since MUNgauss was written in FORTRAN 77, why change? By today's standards FORTRAN 77 is just outdated. There are many deficiencies in FORTRAN 77 that make using FORTRAN 77 unfavorable. Firstly FORTRAN 77's 'punch card' or 'fixed form' source format. FORTRAN 77 was based on the use of punch cards for programming, each punch card represented one line of code, since there were 72 columns on each card, FORTRAN 77 would ignore any thing past the 72nd column. This restriction is no longer necessary[9].

In addition, FORTRAN 77 lacked dynamic storage. The ability to create a temporary array on-the-fly is not there in FORTRAN 77. Therefore programmers would have to create arrays that were "big enough" for any future problem size. There is also a lack of user defined data structures. Having the ability to create compound objects is very favorable, but not present in FORTRAN 77. FORTRAN 77 did not have explicit recursion. Recursion is a very useful mathematical technique which is missing from FORTRAN 77[2].

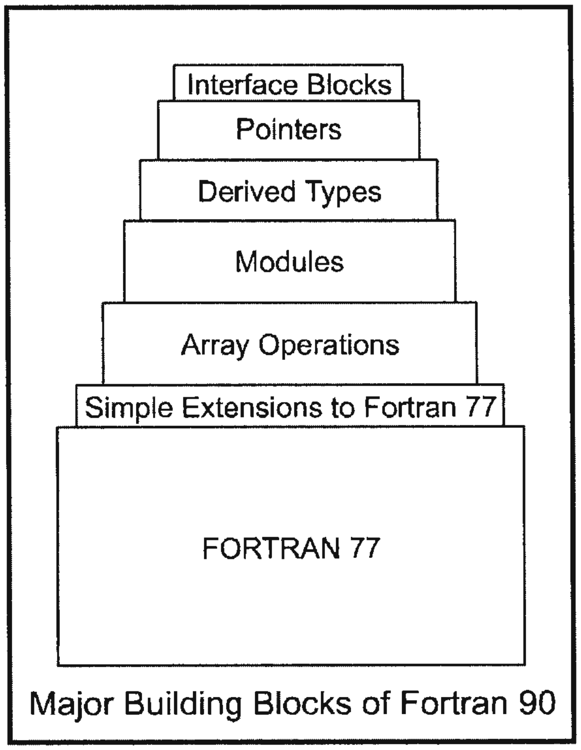
FORTRAN 77 also relied on the COMMON block to give global access to data. This method was often abused and due to lax rules led to users inadvertently doing horrendous things. Also the practice of aliasing an array using an EQUIVALENCE statement is considered unsafe and should no longer be used[9]. These drawbacks of FORTRAN 77 have all been addressed in Fortran 90, making it a much more favorable choice.

2.3.2 Fortran 90 compared to FORTRAN 77

Figure 2.3 illustrates a high-level conceptual view of the composition of Fortran 90. FORTRAN 77 makes up the foundation upon which Fortran 90 is based, approximately two-thirds of the content of the new standard are directly supported by the old standard. Thus much of the data type, such as INTEGER, REAL, and CHARACTER, as well as IF and DO statements remain the bases of most control structures. I/O is virtually unchanged, with some minor enhancements, and the decomposition of large programs is still through the use of subroutines and functions.

However the new features of Fortran 90 are significant and make it easier to work with and more efficient. These features help to bring Fortran 90 more in line with other modern programming languages.

Figure 2.3.1: High-level conceptual view of the makeup of Fortran 90[2]



Source Format

Fortran 90 has a new source format, 'free format'. Free format allows for up to 132 columns per line, no reliance on specific position of special characters, more than one statement per line, in-line comments, both upper and lower case letters (improving readability), object names that can be up to 31 characters, and names that can be punctuated by underscores. All these features of free source format make if very attractive to new programs[2,9].

Dynamic Memory Allocation

Fortran 90 has introduced dynamic storage, which means allocatable arrays and pointers can now be implemented. Arrays can be created on the fly and removed as needed. Also the addition of pointers enables such dynamic data structures as linked lists and trees to be created. Dynamic memory allocation is one of the first features that attract programmers to Fortran 90 over FORTRAN 77, it was also the first feature incorporated in MUNgauss, during the redesign process[9].

User defined data types

User defined data types are now part of Fortran 90. These data types can be constructed by the user using existing types. Defining objects in this way (grouping them together in one data type) is more intuitive and make programming easier and less error prone[2,9].

Recursion

Explicit recursion is now available. The programmer can declare a procedure to be recursive and then it can be used to call itself. This is a very useful approach to many mathematical/scientific problems[9].

Control Constructs

New control constructs have been added to Fortran 90[9]

- `do ... end do`
- `do ... while`
- `exit`, to allow graceful exiting of loops
- `cycle`, for abandoning current iteration
- named control constructs (i.e. labeled); improves code readability
- `select case` control block; more succinct, elegant and efficient than an `if ... elseif ... elseif` block

Internal Procedures

A procedure is allowed to contain a further procedure with local scope. This second procedure can not be accessed from outside the procedure from which it was defined[9].

Modules

A new idea in Fortran 90 is the use of modules. A `module` is a program unit used to package together declaration, subprograms, and definitions of new derived data types. Another program unit simply must 'use' the module to have access to all features in that module[9].

- `blockdata` subprograms are now redundant since a `module` can be used for the same purpose.
- Useful libraries can be written and placed in a module.

2.3.3 Obsolescent Features of Fortran

To protect the investment of all the years of programming preformed using FORTRAN 77, Fortran 90 has included the FORTRAN 77 standard in its entirety. However, many of the features of FORTRAN 77 can now be implemented using more modern features. As a way to allow the language to progress and improve, Fortran 90 has indicated that some of the features of FORTRAN 77 are marked obsolescent[9], which means:

1. They are already redundant in FORTRAN 77 (i.e. they were in FORTRAN 66 or other Fortran standards).
2. Better methods of programming already existed in FORTRAN 77 standard.

Table 2.3.2: Obsolescent features of Fortran[9]

Obsolescent Feature of Fortran	Should be replaced with
Arithmetic `IF` Statement	an equivalent `CASE` or `IF` construct
`ASSIGN` Statement	
`ASSIGNed` `GOTO` Statement	an `IF` statement or a procedure call
`ASSIGNed` `FORMAT Statement`	use a `CHARACTER` string to hold `FORMAT` specifications

Hollerith Format Statements
`      WRITE(*,100)`
`100   FORMAT(17H TITLE OF PROGRAM)`
	use single or double quotes in FORMAT statement:
	`      WRITE(*,100)`
	`100   FORMAT('TITLE OF PROGRAM')`

`PAUSE` statement	use `PRINT` followed by `READ`
`REAL` and `DOUBLE PRECISION` `DO`-loops Variables	use `INTEGER` variables and construct `REAL` or `DOUBLE PRECISION` variables within the loop
Shared `DO`-loop Termination	Use `END DO` instead

3. Programmers should stop using them.

4. It is the intention of the standard's committee to remove these features in subsequent releases of Fortran.

There are also a set of features that have not been marked obsolescent but have been identified as being "undesirable"[9]. These include:

1. Fixed source form
2. Implicit declaration of variables
3. Common blocks
4. Assumed size arrays
5. `EQUIVALENCE` statement
6. `ENTRY` statement
7. Computed GOTO

These features should be avoided in any new code and should be slowly removed from existing code.

Chapter 3

Design Protocols

3.1 Introduction

The key to creating or redesigning a large scientific program that is easily maintained and updated is a strict set of design protocols. This chapter will present such a set. It will include a consistent programming style, consistent use of variables, subroutine, modules, and functions, and consistent documentation practices. These protocols must be followed by every programmer working on the code to ensure others can work on the code with relative ease.

What are design protocols? The use of the term design protocols combines both the visual look of the programs as well as how the tools of Fortran are implemented. The key is consistency. The goal is to design code that can be easily maintained and updated, not only by the primary programmer, but also other programmers. Consistency in the look and use of subroutines, modules, variables, etc., are essential to making this possible.

The process of establishing a strict set of design protocols is not an easy one. The rest of this chapter gives the set of design protocols implemented in MUNgauss. Some may be changed by the users to reflect their personal style (though this is discouraged). Others, however, are believed to be more important to the operation of the code and it is recommend that they be followed as stated. The more important protocols relate to the implementation of the tools of Fortran 90/95, and how functions, subroutines, modules, etc. are used and how they interact.

3.2 Programming Style

Every programmer has their own personal programming style, however for a correctly designed program written by many programmers, having everyone use the same set of guidelines makes the code readable for anyone working on it. Guidelines also helps the person responsible for the entire code (the primary programmer) read and maintain parts of the code written by other programmers. This alleviates the need for everyone working on the code to be intimately knowledgeable about the entire code. Style is a very individual thing and there does need to be some flexibility, however, guidelines are essential for consistency.

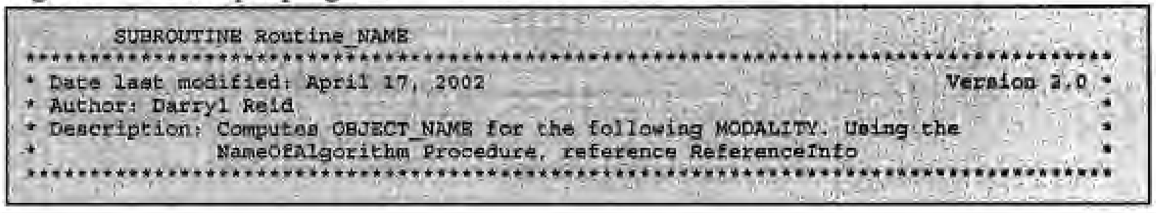
A programming style consists of everything from where variables are defined, how lines of code are aligned (i.e. how indentation is used), and how variables are named, to the use of program headers, comment lines and error messages.

3.2.1 Program headers

The use of a program header is very helpful in writing large code. Following the opening program or subprogram specification there should be a series of comments which describes the program or subprogram, called the program header. The program header should consist of the programmers name, the date the program was written, the version (if applicable), and a description of the routine. This description should include details such as any special algorithms it uses, assumptions the programmer has made, useful references to related information, as well as a directory of the variables passed in or out of the routine. This allows a programmer to read a few lines and know exactly what this subroutine, function, or module does and what it is used for.

In MUNgauss, program headers are to be contained in a box of asterisks. This allows for a sharp visual look that is immediately recognized, and helps to delineate boundaries between subroutines. Figure 3.1 contains an example of a program header.

Figure 3.2.1: Sample program header



```
      SUBROUTINE Routine_NAME
*******************************************************************************
* Date last modified: April 17, 2002                                Version 3.0 *
* Author: Darryl Reid                                                           *
* Description: Computes OBJECT_NAME for the following MODALITY. Using the       *
*              NameOfAlgorithm Procedure, reference ReferenceInfo               *
*******************************************************************************
```

Each subprogram unit should be documented with a program header, this includes (but is not limited to) subroutines, functions, and modules. Information in the program header needs to be kept up to date, or it becomes irrelevant. Also in keeping with the current Fortran 90 standard, changing from a block of asterisks to a block of exclamation points should be considered, since any line that starts with an exclamation point is a comment line in the new "free source form". Fortran 90 does support "fixed source form" for backward compatibility with FORTRAN 77 [2,8,9]. Currently MUNgauss uses fixed source form.

3.2.2 Declarations

The look and order of variable declarations and use of modules and include files is an important feature of a programming style. A comment line is placed before the `USE`, `INCLUDE` or variable declaration which states what follows, i.e. before input scalars are declared in a routine, there is a comment line that states '`* Input Scalars:` '. The visual impact of this style allows for quick and easy location and identification of variables when editing the code. Figure 3.2.2 shows a declaration section of a programming unit with no

Figure 3.2.2: Declaration section, no comments. Not recommended

```
      USE object_based_main
      USE class_name1
      USE class_name2
      implicit none
      include 'osipe_maxdim'
      include 'osipe_message'
      include 'osipe_ios'
      character*(*), intent(IN) :: obj_name
      integer :: Out_integer
      integer lstring
      character(len=132) class
      character(len=132) Object
      character(len=132) modality
      integer get_object_number
```

comments. Figure 3.2.3 shows the recommended style. Both are fine with the Fortran 90 standard, however the second is much more readable by the programmer. In addition, a blank comment line with only an '*' helps the code look cleaner and is always used before a comment line. Again switching to an '!' should be considered.

In addition to those listed in Figure 3.2.3, other separations could be made in the variable declarations. Declarations could be divided into local scalars, local parameters, local arrays, work arrays, input or output arrays, and many more. Comments could also follow a variable definition which states what it represents and how it is (or should be) used, if it would assist the clarity. This is done simply by a statement preceded by an exclamation point. This is

Figure 3.2.3: Declaration style; the comment lines separate and organize different types of declaration for easy identification.

```
* Modules:
       USE object_based_main
       USE class_name1
       USE class_name2

       implicit none
*
* Includes:
      include 'osipe_maxdim'
      include 'osipe_message'
      include 'osipe_ios'
*
* Input scalars:
      character*(*), intent(IN):: obj_name !Object description
*
* Output scalars:
        integer :: Out_integer
*
* Local scalars:
      integer lstring
      character(len=132) class
      character(len=132) Object
      character(len=132) modality
*
* Local functions:
      integer get_object_number
```

also illustrated in Figure 3.2.3.

```
     character*(*), intent(IN):: obj_name !Object description goes here
```

Note that using include files as presented in this example is being phased out. Includes are almost entirely being replaced by modules. The `includes` in Figure 3.2 are remnants of the earlier OSIPE version of MUNgauss. There are still some situations where includes could be used, these are discussed in Section 3.3. The keyword `intent` is used in this figure, `intent` indicates how the variables are being used. `Intent(in)` means the data in those variables will be passed into the subroutine or function, `intent(out)` means data will be passed out of the routine via those variables[2,9,7]. This will be discussed further in Section 3.3.3.

3.2.3 Indentation

Another part of a programming style is the indentation of loops or code segments. Programmers should adhere rigorously to alignment and indentation guidelines. In MUNgauss each line, other than comment lines, is indented seven spaces. An additional two spaces are used within loops (such as `do` loops) or other code segments (i.e. `if` or `case` statements, derived types) to emphasize a relationship between various parts of the program.

Figure 3.2.4: Example of indentation practices; note seven spaces before executable code, and an extra two spaces for each statement sequence within constructs such as `Case` statements and `do` loops.

```
* Sample do loop
       do while (something_is_true)
         Code to change something_is_true to false
       end do

* Sample case construct
       label: select case (selector) !Comment about select case
         case (1)
           Code
         case (2)
           Code
         case default
       end select label
```

Alignment is another visual clue to the execution of the code, and improves readability. FORTRAN 77 required an indent of seven spaces for the execution part of the program. This is known as fixed source form. Since most of the programmers now learning/using Fortran 90 are/were FORTRAN 77 programmers it was decided to keep this protocol; even though it is not required as part of the Fortran 90 standard.

Figure 3.2.4 also show another very important feature of Fortran 90, labeling. Here the select case is labeled so the end of the select case can be easily identified. Such a label helps readability of the code.

3.2.4 Naming and Capitalization

Although Fortran 90/95 is case insensitive, consistency in capitalization is very important. This maintains visual consistency for all occurrences of the variable, module, subroutine, or function name. It is also useful in editing the code. If a variable is used in the exact same manner (same capitalization) then performing a global substitution (throughout the entire code) becomes a much simpler task. Creating a strict set of rules governing naming and capitalization is a difficult task. Below are some of the general rules for naming and capitalization:

1. Assume case sensitive, once a variable is named, always use the same capitalization at every occurrence of the variable.
2. Names should hold some meaning. Names should be long enough to have meaning when read.

```
      Distance = Rate * Time
```

is more meaningful than

```
      D = R * T
```

Do not use "skimpy" abbreviations just to save a few keystrokes. Also the use of underscores (_) between words in a name is encouraged.

```
      USE Program_Manager
```

is better than

```
      USE PrgMan
```

3. When using an indexing variable, give the indexing variable a name that pertains to the application, i.e. do not just call it `i` or `j`, use `Iindex` or `Jindex`.
4. Upper case variable names and lower case the part that would be subscript (e.g. `Iatom, Ibasis`)
5. In MUNgauss subroutines are combined into groups with certain prefixes. These prefixes give programmers a clue of what the subroutine does simply by looking at the name. The prefix to subroutine names should be capitalized. Table 3.1 shows the list of prefixes used in MUNgauss and some examples of their use.

Table 3.2.1: Prefixes used in MUNgauss

Prefix	Meaning	Example

PRT	Prints something to file or screen	`PRT_matrix`
		`PRT_GRAPH_NO_BONDS`
		`PRT_ Object`

BLD	Builds an object	`BLD_GUESS_MO`
		`BLD_Bmatrix`
		`BLD_guess_MO_GVB`

PRG	Program infrastructure routines, dealing with the design and functionality of the code	`PRG_manager`
GET	Usually "gets" a scalar value. Normally a call to a utility	`GET_object_number`
MENU	Routines that deal with the menu/input	`MENU_ints`
I2E, I1E	Routines used in one or two electron integrals	`I2E_SSSS`
MAX	Used in variable names that define a maximum value, normally a parameter	`MAX_atoms`

6. All Fortran keywords (`integer, print, if...then,` etc...) should be lower case. This is contrary to much of the suggested programming practices in the literature[2]. However, scientists are reading code to see the algorithm or method being used. The Fortran code is secondary. Capitalizing the Fortran keywords has the opposite effect of emphasizing exactly the part of the code that should not be emphasized.

The stated naming and capitalization rules work together to create code that is much more readable. The meaningful variable, module, subroutine, and function names give the scientific programmer information on the purpose of (and possibly theory used in) the program. In addition, these rules emphasize the algorithm and minimize the Fortran code, creating a much more readable and understandable program.

3.2.5 Comment Lines

The use of comment lines and comments in general, may be one of the most under utilized practices in all of programming. Comments are extremely important. The current programmer may know that `matx` and `maty` refer to a certain pair of matrices, but the next programmer using/reading their code may not. Adding a simple comment after the declaration, by placing an exclamation point (!) then the comment could save someone the time it takes to trace the code. Also, as stated in the section 3.2.4, the variable name should have some meaning, which will also help in reading the code.

Comment lines are also used to help separate pieces of the code. For example when variable declarations are made, the lines "`* Local scalars`" or "`* Input scalars`" are included before these variables are declared. This is an easy way to tell future programmers where and how the variables are being used. Also a blank line containing just an asterisk (*) in the first column helps to separate blocks of code. Fortran 90 does allow just a blank line; however the '*' is helpful in defining the separation. Again, to keep with Fortran 90 standard the exclamation,'!', point should be considered in future codes.

Comments between blocks of code explaining what those code segments are doing, maybe stating the algorithm used and including a reference, both helps another programmer to read and follow the execution of the code, and gives them information on how they may change or improve the code. That is, knowing the algorithm and reference helps future programmers find newer more efficient algorithms. Though not likely, it is possible to have too many comments. If comments begin to clutter the program, some restraint may be needed.

3.2.6 Error Messages

Error messages are a debuggers friend. At least **meaningful** error messages are. Error messages exist to allow the program to exit gracefully, rather than crash catastrophically. Wherever possible including an error check, to determine if the program is running as it should, and that the data is appropriate, is very important. Just as important as allowing the program it exit gracefully, is to print a message for the potential debugger. All error messages must follow a consistent style and include the following:

a) Where the error occurred, i.e. the routine that detected an error.

b) What variable or object caused the error.

c) A possible solution to the error.

d) Stop command with a brief message of where the error occurred

Figure 3.2.5: Example of an error message.

```
      SUBROUTINE ADD_atom
 .
 .
 .
      if(NatomsW.GT.MAX_ATOM)then
        write(uniout,*)'ERROR> ADD_atom: Too many atoms: must increase MAX_ATOM'
        stop' Error_ADD_atom> Too many atoms: must increase MAX_ATOM'
      end if
 .
 .
 .
      END subroutine ADD_atom
```

Error messages should be contained in `if...then...end` statements as seen in the example in Figure 3.2.5. In the example, the subroutine is named `ADD_atom`, and the error message prints:

```
ERROR> ADD_atom: Too many atoms: must increase MAX_ATOM
```

where `MAX_ATOM` is the maximum number of atoms allowed in the calculations (a parameter set by the module which contains `ADD_atom` subroutine). In this error message the user instantly knows that an error occurred in the `ADD_atom` subroutine, and that a likely solution is to increase the maximum number of atoms allowed.

3.2.7 Other

There should be an option to echo all input. That is, any value given to the code by the user, via the menu in MUNgauss, should be able to be printed as part of the output of the code. This gives the user a record of what they asked for, and having it with the output helps them understand the results. This is very useful in situations where the output seems "fishy", it may be that an input variable was incorrect. It is a way to ensure that the program is reading the input correctly. If the program can reproduce the input then it read it correctly.

Also every piece of output should be labeled. Labels should be clear and concise and include units if applicable. For example:

```
    print (uniout,100), 'Rate =', Rate, '(m/s)  Time = ', Time, ' (s)'
```

produces much more informative output than

```
    print (uniout,*), Rate, Time
```

The use of the asterix (*) is not recommended for printing. This practice is not exactly portable, a line printed with the command '`print *, rate`' on one machine may not be exactly the same as on another machine. This is particularly noticeable when printing real numbers. The same thing applies for the `write` command.

Only information requested by the user should be printed in the output. It can be very confusing if everything the program computes is dumped to the output when all the user wants is a specific piece of information. The product of any scientific code is the output it creates. The program can produce log files of execution, output from certian diagnostics, scientifically relevant results, or intermediate values for debugging and verifying. It is important the the program have controls which allows the user or programmer to select only the output or product they desire.

Output should also be presented in an organized manner. Tables are very useful in this regard. Remember to label the table clearly with each column or row having a label. Headers or titles for tables are another good practice.

3.3 Use of Variables, Modules, Subroutines, and Functions

As with programming style, consistency is the key when using variables, modules, subroutines, functions, etc. The programming practices governing these Fortran constructs are important to state and follow to help with the overall plan of developing well-written code that is easily written and maintained by several programmers. This section outlines some of the programming practices which will help in meeting the programming goals discussed in Section 2.2.

3.3.1 Variables and Constants

In addition to the style considerations discussed in the previous section there are other, more logistical, matters regarding the implementation of variables and constants. In Fortran 90 a variable is any data item that can have its value changed at execution time, while a constant's value can not change during execution.

implicit none

The most important programming practice dealing with variables and constants it the inclusion of `implicit none` in every program or program subunit in Fortran, any variable whose type is not explicitly declared in a type statement will be assigned a type according to an implicit naming convention. That is, any undeclared identifier whose name begins with I, J, K, L, M, or, N or the lowercase equivalents will be typed as integer and all others will be typed as real[8]. This means that failing to declare a variable is not an error. Thus a typing error may not be picked up by the compiler and could end up causing debugging problems.

Fortran 90 provided the `implicit none` statement to cancel this naming convention. Placing `implicit none` at the beginning of the specification part of every program or program subunit requires that the types of all named constants and variables must be specified explicitly in type statements.

Variable Initialization

All variables are initially undefined in Fortran. The variable declaration statement has the following format:

```
    type-specifier, Attribute_List :: variable_name = initial_value
```

(Note: this is not recommended)

While it is possible to initialize a variable in their declarations, this practice is discouraged. For readability purposes, variables should be given initial values in the body of the program, so that when a programmer is reading the code they can clearly see the initialization in the same place as the variable is used. This is in contrast to constants.

Constant Setting

A constant is specified by including a parameter attribute in the declaration of an identifier:

```
    type-specifier, PARAMETER :: name = constant_value
```

Here it is recommended that the programmer set the value of the constant at the declaration stage. As stated previously, constants are to be declared in a separate grouping in the declaration section of the program (and usually in modules, see Section 3.3). Keeping all parameters together and setting their values, allow for quick and easy lookup by the programmer. All the constants are together and their values are right there. This also allows for quick and easy changes to the code if a parameter needs to be changed, the programmer knows where to look, no matter what program or subprogram they are looking at.

Arrays

Fortran 90 has two kinds of array types, *compile-time arrays* and *run-time arrays*[8]. The difference in the two is when the memory is "put aside" or allocated for the array. As the names suggest *compile-time arrays* set aside memory when the program is compiled and can not be changed during the execution of the code, where as *run-time* arrays have their memory allocated during the execution of the code. Run-time arrays are also known as *allocatable* arrays. As mentioned in Section 2.3, it is this dynamic memory allocation that first interests many FORTRAN 77 programmers in Fortran 90. Each type of array has their own programming practices associated with them, which are presented below.

Compile-time arrays

1. Use named constants to dimension compile-time arrays. Using named constants allows for the dimension of the array to be changed more easily (they just have to change the parameter value). It can also indicate why the array is dimensioned to a certain size. For example:

```
integer, parameter :: MAX_number_atoms = 100
integer, DIMENSION(MAX_number_atoms) :: Atom_list
```

   Here it is clear that the array `Atom_list` is dimensioned to the maximum number of atoms allowed in the computation. To change the array dimension only the parameter need be changed. In addition, if there were other arrays that depended on the total number of atoms, they would share the same named constant in their dimension value and thus would all change when the parameter was changed.

2. Specify reasonable sizes for arrays. Overestimating the space need for an array will result in a waste of memory, and could reduce performance of the code.

Run-time arrays

To make a run-time array, that is one that can change shape and size during code execution it must be declared with the key word 'allocatable'.

```
double precision, dimension(:,:), allocatable, save, target :: Bmatrix_ZM
```

1. When allocating space for a run-time array, embed it in an if statement which checks if the array has already been allocated. It is also good practice to check the size of the array to make sure it has been allocated to the correct size. If not then reallocate it.

```
if(.not.allocated(Bmatrix_ZM))then
  allocate (Bmatrix_ZM(NMCOOR,3*NATOMS))
else
  if (size(Bmatrix_ZM,1).ne.NMCOOR)
    allocate (Bmatrix_ZM(NMCOOR,3*NATOMS))
end if
```

Arrays in general

1. Use broadcasting rather than a loop to construct an array where the elements are all the same.

```
A(1:Array_Length) = 0
```

rather than

```
do i = 1, Array_Length
  A(I) = 0
end do
```

2. When using an array always include the dimension. This indicates, when reading the code, that it is an array and not a simple scalar.

```
A(1:Array_Length) = 0
```

is preferred over

```
A = 0
```

though both will produce the same results. Looking at `A(1:Array_Length)` shows that `A` is an array with elements 1 through `Array_Length`.

3.3.2 Modules

A module is a program unit that is used to combine type declarations, subprograms and defined data types. The basic form of a module is:

```
Module Name_of_module
   Variable Declarations
Contains
   Subprogram
End Module Name_of_module
```

The module can be used to package a group of subprograms, data types or functions together so they can be used in other program units[8]. Once written the contents of a module can be made available to another programming unit through the **use statement**.

```
USE Module_name
```

The use statement should be placed at the beginning of the specification part of the program unit.

More important than the syntactical implementation of a module is "how to correctly" use modules in large programs to make them effective in meeting the programming goals. The *module* is a key new feature of Fortran 90 and its implementation can be difficult to do correctly. To help clarify the use of modules three kinds of modules were defined, global, work and object modules.

Global Modules

Global modules are modules which contain information that can be used anywhere in the code. Global modules are used for information that is needed virtually everywhere in the code. Defaults, constants, user defined setting, debugging controls, or information required to track code execution, input/output (filenames, etc.) could all be placed in global modules. The global module can be used in much the same way as COMMON BLOCKs were used in FORTRAN 77 (with more restraint).

Figure 3.3.1: Global Module. This 'objects' module is self contained and can be used virtually everywhere in the code.

```
      MODULE objects
*****************************************************************
*     Date last modified: February 18, 2000Version 2.0          *
*     Author: R.A. Poirier                                      *
*     Description: Debugging tools.                             *
*****************************************************************
      implicit none
*
      integer, parameter :: MAX_objects=1000
      integer :: NObjects
      integer :: ObjNum

      type object_definition
        character(len=132) :: Class
        character(len=132) :: name
        character(len=132) :: modality
        character(len=132) :: routine
        logical :: Current
        logical :: exist
      end type object_definition

      type (object_definition), dimension (MAX_objects) :: Object

* the below should be placed in object_definition when code is ready for it

      logical, dimension(:), allocatable :: Object_debug
      logical, dimension(:), allocatable :: Object_cputiming
*
      END MODULE objects
```

The global module should be a "stand-alone" module. That is to say, it should not CONTAIN any subprogram units.

Object Modules

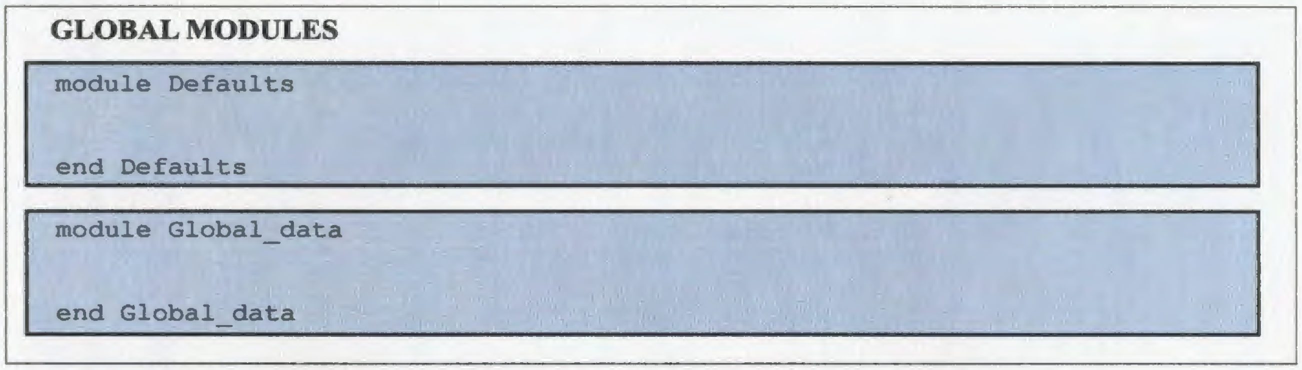
An object module will contain all the information about the object. The object module will be "used" by other routines which require that object. The object module would contain all the declarations of the information / data other routines using that object would require. The object module would also contain all the subroutines needed to build the object.

Work Modules

The work module is a local module that should only be used by routines of the associated object module. The work module will have all the declarations of the variables needed in the routines that build the object, eliminating the need for long argument lists. Any subroutines or functions contained within a work module must be utilities and thus should not build objects. Figure 3.3.3a shows a schematic of the relationship between work and object modules. Note that pseudo-code is used in these diagrams.

The alternative to work modules is to contain all the subroutine and function utilities within the main object building subroutine. This will also remove the need for long argument lists for subroutine or function calls and it tends to keep utilities for a specific object together. A diagram of this implementation can be seen in Figure 3.3.3b.

Modules in MUNgauss

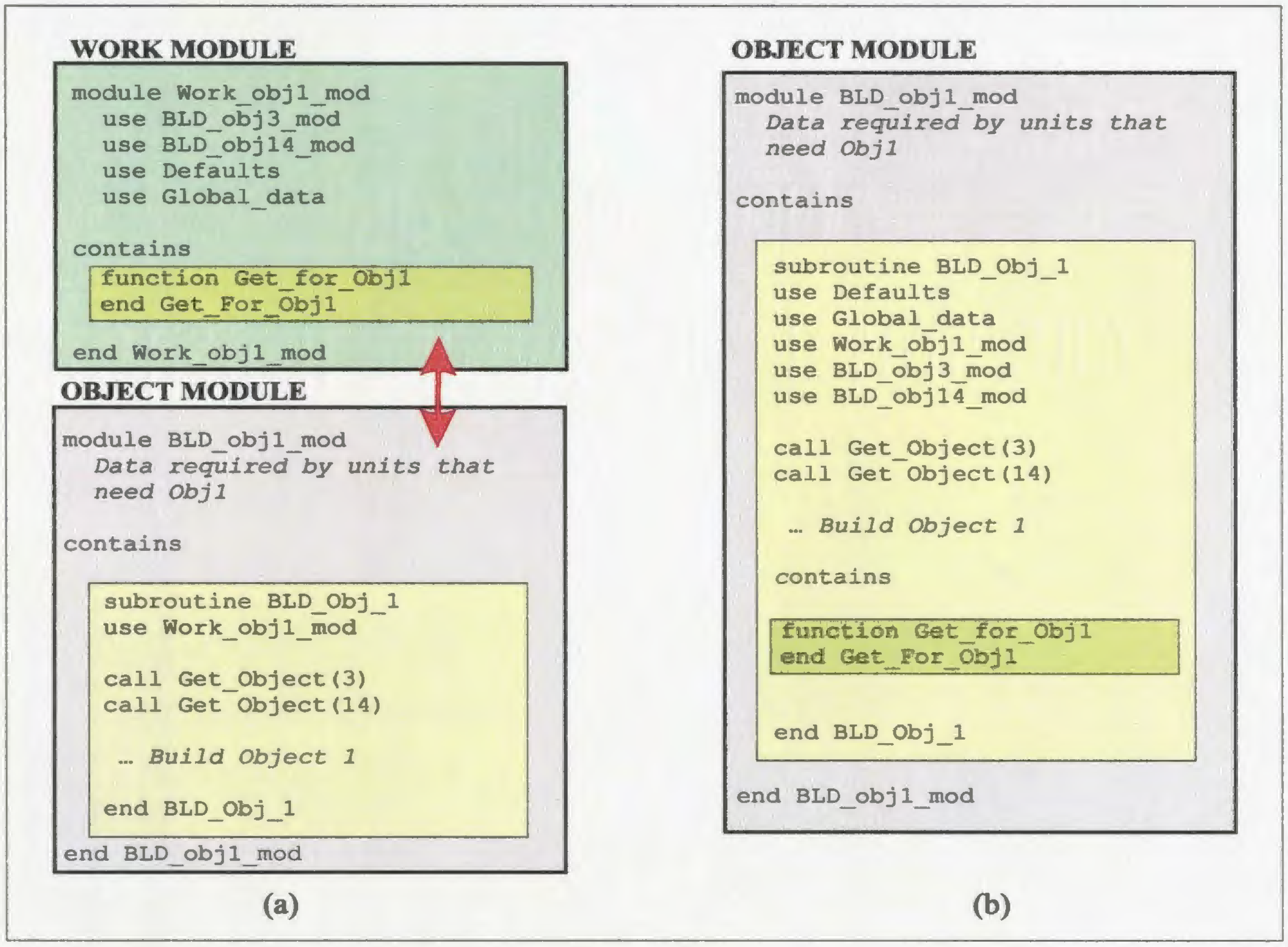


GLOBAL MODULES

```
module Defaults

end Defaults
```

```
module Global_data

end Global_data
```

Figure 3.3.2: Illustration of global modules. Global modules can be used throughout the code, but contain no subroutines or functions.



WORK MODULE

```
module Work_obj1_mod
  use BLD_obj3_mod
  use BLD_obj14_mod
  use Defaults
  use Global_data

contains
  function Get_for_Obj1
  end Get_For_Obj1

end Work_obj1_mod
```

OBJECT MODULE

```
module BLD_obj1_mod
  Data required by units that
  need Obj1

contains

  subroutine BLD_Obj_1
  use Work_obj1_mod

  call Get_Object(3)
  call Get Object(14)

   … Build Object 1

  end BLD_Obj_1
end BLD_obj1_mod
```

(a)

OBJECT MODULE

```
module BLD_obj1_mod
  Data required by units that
  need Obj1

contains

  subroutine BLD_Obj_1
  use Defaults
  use Global_data
  use Work_obj1_mod
  use BLD_obj3_mod
  use BLD_obj14_mod

  call Get_Object(3)
  call Get Object(14)

   … Build Object 1

  contains

   function Get_for_Obj1
   end Get_For_Obj1

  end BLD_Obj_1

end BLD_obj1_mod
```

(b)

Figure 3.3.3: a) Schematic of relationship between object modules and work modules b) Schematic of use of contains withing object building subroutine, this method is preferred over work modules in most cases. NOTE: pseudo-code used.

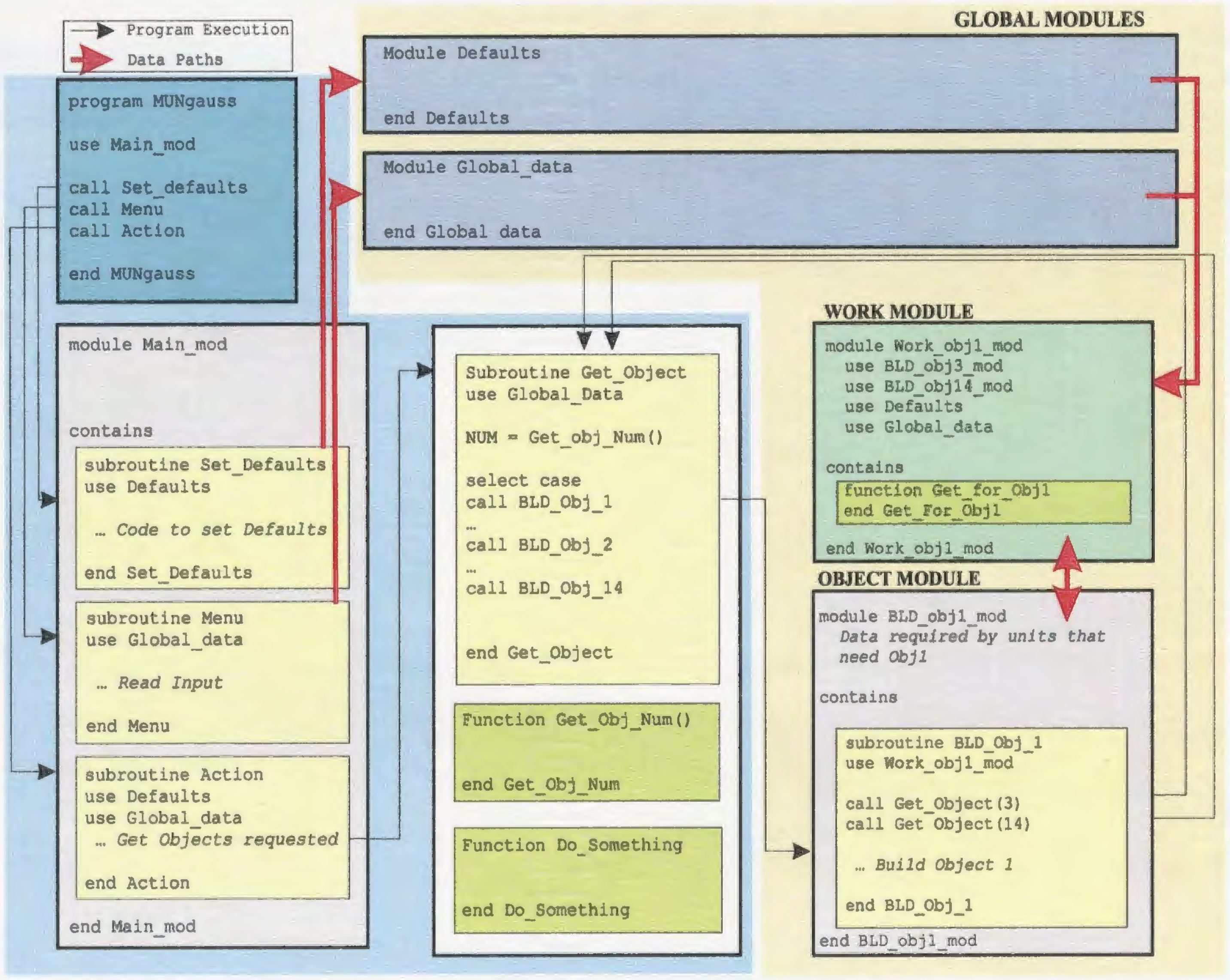
How Object and Work modules work together

The object module and work modules are very closely linked. Every subroutine or function contained in the object module must use the associated work module. This creates a little common block, if you will, of data needed in building that one object. As a consequence, each subprogram unit depends on the work module. To ensure this will compile correctly the work module must be compiled before the object module, either by placing it first in the Makefile, or by placing it in the top of the same file as the object module.

The object module should not have any 'use module' statements other than to use the work module. This keeps the object module clean and readable, allowing a reader of the code to see exactly what data the object provides. This also aids in encapsulation and information hiding. The associated work module will 'use' any module required to build the current object.

Replacing the work module by containing all the subroutine and function utilities, that would be in the work module, in the main object building subroutine does make the main subroutine a little 'messier'. However, this could also be looked at as a documentation source, as all the required modules and utilities will be right there in one file. Basically, there is a tradeoff between modularization and encapsulation. In MUNgauss, the practice of containing functions and subroutines within the main subroutine has become favored.

Use of Modules in MUNgauss



```
program MUNgauss

use Main_mod

call Set_defaults
call Menu
call Action

end MUNgauss
```

```
module Main_mod

contains
  subroutine Set_Defaults
  use Defaults

   … Code to set Defaults

  end Set_Defaults

  subroutine Menu
  use Global_data

   … Read Input

  end Menu

  subroutine Action
  use Defaults
  use Global_data
   … Get Objects requested

  end Action
end Main_mod
```

```
Module Defaults

end Defaults
```

```
Module Global_data

end Global_data
```

```
Subroutine Get_Object
use Global_Data

NUM = Get_obj_Num()

select case
call BLD_Obj_1
…
call BLD_Obj_2
…
call BLD_Obj_14


end Get_Object

Function Get_Obj_Num()

end Get_Obj_Num

Function Do_Something

end Do_Something
```

```
module Work_obj1_mod
  use BLD_obj3_mod
  use BLD_obj14_mod
  use Defaults
  use Global_data

contains
 function Get_for_Obj1
 end Get_For_Obj1
end Work_obj1_mod
```

```
module BLD_obj1_mod
  Data required by units that
  need Obj1

contains

  subroutine BLD_Obj_1
  use Work_obj1_mod

  call Get_Object(3)
  call Get_Object(14)

   … Build Object 1

  end BLD_Obj_1
end BLD_obj1_mod
```

Figure 3.3.4: How modules are used in MUNgauss. Work modules are ‘used’ by object modules and contains all utilities needed in the object module. Features in the blue portion to the left are parts of the program architecture and will be discussed later.

Use of Modules in MUNgauss

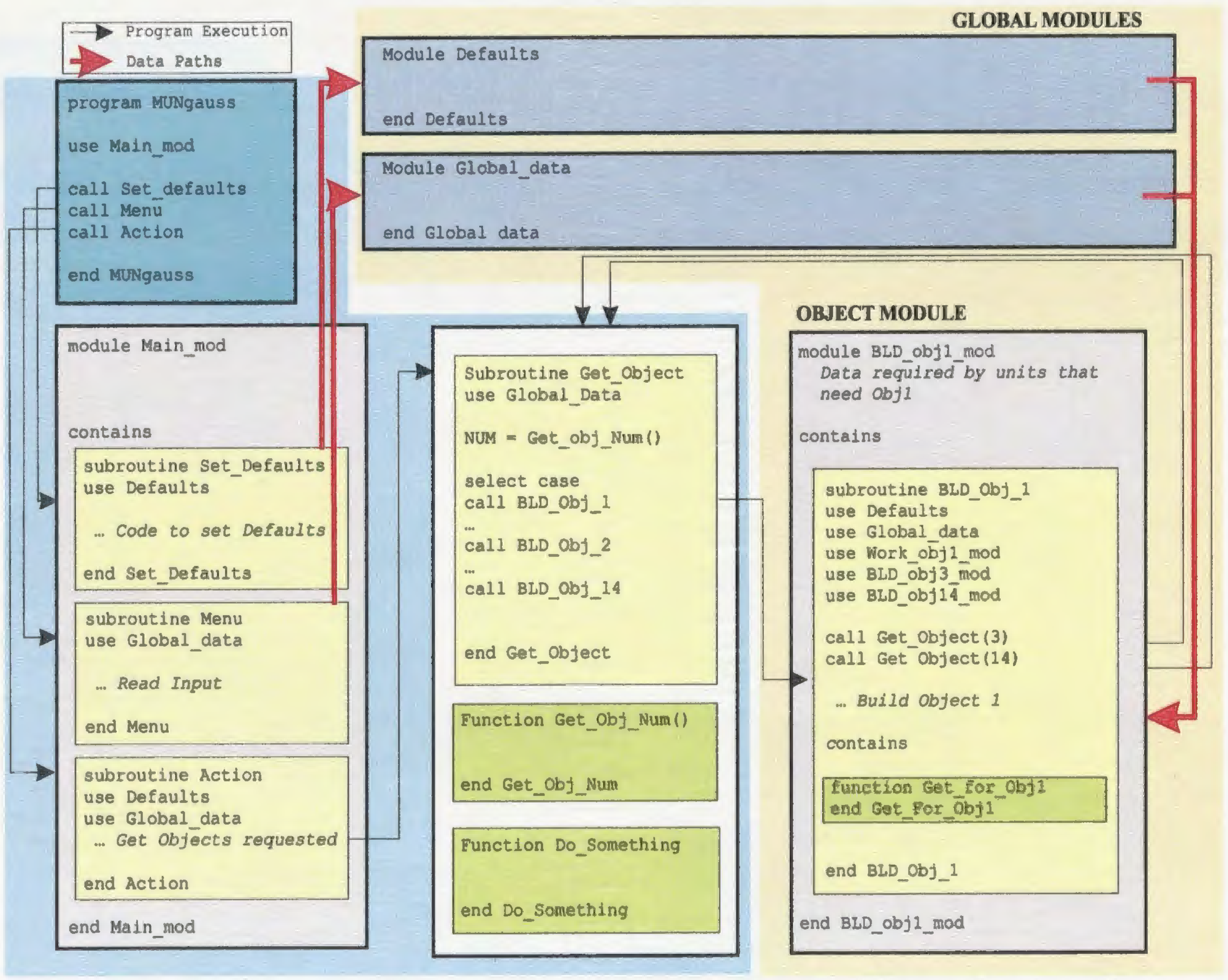


Figure 3.3.5: How modules are used in MUNgauss. Object building routines 'contains' all utilities needed to build the object. Features in the blue portion to the left are parts of the program architecture and will be discussed later.

To illustrate how all the different types of modules work together, Figure 3.3.4 and Figure 3.3.5 diagrams the relationship. Figure 3.3.4 shows the use of work modules, while Figure 3.3.5 shows the containing approach. Both diagrams have different programming features such as the "*get_object*" routine which will be discussed in Chapter 4.

3.3.3 Subroutines

The implementation of the subprogramming unit, subroutines, is relatively straight forward. The basic form of a subroutine is as follows.

```
subroutine Subroutine_Name(formal-argument-list)
specification part
execution part
end subroutine Subroutine_Name
```

Subroutines should conform with all the design protocols defined throughout this chapter. They should contain a header, an informative routine name, the formal argument list should be commented, etc. The features INTENT(IN) and INTENT(OUT) are important to specify in a subroutine[2,7,8]. This prevents inappropriate or incorrect use of a variable. Subroutines can pass information back to the calling program via the arguments in the argument list. Therefore specifying if a particular argument is intended to be used to pass information back or not is important. Subroutines do not have to pass any information to the calling program, they may simply perform some task such as display a menu to the user or print a matrix to a file.

The formal argument list should be well commented as can be seen in the following example. Each variable in the list is followed by a comment describing it's use or contents.

```
subroutine find_one ( Variable1,      !comment
                      Variable2,      !comment
                      Variable3 )     !comment
end subroutine find_one
```

Subroutines can be used to create an object or they can simply be used as utilities.

3.3.4 Functions

Functions are another subprogramming unit that is controlled by another programming unit, as is the case for a subroutine. However, functions pass a single value back to the calling program via the function names, as opposed to a list of arguments like subroutines. Functions, since they return a value, must be given a type declaration[8]. This can be done within the function as:

```
fuction Function_Name(formal-argument-list)
real :: Function_Name
execution part
end function Function_Name
```

or as part of the functions heading

```
real function function_name(formal-argument-list)
```

MUNgauss protocol favors the latter. The type of a function should be specified in the

Figure 3.3.6: Get_object_number Function

```
      recursive integer function get_object_number (obj_name)
******************************************************************************
*     Date last modified: April 3, 2000                        Version 2.0   *
*     Author: R.A. Poirier                                                   *
*     Description: Given an object name, determine the object                *
*         number it corresponds to.                                          *
******************************************************************************
* Modules:
      use program_manager
      use objects

      implicit none
*
* Input scalars:
      character*(*), intent(IN):: obj_name
*
* Local scalars:
      integer Nobj
      logical found
      character(len=132) class, name, modality
*
* Begin:
      call PRG_manager ('enter', 'get_object_number', 'UTILITY')
*
      get_object_number=0
      found=.false.
      Nobj=0
* Extract the class/object name and modality:
      class=obj_name(1:index(obj_name,':')-1)
      name=obj_name(index(obj_name,':')+1:len_trim(Obj_name))
      modality=' '
      if(index(obj_name,'%').ne.0)then
        name=obj_name(index(obj_name,':')+1:index(obj_name,'%')-1)
        modality=obj_name((index(obj_name,'%')+1):len_trim(Obj_name))
      end if
      do while (Nobj.lt.NObjects.and..not.found)
        Nobj=Nobj+1
        if(Object(Nobj)%class.eq.class)then
          if(Object(Nobj)%name.eq.name) then
            if((Object(Nobj)%modality.eq.modality).or.(modality.eq.' ')) then
              get_object_number=Nobj
              found=.true.
            end if
          end if
        end if
      end do ! while
      if(.not.found)then
        write(uniout,'(6a)')'ERROR> get_object_number: Object,',
     .            class(1:len_trim(class)),':', name(1:len_trim(name)),
     .           '%',modality(1:len_trim(modality)),' not found in list'
        write(uniout,'(a)')'Add the Object or make sure the name is correct'
        stop'ERROR> get_object_number: Object not found in list'
      end if
*
*      call PRG_manager ('exit', 'get_object_number', 'UTILITY')
      return
      end
```

function's heading. This again improves readability since the information is all in one line.

Since the arguments of a function do not communicate back to the calling program, they should be declared INTENT(IN).

Figure 3.3.6 shows a function from MUNgauss, get_object_number. This function is key to some of the programming features discussed in Chapter 4. Get_object_number is a utility that determines and objects number by determining the array index for that object. Functions, like get_object_number, are mostly utilities, they normally are not building objects. However some scalar objects may be able to be created using functions, though this is rare.

3.4 Documentation Practices

Documentation means many different things. The term documentation could mean the creation of a document, external to the program which explains how the program works, what it can do, and how it does it. This kind of documentation, or "user manual" would be very useful to users of the code and is something that will need to be created at a later stage in the development. The beginning of this kind of document has already been created when the list of objects and classes were created in Chapter 2. Looking at that list can give the user of MUNgauss an idea of the features (methods) available in MUNgauss. However a complete users manual, external to the program, does not exist for MUNgauss at this time.

However, here documentation refers to *internal documentation.* That is to say, documentation contained within the line of code. This documentation is designed to help the programmers edit and maintain code, and also use the existing code more easily. Another goal of proper internal documentation is to eventually use it to create a user manual. That is have the code write its own documentation. This ultimate goal has not yet been reached but by implementing the consistent documentation practices outlined below, that task should become easier.

Most of the material covered in this section has been mentioned throughout various sections of this (and other) chapters in one way or another. However it is important to emphasize this very important feature of good programming by combining them in one section.

Documenting code is done using comments. The appropriate use of comments throughout a program serves to tell the reader what the code does, how it does it, and how to use it. Comments can also be used to give credit to contributors to the program as well as other references applicable to the code. There are four main areas where comments can be useful in providing documentation.

Headers

The header of a programing unit contains most of the information required about that routine/function. Within the header one should include the author of the code (as well as any other contributors). The date the code was created and or modified. Which version (if applicable) of the program the code is for. As well as a description of the code to follow, what is does, and possibly how to use it. This header could be extracted at a future date and be the basis of a manual for that routine. With this future goal in mind, these headers gain a level of importance and thus should be written in a very useful manner.

Variables

Another form of documentation are comments that following variable declarations, for example,

```
    character*(*), intent(IN):: obj_name !Object description goes here
```

can tell the reader what the variable represents. This practice provides readers with information that otherwise could take a significant amount of time to decipher. These kind of variable comments are especially important when calling a subroutine of function with a group of arguments as was discussed in Section 3.3.3.

Algorithms

Placing a comment before a section of code which tells the reader the algorithm used in the following section of code, when applicable, is a very useful form of documentation. This allows the reader to follow the logic of the program easily by giving them an idea of what is happening. A reference to a paper where the algorithm was taken is also very useful, plus it gives credit to the creators of the algorithm.

Error messages

Error messages are a very important part of a well documented code. When the program reaches and error and stops, the resulting message must tell the user/programmer what caused the error, where the error occurred and give an idea how to fix the problem. Section 3.2.6 discussed the style of error messages in MUNgauss. That style provides the reader with the required information in a clear and concise manner. The example in Section 3.2.6 shows a sample error message output:

```
ERROR> ADD_atom: Too many atoms: must increase MAX_ATOM
```

3.5 Testing

Although testing is not a design protocol it is an essential practice in the design and production of any piece of code. With every change made during the conversion process of any code, it must be tested. MUNgauss has a test suite of problems which are used to verify the correctness of any changes made. Once a routine or function is changed or added the test suite is executed to ensure the results are the same. The test suite has been developed over years to encompass virtually all possible cases. Whenever new functionality is added to MUNgauss, new test cases are created to test those new features and are added to the test suite. Whenever a bug is found during production runs, that run becomes part of the test suite. Testing is an on-going part of code development. Before a version of MUNgauss is deemed "production ready" it must be verified by all the cases in the test suite.

Chapter 4

Code Infrastructure - Overall Program Design

4.1 Introduction

With the plan in place and the protocols defined it is now time to begin coding. This is the most time consuming part of code design, however it is made much more manageable with the proper planning completed. This chapter outlines the implementation of the desired program features described thus far. It is a recipe one can follow when developing large scientific code. The procedure presented in this chapter is the backbone, or infrastructure, upon which virtually any large scientific program can be written.

This infrastructure was designed to make it both easy to implement for the scientific programmer and sufficiently efficient for codes which are computationally demanding. There are some aspects of this design which can still be improved and those shortcomings will be discussed throughout the chapter.

The infrastructure is divided into four main sections, the *object list*, the *get_object*, the *building routine*, and the *program manager* sections. Each of these parts play a key role in satisfying the desired features of well written codes. None of them work independently so the order of description is not necessarily the order they should be written. The development of these routines will be intimately connected and should be preformed simultaneously.

To help prevent some of the problems encountered in designing this infrastructure, some of the approaches attempted and later removed will also be presented, along with reasons why they were deemed unacceptable.

4.2 Object List

In section 2.1.2, table 2.1.2, a list of all the objects used in the code was created. That list can now be used to build an *object list*. One of the key features of this code design is the creation of a hard coded list of all the objects in the code. By creating such a list each object can be assigned a unique number, “the object number” (`ObjNum`). This object number is essential to many of the features described later, such as debugging, timing functions, and tracing. A hard coded object list also gives the code access to information about the object whenever and wherever it is needed.

The object list should contain useful information about each object. It should contain the name of the object, the class and, if appropriate, the modality. As stated in chapter 2, each object will have a unique combination of class, name and modality that will serve to identify the object. In addition to the object name, the object list will also contain other information deemed important throughout the code.

Firstly, the name of the routine which builds the object is included in the object list. This serves two main purposes, self-documentation and error checking. Another programmer can simply look at the object list and find the name of the routine which builds a particular object. This greatly improves the readability of the overall code, and with large codes this is very important. Secondly, the inclusion of the building routine name is used in the debugging and error checking of the code. An explanation of how this works can be found in section 4.5.1.

In addition to the routine name, there are several other characteristics of an object which makes sense to include in the object list. These include information regarding the status of the object, that is, if the object is current (i.e. has already been created and is up to date) or if it exists at all. The information included in the object list can be extended to include options regarding things such as timing routines, debugging routines, etc.

Implementation of the *object list* went through some evolution throughout the process. Below are descriptions of two approaches to implementing an object list. Following these descriptions, pros and cons of each implementation is presented as well as reasons one was chosen over the other.

4.2.1 Implementation of Object List #1.

In the first approach, a series of arrays were declared to hold the required information. The object name and routine name were stored in character arrays, while the other components were stored in logical arrays. To make the information accessible to the rest of the code the declarations were placed in a module. An example of the module containing the declarations is shown below in Figure 4.2.1.

Figure 4.2.1: Objects Module, Implementation #1

```
      MODULE objects
*************************************************************************************
*     Date last modified: February 18, 2000                            Version 2.0  *
*     Author: Darryl Reid                                                           *
*     Description: Object list module.                                              *
*************************************************************************************
      implicit none
*
      integer, parameter :: MAX_objects=1000
      integer :: NObjects
      character(len=132) Object_name(MAX_objects)
      character(len=132) Object_routine(MAX_objects)

      integer ObjNum
      logical, dimension(:), allocatable :: Object_exist
      logical, dimension(:), allocatable :: Object_current
      logical, dimension(:), allocatable :: Object_debug
      logical, dimension(:), allocatable :: Object_cputiming
*
      END MODULE objects
```

In this approach the three parts of the object name (the class, name and modality) were all stored in one array, `Object_name`. The format of this name was `CLASS:Object_Name%MODALITY`. It should also be noted that the parameter `MAX_objects` defined in the module represents the maximum number of objects the entire program can have. It is important to put a check in the routine that creates the object list to ensure that this `MAX_objects` value is not exceeded.

Also included in the object module is the declarations for the object number, `ObjNum`, and the variable that will store the total number of objects in the code, `NObjects`. Both these integers are used throughout the code. The `NObjects`, in this implementation approach, is used to declare the dimension of the logical arrays declared in the module.

The module in Figure 4.2.1 also illustrates some of the design protocols, and some of the features of well written code. Here it can be seen that the program header is used which clearly identifies the date the routine/module was last edited, the author and a description of what it subprogram/unit is. Also, remember that one of the features of well written code is that it was readable. It can be seen here that the variables clearly state what is being stored in them, e.g. `Object_current`, without any further comments or documentation anyone can see that this logical array will tell if the object is current or not. Variable naming in this fashion greatly improves a codes readability.

Once the object module is created, object names need to be put in the `Object_name` array. This is done by creating a `Build_Object_List` routine which increments a counter, `Nobjects`, then stores the object name in `Object_name(Nobjects)`. Similarly, it will store the routine name in `Object_routine(Nobjects)`. A short example of this building routine is seen in Figure 4.2.2 (note this is only a piece of the building routine).

A "Dummy Class" has been included and serves as a template for programmers looking to add new objects to the code. It also places a unique character in the last position of the arrays (a '?') which may be used in some error checking applications. Also the list is organized by classes, in alphabetical order which makes it simple to code and easy to add new objects. Just locate or create a class and place the new object in the middle of the list.

It can also be seen that in this implementation, the logical arrays, declared in the object module, are allocated at the end of the building routine. At this point the total number of

Figure 4.2.2: Build Object List Subroutine, Implementation #1

```
      subroutine BLD_object_list
************************************************************************************
*       Date last modified June 23, 2000                                          *
*       Author: Darryl Reid                                                       *
*       Description:  Builds a complete list of all the objects which can be      *
*             created in MUNGAUSS.  When adding a class or object within a class  *
*             ensure it is done in alphabetical order                             *
************************************************************************************
* Modules:
      use program_manager
      use objects

      implicit none
*
* Begin:
      call PRG_manager ('enter', 'BLD_OBJECT_LIST', 'UTILITY')

              .
              .
              .

* Class PROGRAM
      Nobjects = Nobjects + 1
      Object_name(Nobjects) = 'PROGRAM:OBJECTS_CREATED'
      Object_routine(Nobjects) = 'PRT_objects_created'
*
      Nobjects = Nobjects + 1
      Object_name(Nobjects) = 'PROGRAM:OBJECTS_STATUS'
      Object_routine(Nobjects) = 'PRT_objects_status'
*
* Dummy Class
      Nobjects = Nobjects + 1
      Object_name(Nobjects) = '?:?'
      Object_routine(Nobjects) = '?'

              .
              .
              .

      allocate (Object_exist(Nobjects), Object_current(Nobjects),
Object_debug(Nobjects),
    .           Object_cputiming(Nobjects))

      call PRG_manager ('exit', 'BLD_OBJECT_LIST', 'UTILITY')
      return
*
      end subroutine BLD_object_list
```

objects in the code is known, `NObjects`. Therefore the logical arrays can be allocated to exactly the correct size, thus saving memory use. This is just one of the advantage of this implementation approach. Following the description of the second approach, both approaches will be compared and a conclusion will be drawn.

4.2.2 Implementation of Object List #2.

In this implementation the new feature in Fortran 90 of derived types is used. In this case the *object list* is a large array of derived types. In this implementation the `objects` module contains the definition of the `type object_definition`. Here the different components described above, the class, object name, modality, routine name, current, exist, etc. have been placed within the type definition. Figure 4.2.3 shows this module implementation.

Figure 4.2.3: Object Module, Implementation #2

```
      MODULE objects
*************************************************************************
*     Date last modified: June 18, 2001                    Version 2.0  *
*     Author: Darryl Reid                                               *
*     Description: Object List Module.                                  *
*************************************************************************
      implicit none
*
      integer, parameter :: MAX_objects=1000
      integer :: NObjects
      integer :: ObjNum

      type object_definition
        character(len=132) :: Class
        character(len=132) :: name
        character(len=132) :: modality
        character(len=132) :: routine.
        logical :: Current
        logical :: exist
      end type object_definition

      type (object_definition), dimension (MAX_objects) :: Object

* the following should be placed in object_definition when code is ready for it

      logical, dimension(:), allocatable :: Object_debug
      logical, dimension(:), allocatable :: Object_cputiming
*
      END MODULE objects
```

In accordance with the design protocols of chapter 3 it can be seen that the type definition is contained in a module, this will be a global module and will be available to every routine in the program. `MAX_objects=1000` is again the maximum number of objects the code can have. This is the size that array `Object` will be dimensioned, which could cause some memory to be wasted. The current version of MUNgauss contains around 150 objects however space is being allocated for 1000. This is for developmental purposes only, the value should and could be reduced by just changing that one parameter to a more appropriate size. In the future a better way of determining the amount of space required for the object list would be to copy the information contained in the `Object` array to a new array which has been dimensioned to the `Nobjects` value, upon completion of the `BLD_ObjectList`.

The actual construction of the object list is again performed in a `BLD_ObjectList` routine which simply assigns values to the derived type for each object. As can be seen in Figure 4.2.4 the index of this array is unique for each object and thus becomes the unique object number. As in approach #1 the dummy class is included as a template and contains the unique character, "?", which could be used in error checking applications.

It is important to point out that in this implementation the object name is divided into its components, class, name and modality. This gives access to any portion of the name without having to preform any string manipulations (as is required in approach #1). Access to the

Figure 4.2.4: Build Object List Subroutine, Implementation #2

```
      SUBROUTINE BLD_object_list
***************************************************************************
*       Date last modified June 23, 2000                                  *
*       Author: Darryl Reid                                               *
*       Description:  Builds a complete list of all the objects which     *
*       can be created in MUNGAUSS                                        *
*       When adding a class or object within a class ensure it is done    *
*       in alphabetical order                                             *
***************************************************************************
* Modules:
      USE program_manager
      USE objects

      implicit none
*
* Begin:
      call PRG_manager ('enter', 'BLD_OBJECT_LIST', 'UTILITY')
      Ldebug=Local_Debug
*
      Do Iobject = 1,Max_objects
        Object(IObject)%modality = 'other'
      end do
      Nobjects = 0
*
* Class DENSITY
      Nobjects = Nobjects + 1
      Object(Nobjects)%class = 'DENSITY'
      Object(Nobjects)%name = '1MATRIX'
      Object(Nobjects)%modality = 'WAVEFUNCTION'
      Object(Nobjects)%routine = 'BLD_density_1MATRIX'

*     ... The Rest of the List goes here ...

* Class PROGRAM
      Nobjects = Nobjects + 1
      Object(Nobjects)%class = 'PROGRAM'
      Object(Nobjects)%name = 'OBJECTS_CREATED'
      Object(Nobjects)%routine = 'PRT_objects_created'
*
      Nobjects = Nobjects + 1
      Object(Nobjects)%class = 'PROGRAM'
      Object(Nobjects)%name = 'OBJECTS_STATUS'
      Object(Nobjects)%routine = 'PRT_objects_status'
*
*
* Dummy Class
      Nobjects = Nobjects + 1
      Object(Nobjects)%class = '?'
      Object(Nobjects)%name = '?'
      Object(Nobjects)%modality = '?'
      Object(Nobjects)%routine = '?'
```

information about the objects are made by appending the variable of interest to the derived type with a "%" between, e.g. `Object(Nobjects)%class = 'DENSITY'`, where `Nobjects` is the object number of the desired object. Again this illustrates the attempt to make the code self-documenting and readable. Without any other information it is clear that the class of this object is DENSITY.

It should be noted that at the beginning of the `BLD_object_list` routine all the modalities are set to the default value 'other'. This is just to ensure that every object has a modality associated with it, the importance of this will be seen in the discussion of *get object* (Section 4.3). The default value for the modality can be over written by assigning a new value as in done in the example given, `Object(Nobjects)%modality = 'WAVEFUNCTION'`.

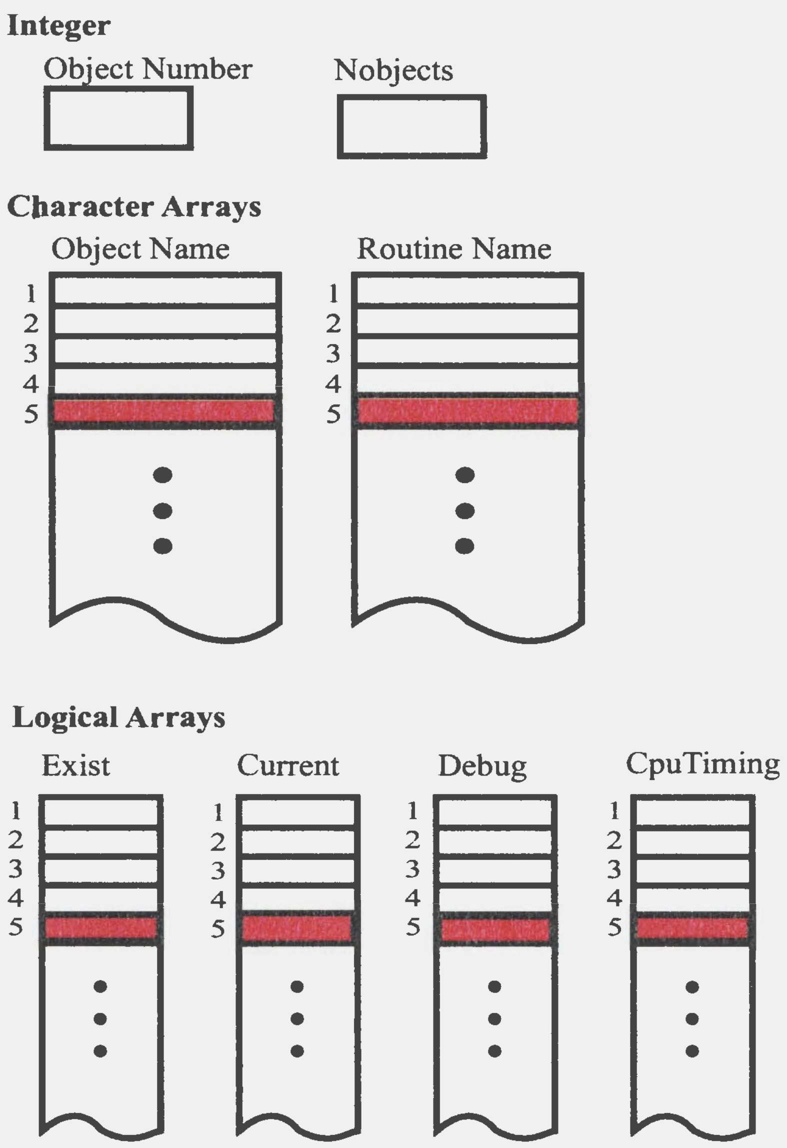
4.2.3 Comparison of Object List Implementations

The initial attempt of implementing object list was a normal Fortran 77 approach of creating separate arrays for each piece of data required. This implementation worked and allowed us to develop other sections of the code (to be described throughout this chapter). Most of the work done on converting and updating MUNgauss was done using this first implementation. It did have many good features, it allowed us to have objects with unique object numbers, gave us access to information about an object from anywhere in the code and gave us an easy and manageable way of adding new objects to the code.

However, once some of the new features of Fortran 90 were better understood, this implementation was re-examined. It was decided that a large array of derived types could be used in a similar manner as the groups of arrays created in the first implementation. The derived type implementation has several favorable characteristics:

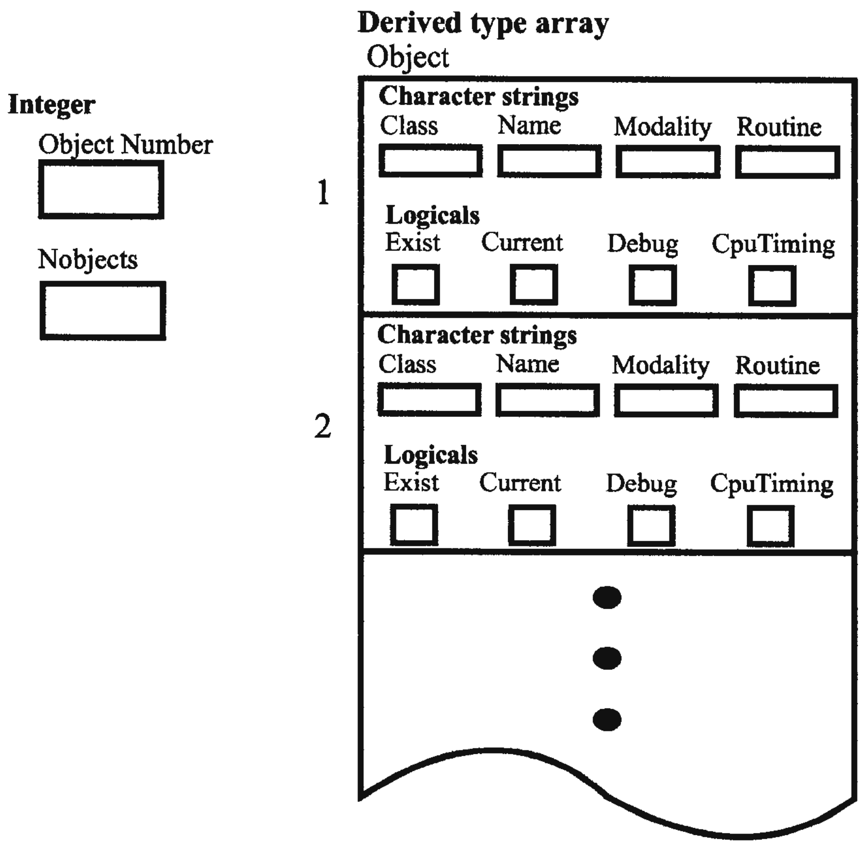
1.1 It provided greater *encapsulation.* Encapsulation is a very good characteristic of a program. Encapsulation means that things are kept together that belong together. Implementing objects as derived type allowed all the properties of the object to be together (from the programmers point of view). Figure 4.2.6 illustrates this encapsulation.

1.2 Access to all parts of the object equally. Since in the second implementation the class, object name, and modality were all separate variables within the derived type, the program has equal access to any part, without any string manipulations. Rather than having to break down the long object name, `class:object_name%modality`, the program can now simply access the piece it requires, `object(objNum)%class`, `object(objNum)%name`, or `object(objNum)%modality`.



```
      module objects
*************************************************************************************************
*     Date last modified: February 18, 2000                                     Version 2.0   *
*     Author: Darryl Reid                                                                      *
*     Description: Object list module.                                                         *
*************************************************************************************************
      implicit none
*
      integer, parameter :: MAX_objects=1000
      integer :: NObjects
      character(len=132) Object_name(MAX_objects)
      character(len=132) Object_routine(MAX_objects)

      integer ObjNum
      logical, dimension(:), allocatable :: Object_exist
      logical, dimension(:), allocatable :: Object_current
      logical, dimension(:), allocatable :: Object_debug
      logical, dimension(:), allocatable :: Object_cputiming
*
      END MODULE objects
```

Figure 4.2.5: Schematic representation of memory allocation of object list in implementation #1.



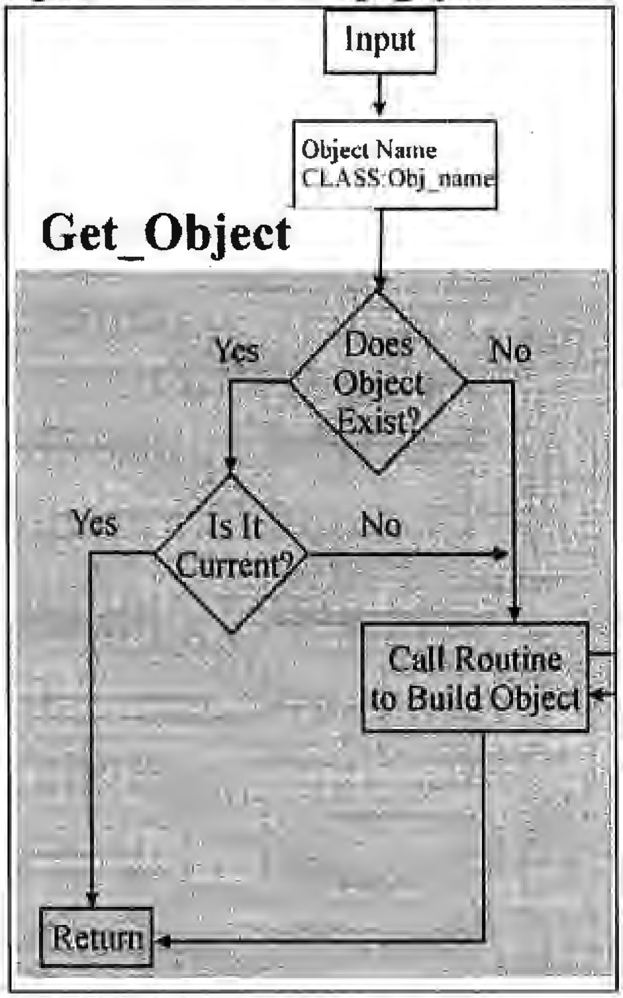
```
      MODULE objects
*************************************************************************
*     Date last modified: June 18, 2001                       Version 2.0  *
*     Author: Darryl Reid                                                  *
*     Description: Object List Module.                                     *
*************************************************************************
      implicit none
*
      integer, parameter :: MAX_objects=1000
      integer :: NObjects
      integer :: ObjNum

      type object_definition
        character(len=132) :: Class
        character(len=132) :: name
        character(len=132) :: modality
        character(len=132) :: routine
        logical :: Current
        logical :: exist
        logical :: Object_debug
        logical :: Object_cputiming
*
      end type object_definition

      type (object_definition), dimension (MAX_objects) :: Object

      END MODULE objects
```

Figure 4.2.6: Schematic of the encapsulation created by using a derived type to implement the object list, implementation #2.

4.3 *Get_object* Routine

The next major section of the code's infrastructure is the *get_object* routine. When a section of the code requires an object, the code must know how to call the appropriate routine to build that object. This is the purpose of the *get_object* routine. The *get_object* routine is

Figure 4.3.1: Flow chart of *get_object* routine



invoked whenever any routine requires an object. The calling routine will give *get_object* the name of the required object and *get_object* will, if necessary, call the routine that builds that object.

Figure 4.3.1 shows a general flowchart of *get_object*, there were two implementations of this routine considered during the development process, however, the main function remains the same. When presented with an object name, *get_object* first has to determine the object number of that object and set it for the rest of the code. With this information, *get_object,* can check the status of the object, that is, if the object exists (has already been created) and if it is current (still valid and useable) from the values stored in the object. If an object exists and is current then *get_object* simply returns. If either value is false, then the object must be built or updated. In either case the object's building routine needs to be called. Creating an object's building routine will be presented in section 4.4.

The manner in which *get_object* determines which building routine to call differs in each method described in this section. The *get_object* routine is the implementation of the organization of the code presented in section 2.1.3. Remember that the code has been organized into objects that have associated names, classes and modalities. *Get_object* uses these characteristics to call the appropriate building routine. The way *get_object* performs this task impacts greatly on the overall organization of the code. Two organization schemes were considered and each will be discussed. The implementations do correlate with the two implementations of the *build object list* of section 4.2 to a degree. However, there are also differences in the organization scheme that are beyond the differences in the *build object list* implementations.

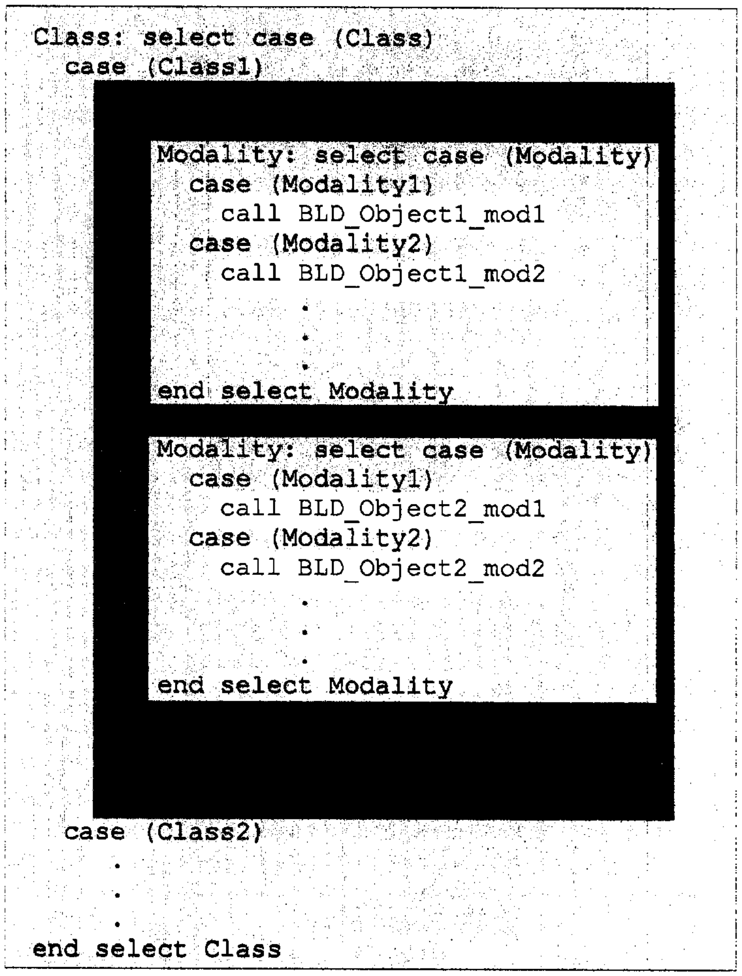
4.3.1 *Get_object* implementation #1.

The basic way to determine which routine to call is through a series of nested select cases, each with many options. The select cases are based on the total object name (which is passed to *get_object* by the calling routine). However, the object name consists of three parts, class, object name, and modality. In the first approach the select case was divided along class lines, then object name, then modality. Figure 4.3.2 shows a pseudo-code schematic of implementation number one's approach to the series of nested select cases. As can be seen from Figure 4.3.2, the list of options can become quite long and the file containing all options would quickly grow out of control. Therefore each portion of the nest was divided into smaller parts to make them manageable.

Figure 4.3.3 shows a portion of the top select case of the *get_object* routine. This select case is based on the class of the object and, as can be seen in the figure, the is a call to different routine for each class. This allowed all the objects which belonged to one class to have there calls in one place.

As mentioned in Section 4.3, *get_object* must first determine the object number of the object requested by the calling routine. This is accomplished by a call to the function `get_object_number()` which returns the object number of the object to be built. The

Figure 4.3.2: Schematic of series of nested select cases for *get_object* implementation #1.



```
Class: select case (Class)
  case (Class1)
        Modality: select case (Modality)
          case (Modality1)
            call BLD_Object1_mod1
          case (Modality2)
            call BLD_Object1_mod2
                   .
                   .
                   .
        end select Modality

        Modality: select case (Modality)
          case (Modality1)
            call BLD_Object2_mod1
          case (Modality2)
            call BLD_Object2_mod2
                   .
                   .
                   .
        end select Modality
  case (Class2)
     .
     .
     .
end select Class
```

implementation of this function will be described in detail in Section 4.3.2. Next the status of the object is checked, that is to say if the object exists and is current. If the object is current then it does not need to be rebuilt and therefore the routine simply returns.

Figure 4.3.3: *Get_Object* Routine, Implementation #1

```
      recursive subroutine get_object (obj_name)
***********************************************************************************
*     Date last modified: April 3, 2000                              Version 2.0  *
*     Author: R.A. Poirier                                                        *
*     Desciption: Given an object name call the appropriate routine               *
***********************************************************************************
* Modules:
      USE program_manager
      USE objects

      implicit none
*
* Input scalars:
      character*(*), intent(IN):: obj_name
*
* Local scalars:
      character(len=132) class
      character(len=132) Object
      character(len=132) modality
*
* Local functions:
      integer get_object_number
*
* Begin:
      call PRG_manager ('enter', 'GET_OBJECT', 'UTILITY')
*
      ObjNum=get_object_number(obj_name//' ')
*
* Object will exist and will be current:
      if(Object_current(ObjNum))then
        call PRG_manager ('exit', 'GET_OBJECT', 'UTILITY')
        RETURN
      end if

      Object_exist(ObjNum)=.true.
      Object_current(ObjNum)=.true.
*
* Extract the class/object name and modality:
      class=obj_name(1:index(obj_name,':')-1)
      Object=obj_name(index(obj_name,':')+1:len_trim(Obj_name))
      modality=' '
      if(index(obj_name,'%').ne.0)then
        modality=obj_name(index(obj_name,'%'):len_trim(Obj_name))
      end if
*
* Check for class
      CLASS_name: select case (class)

      case ('COORDINATES')
        call COORDINATES_objects (class, Object)

      case ('DEFAULTS')
        call DEFAULTS_objects (class, Object)

      case ('DENSITY')
        call DENSITY_objects (class, Object)

     ... Rest of Select Case here ...

      case default
        write(uniout,*)'No such class "',class(1:len_trim(class)),'" for object
"',Object(1:len_trim(Object)),'"'
        stop'No such class'
      end select CLASS_name
*
      call PRG_manager ('exit', 'GET_OBJECT', 'UTILITY')
      return
      end
```

Otherwise, the object will need to be built. Therefore the appropriate building routine must be invoked, which is accomplished by the series of nested select cases.

The series of nested select cases can be looked at as a tree, with each option being another branch. The tree created by the series of select cases is the code implementation of the organization scheme mentioned above. In implementation #1, the first select case (the one found directly in *get_object* and can be seen in Figure 4.3.3) is based on the class of the object. However before this select case can occur the class must be extracted from the object name. The lines following the comment line

```
    * Extract the class/object name and modality:
```

in Figure 4.3.2, performs this extraction (along with the extraction of the object name and modality). With the different parts of the object name extracted the code can now execute the select case to determine which routine to call. At this level the select case calls a "class associated" routine (i.e. DENSITY_objects) which determines which object building routine to call. A "class associated" routine is a second level of select cases that are based on the object name. In theory there would be a third level of the select case which would contain the modalities (as is seen in the schematic in Figure 4.3.2), however in practice, this was not implemented, since the number of objects with modalities were so small, they were incorporated with the object name select cases. Figure 4.3.4, shows an example of one of these "class-associated" routines. Again, this simply consists of a select case based on the object name. Note that when modality applies they are also included in this select case. The plan was to separate the modality part into another level of calls, but was never implemented, due to the overhead (mainly in programmer time) associated with it, for such a small number of cases.

Figure 4.3.4: Example of select case based on object name, contained in a "class associated" subroutine

```
      subroutine DENSITY_objects (class, Object)
***************************************************************************
*     Date last modified: October 5, 2000                     Version 2.0 *
*     Author: R.A. Poirier                                                *
*     Description: GVB objects.                                           *
***************************************************************************
* Modules:
      use program_manager

      implicit none
*
* Input scalar:
      character*(*) class,Object
*
* Begin:
      call PRG_manager ('enter', 'DENSITY_objects', 'DENSITY:')
*
      select case (Object)
      case ('1MATRIX')
        call BLD_density_1MATRIX
      case ('1MATRIX%RHF')
        call DENSITY_1MATRIX_RHF
      case ('1MATRIX%GVB')
        call DENSITY_1MATRIX_GVB
      case ('1MATRIX%UHF')
        call DENSITY_1MATRIX_UHF
      case ('ENERGY_WEIGHTED')
        call BLD_Energy_weighted_density
      case ('ENERGY_WEIGHTED%RHF')
        call DENSITY_Eweighted_RHF
      case ('ENERGY_WEIGHTED%UHF')
        call DENSITY_Eweighted_UHF
      case ('ENERGY_WEIGHTED%GVB')
        call DENSITY_Eweighted_GSCF
      case ('ENERGY_WEIGHTED%ROHF')
        call DENSITY_Eweighted_GSCF
      case default
          write(uniout,*)'No such object "',Object(1:len_trim(Object)),
     .                   '" for class "',class(1:len_trim(class)),'"'
          stop'No such object'
        end select
*
      call PRG_manager ('exit', 'DENSITY_objects', 'UTILITY')
      RETURN
      END subroutine DENSITY_objects
```

Figure 4.3.5: Function which determines the object number

```
            .
            .
            .
      implicit none
*
* Includes:
      include 'osipe_maxdim'
      include 'osipe_message'
      include 'osipe_ios'
*
* Input scalars:
      character*(*), intent(IN):: obj_name
*
* Local scalars:
      integer Nobj
      logical found
*
* Begin:
      call PRG_manager ('enter', 'get_object_number', 'UTILITY')
*
      get_object_number=0
      found=.false.
      Nobj=0
*
      do while (Nobj.lt.NObjects.and..not.found)
        Nobj=Nobj+1
        if(obj_name(1:len_trim(obj_name)).eq.Object_name(Nobj)
     .             (1:len_trim(Object_name(Nobj))))then
          get_object_number=Nobj
          found=.true.
        end if
      end do ! while

      if(.not.found)then
        write(uniout,*)'ERROR> get_object_number: Object,
     .       ',obj_name(1:len_trim(obj_name)),' not found in list'
        write(uniout,*)'Add the Object or make sure the name is correct'
        stop'ERROR> get_object_number: Object not found in list'
      end if
*
      call PRG_manager ('exit', 'get_object_number', 'UTILITY')
      return
      end
```

It can be seen in Figure 4.3.4, that the modality is included in the `Object` part of the object name. Thus the appropriate building routine is called based on all three parts of the object name. Section 4.4 will discuss the object building routines.

Also note that the class is passed into the "class associated" routine, this variable is never used in the select case (since all objects in that routine are of the same class) but it is used in the error message printed if the requested `object%modality` is not found in the select case. This illustrates two of the design protocols described in Chapter 3, that every select case must have a default value that prints an error message that is useful and helps identify exactly where the problem is.

As mentioned, the series of nested select cases are the implementation of the code organization. Figure 4.3.6 shows the resulting tree structure created by the order of select cases in implementation #1. This figure shows how the program is broken up. Each horizontal line represents another select case. Each block will be in its own routine, ie, O1, O2, O3, and O4 will all be cases in the "class associated" routine of class 1 (C1). Once the program works its way through the select cases it will end up at the bottom of the tree (the modalities) and will then call the object building routine.

Looking at the code organization in a structure like that in Figure 4.3.6 gave incite into ways it could be improved. This will be discussed when both implementation #1 and #2 are compared in Section 4.3.3.

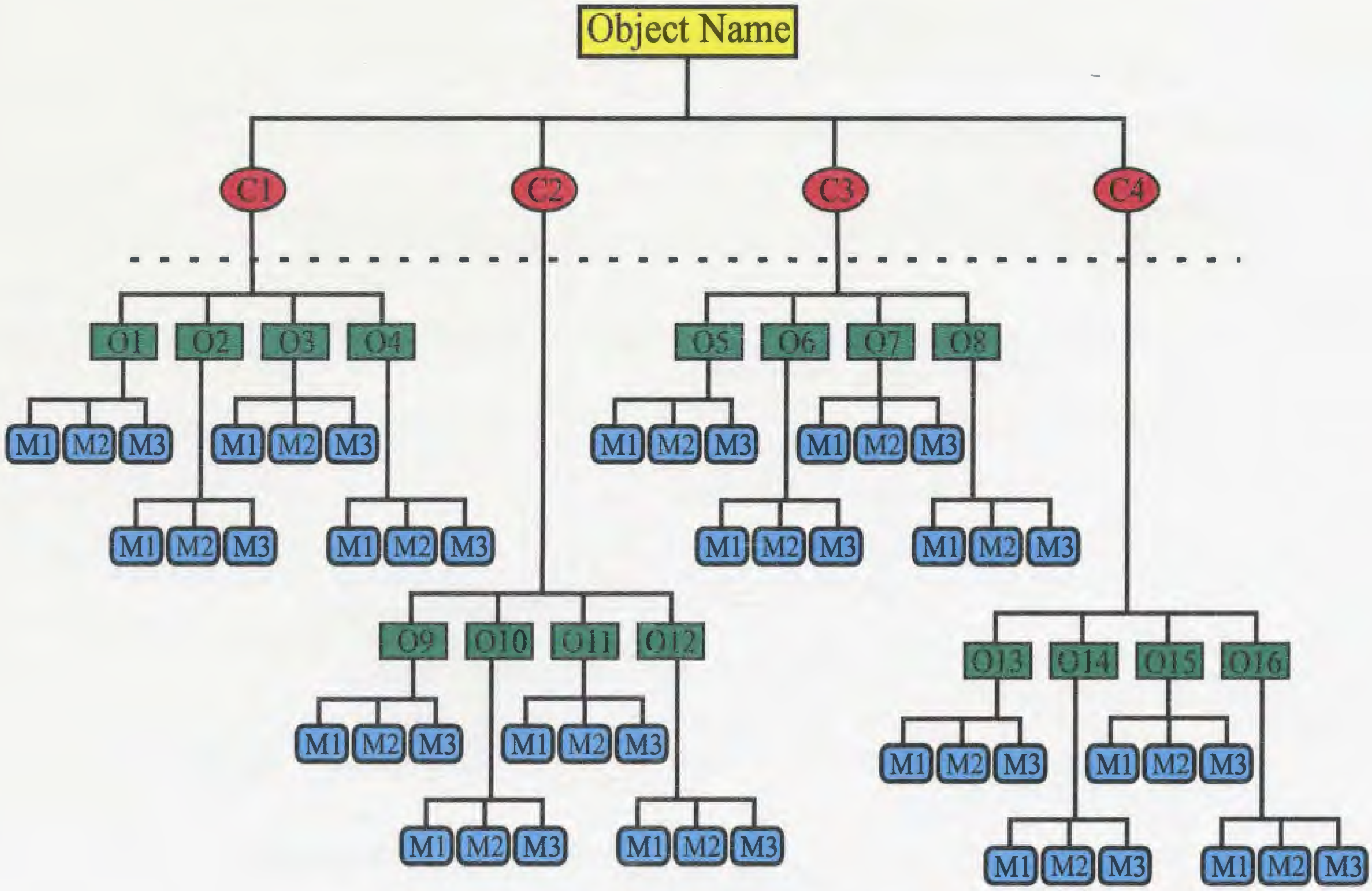


Figure 4.3.6: Organization scheme for *get_object*, implementation #1.

4.3.2 `Get_Object_Number` function, implementation #1

The function which determines the object number is also different in each implementation. In this implementation it simply searches the array `Object_name`, and does a string comparison to match the name passed into *get_object* and that stored. When a match is found the object number is set to the index of the array. This function is shown below, Figure 4.3.7.

Note that when `PRG_manager` is called in this routine the routine name supplied is UTILITY. The reason will be explained in section 4.5 when the *program manager* is discussed in detail.

4.3.3 *Get_object* Routine, Implementation #2

In the second implementation of *get_object* the organization scheme of the code was re-examined. Taking into consideration some of the desired features of the code, it was decided that the series of select cases in *get_object*, which determines which object building routine is invoked, would be changed. Once the *build object list* was changed to use a derived type for the object list, it was realized that class, name, and modality were three independent characteristics, that collectively identify a particular object. However, since they were independent, the order that the select case is executed does not matter. Originally the fact that the object name was stored in the form `class:Object_name%modality` steered the decision to organized the code according to the scheme presented in section 4.3.1. Once this

Figure 4.3.7: Get_object_number function for *get_object* implementation #1

```
      recursive integer function get_object_number (obj_name)
*******************************************************************************
*     Date last modified: April 3, 2002                         Version 2.0  *
*     Author: R.A. Poirier                                                   *
*     Description: Given an object name, determine the object number it      *
*     corresponds to.                                                        *
*******************************************************************************
* Modules:
      USE program_manager
      USE objects

      implicit none
*
* Input scalars:
      character*(*), intent(IN):: obj_name
*
* Local scalars:
      integer Nobj
      logical found
      character(len=132) class
      character(len=132) name
      character(len=132) modality
*
* Begin:
*      call PRG_manager ('enter', 'get_object_number', 'UTILITY')
*
      get_object_number=0
      found=.false.
      Nobj=0
*
* Extract the class/object name and modality:
      class=obj_name(1:index(obj_name,':')-1)
      name=obj_name(index(obj_name,':')+1:len_trim(Obj_name))
      modality=' '
      if(index(obj_name,'%').ne.0)then
        name=obj_name(index(obj_name,':')+1:index(obj_name,'%')-1)
        modality=obj_name((index(obj_name,'%')+1):len_trim(Obj_name))
      end if
*
      do while (Nobj.lt.NObjects.and..not.found)
        Nobj=Nobj+1
        if(Object(Nobj)%class.eq.class)then
          if(Object(Nobj)%name.eq.name) then
            if((Object(Nobj)%modality.eq.modality).or.(modality.eq.' ')) then
              get_object_number=Nobj
              found=.true.
            end if
          end if
        end if
      end do ! while

      if(.not.found)then
        write(uniout,'(6a)')'ERROR> get_object_number: Object,
',class(1:len_trim(class)),':',
     .                  name(1:len_trim(name)), '%',
     .                  modality(1:len_trim(modality)),' not found in list'
        write(uniout,'(a)')'Add the Object or make sure the name is correct'
        stop'ERROR> get_object_number: Object not found in list'
      end if
*
*      call PRG_manager ('exit', 'get_object_number', 'UTILITY')
      return
      end
```

mind set was changed, the chance to meet one of the primary goals of the program was noticed.

A primary goal of the program was that it have to ability to be divided into packages. Looking back at Figure 2.1.2 it can be see that these packages include, *ab initio*, Molecular Mechanics, DFT, etc. It was seen that if the order the select cases were performed was changed, division into packages would be much easier to accomplish.

The new scheme that was implemented first performs the select case on a new characteristic, modality type. The modality type represents the packages listed above. By performing the first select case based on modality type, it becomes easy to cut out that section of the select case, thus removing that package from the code. Every object that is required in one package will be below that "branch" of the select case. Figure 4.3.8 shows the organization scheme for the second implementation of *get_object*. Notice that if the code was broken at the dotted line, then everything below the modality type, will belong to that one package.

Figure 4.3.9 shows the code for the second implementation of *get_object*. The main algorithm is the same as implementation #1. First the object number is determine, by calling get_object_number function. Then the status of the object is determined. Then the select case is executed.

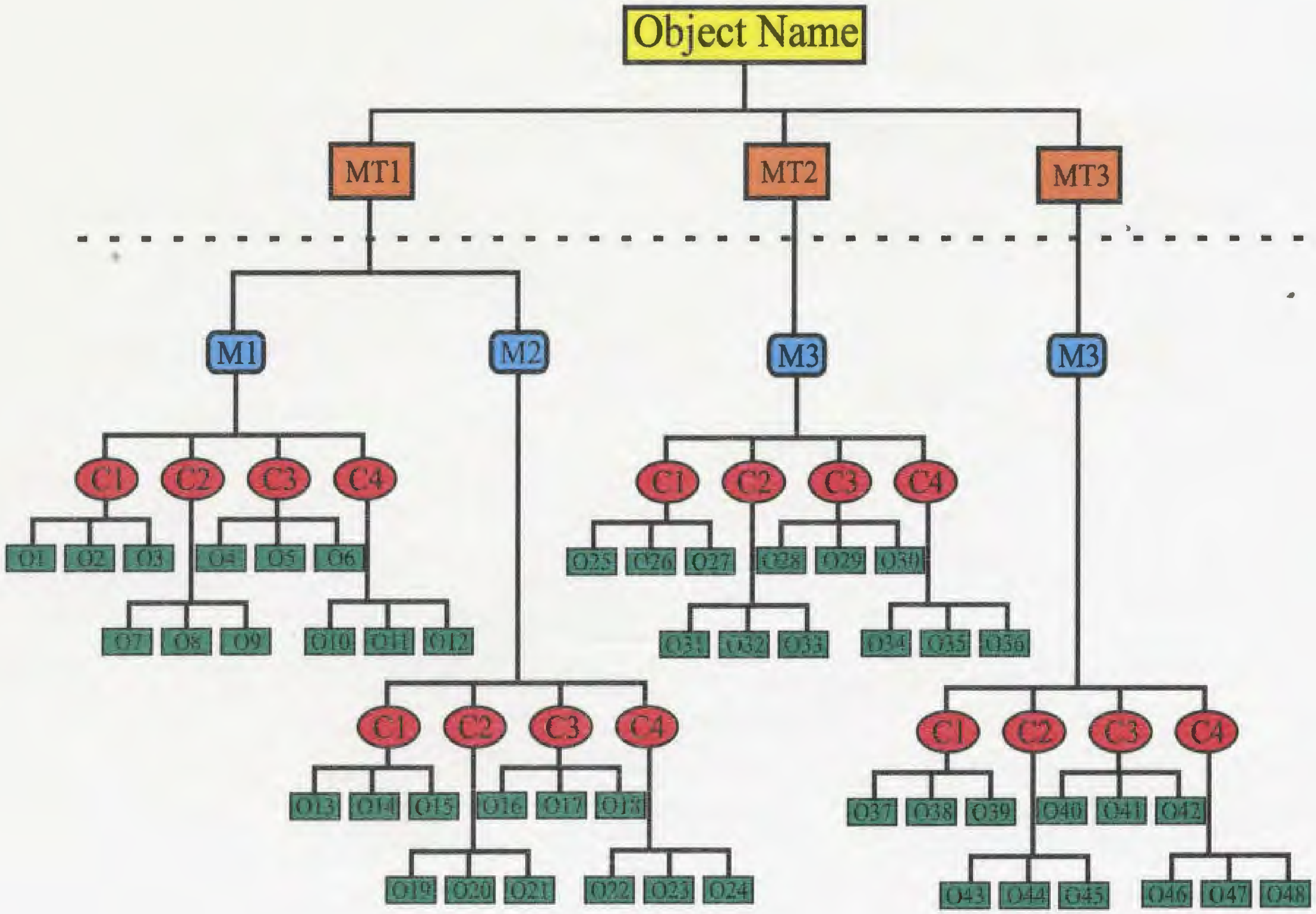


Figure 4.3.8: Organization scheme for *get_object*, implementation #2.

Figure 4.3.9: Function to determine Modality Type. This function will disappear once modality type is added to the object identification.

```
      recursive character (len=132) function modality_type (modality, ObjName)
************************************************************************************
*     Date last modified: June 14, 2001                             Version 2.0  *
*     Author: Darryl Reid                                                        *
*     Description: Given an object Classe, determine the modality type that      *
*     corresponds to.                                                            *
************************************************************************************
* Modules:
      USE program_manager
      USE objects

      implicit none
*
* Input scalars:
      character*(*), intent(IN):: modality
      character*(*), intent(IN):: ObjName
*
* Local scalars:
*
* Begin:
      call PRG_manager ('enter', 'modality_type', 'UTILITY')

      CLASS: select case (modality)

      case ('ZM')
        modality_type = 'COORD'

      case ('PIC')
        modality_type='COORD'

      case ('RIC')
        modality_type='COORD'

      case ('RHF')
        modality_type='WFN'

      . . .

      case ('MM')
        modality_type='FF'

      case ('WAVEFUNCTION')
        modality_type='WFN'

      case ('COORDINATES')
        modality_type='COORD'

      case ('other')
        modality_type='other'

      case default
        modality_type='other'
        write(uniout,'(5a)')'No such modality "',modality(1:len_trim(modality)),
     .                      '" for object "',ObjName(1:len_trim(ObjName)),'"'
        stop'No such modality'
      end select CLASS
*
      call PRG_manager ('exit', 'modality_type', 'UTILITY')
      return
      end
```

One difference is that now the modality type must be determined. The function Modality_Type is invoked to obtain this value. Figure 4.3.9 shows the code for the modality_type function. This function and the call to it is an intermediate step in the code evolution. Eventually this should be removed and the modality type should be added to the object definition, and then the value of Object(Obj_Num)%modality_type will have to be checked to determine the modality type. For now each modality that exists in the code is an option of the select case in the function Modality_type, and when called with a modality value, the function returns the corresponding modality type.

With the value of modality type in hand, *get_object* then performs the select case based on it. The select case then goes through a series of steps, beginning with the modality, then the class, then the object, as can be seen in Figure 4.3.8. Although this does require at least on additional level of routine calls before the *object building routine* is called, it is accepted to enable the feature of separate packages to be implemented.

Figure 4.3.10: *Get_object* routine with select case based on modality type.

```
      recursive subroutine get_object (obj_name)
*************************************************************************
*     Date last modified: April 3, 2000                     Version 2.0  *
*     Author: R.A. Poirier                                               *
*     Desciption: Given an object name call the appropriate routine      *
************************************************************************
* Modules:
      USE program_manager
      USE objects

      implicit none

... Variable declarations go here, remove for space reasons ...

* Begin:
      call PRG_manager ('enter', 'GET_OBJECT', 'UTILITY')
*
      ObjNum=get_object_number(obj_name//' ')
*
* Object will exist and will be current:
      if(Object(ObjNum)%current)then
        call PRG_manager ('exit', 'GET_OBJECT', 'UTILITY')
        RETURN
      end if
      Object(ObjNum)%exist=.true.
      Object(ObjNum)%current=.true.
*
* Obtain the class/object name and modality:
      class=Object(ObjNum)%Class
      Objname=Object(ObjNum)%name
      name=Object(ObjNum)%name
      modality=Object(ObjNum)%modality

      Mod_type = modality_type(modality, name)
      if(index(modality,'other').eq.0)then
      name=Objname(1:len_trim(Objname))//
     .       '%'//modality(1:len_trim(modality))
      end if
*
* Check for class
      ModalityType: select case (Mod_type)

      case ('other')
        call Other_objects (modality, class, name)

      case ('COORD')
        call COORD_objects (modality, class, Objname)

      case ('FF')
        call FF_objects (modality, class, name)

      case ('WFN')
        call WFN_objects (modality, class, Objname)

      case default
        write(uniout,'(5a)')'No such Modality Type "',Mod_type(1:len_trim
     .  (Mod_type)),'" for object "',obj_name(1:len_trim(obj_name)),'"'
        stop'No such Modality Type'
      end select ModalityType
*
      call PRG_manager ('exit', 'GET_OBJECT', 'UTILITY')
      return
      end
```

Figure 4.3.11: Get object select case based on Modality. This is the second level of select case hierarchy.

```
      subroutine WFN_objects (modality, class, Object)
**************************************************************************
*     Date last modified: June 21, 2001                       Version 2.0 *
*     Author: Darryl Reid                                                 *
*     Description: Wavefunction (WFN) Modality Type objects.              *
**************************************************************************
* Modules:
      USE program_manager

      implicit none
*
* Input scalar:
      character*(*) modality, class,Object
*
* Begin:
      call PRG_manager ('enter', 'WFN_objects', 'UTILITY')
*
        select case (modality)
        case ('RHF')
        include 'case_RHF_objects'

        case ('UHF')
        include 'case_UHF_objects'

        case ('GVB')
        include 'case_GVB_objects'

        case ('WAVEFUNCTION')
        include 'case_WFN_objects'

        case ('ROHF')
        write(uniout,*)'case modality ',modality(1:len_trim(modality))
        include 'case_ROHF_objects'

        case default
          write(uniout,*)'No such object "',modality(1:len_trim(modality)),
     .                   '" for class "',class(1:len_trim(class)),'"'
          stop'No such modality'
        end select
*
      call PRG_manager ('exit', 'WFN_objects', 'UTILITY')
      RETURN
      END subroutine WFN_objects
```

To code this implementation, three levels of the select case need to be created. Putting the entire select case in one subroutine and file would have created a very large file that would be very difficult to maintain. A file, main_modality.f, was created that holds the required routines of select cases. As can be seen in Figure 4.3.10 the modality type, 'WFN' or wave function, calls a routine WFN_objects. This routine, found in main_modality.f, was then invoked and the modality is used to select the next choice. Figure 4.3.11 shows a portion of the routine WFN_objects, other modality routines are similar.

Here, an acceptable use of include files can be seen. As mentioned in chapter three, include files should be avoided. They can be over used and can make code hard to read, and cause

Figure 4.3.12: Get object select case step three, based on class. This file will be an 'include' file.

```
          select case (Class)
           case ('DENSITY')

           select case (Object)
           case ('1MATRIX')
             call DENSITY_1MATRIX_RHF

           case ('ENERGY_WEIGHTED')
             call DENSITY_Eweighted_RHF

           case default
             write(uniout,*)'No such object "',Object(1:len_trim(Object)),
     .                      '" for class "',class(1:len_trim(class)),'"'
             stop'No such object'
           end select

         case ('GUESS')
           select case (Object)
           case ('DENSITY')
             call DENSITY_guess_RHF

           case ('MO')
             call BLD_GUESS_MO

           case default
             write(uniout,*)'No such object "',Object(1:len_trim(Object)),
     .                      '" for class "',class(1:len_trim(class)),'"'
             stop'No such object'
           end select

         case ('MO')
           select case (Object)
           case ('COEFFICIENTS')
             call RHCCLC

           case default
             write(uniout,*)'No such object "',Object(1:len_trim(Object)),
     .                      '" for class "',class(1:len_trim(class)),'"'
             stop'No such object'
           end select
        case default
           write(uniout,*)'No such object "',Object(1:len_trim(Object)),
     .                    '" for class "',class(1:len_trim(class)),'"'
           stop'No such object'
         end select
```

trouble with dependencies in Makefiles. However in this case, include files are used very carefully to improve the readability of the code. By having the include files in the select case it can be seen exactly which objects are being included. The naming is very important here. The names used mean something to the reader of the code. For example, it is clear that the file 'case_RHF_objects' will have all the RHF objects in it and will be a select case. Figure 4.3.12 shows the 'case_RHF_objects' include file. Here it can be seen that both the class select case and the object select case are in the same file, this helps for maintaining the code. When adding a new object, the programmer would just have to find the appropriate include file, and place the call to the building routine in the correct spot in the select case (based on the class and object name, alphabetical order).

The next two sections described, *Build object routine*, and *program_manager* will not be presented as two separate implementations, since the implementations did not change that much. Some instances of each implementation will be mentioned, but not to a large degree.

4.4 Routine to Build / Create Objects

Another major piece of the infrastructure are the routines that build the objects. As discussed in Section 4.3, *get_object* is responsible for calling the *object building routine* for the requested object. It is the *object building routine* that the scientific programmer will create. The *object building routine* is the code implementation of the scientific theories. The algorithm developed from scientific theory is implemented in the *object building routine.* Once created, the other three main parts of the infrastructure, the *build object list, get_object,* and *program manager,* require very little input or maintenance, it is the *object building routine* that the scientific programmer will spend most of their time on.

In accordance with the design protocols presented in Chapter 3, the *object building routine* must follow a very strict template. Figure 4.4.1 shows the template of a routine that builds an object. As with all routines in the program, the *object building routine* starts with a meaningful routine name and a routine header, encased in a square of arteries, which contains all the information discussed in section 3.2.1. The required modules are then listed after a comment line which labels them as modules (`* Modules:`). Then the most important line in any routine is included, "`implicit none`". As stated, every routine must have implicit none in it. See section 3.3.1 for a more detailed discussion of implicit none. Following implicit none, all required variables are declared.

Once variables are declared, the routine must call the *program manager*, PRG_manager. The program manager oversees the operations of the program and is very important to the development and maintenance of large scientific codes, section 4.5 will discuss the *program manager* in detail.

The building routine then builds the object. The *object building routine* must, first, "get" all the objects it requires to build the object, that is all the objects it depends on. This is done by a series of calls to *get_object* with the appropriate object names. Once these calls are complete, the *object building routine* has all the information it needs to start building the object. Before the code to build the object is executed, the *object building routine* checks if debugging has been requested, and acts accordingly.

The comment line "`*            Code to build object`" in the template looks almost insignificant. However this is the line that is replaced to implement the algorithms which the scientific programmer develops from scientific theory. Often this section is hundreds to thousands of lines of code, and can take a programmer months of work to write. However once the general template has been created the scientific programmer is able to concentrate their efforts on this most important section of the code. Incorporating this into the overall program then becomes a trivial task. Chapter 5 will present the simple steps a programmer must follow to add new objects to the program.

Figure 4.4.1: Object building routine template

```
Sample subroutine:
      MODULE class_name
**************************************************************************************
*     Date last modified: February 16, 2000                             Version 2.0  *
*     Author: R.A. Poirier                                                           *
*     Description: Contains this and that                                            *
**************************************************************************************
      implicit none
*
* Scalars
      integer scalar1
*
* Arrays
      double precision, dimension(:), allocatable :: array1
      double precision, dimension(:), allocatable :: array2
      double precision, dimension(:), allocatable :: array2
*
      CONTAINS
      SUBROUTINE Routine_NAME
**************************************************************************************
*     Date last modified: September 28, 2000                            Version 2.0  *
*     Author: Darryl Reid                                                            *
*     Description: Computes OBJECT_NAME for the following MODALITY.                  *
**************************************************************************************
* Modules:
      USE object_based_main
      USE class_name1
      USE class_name2

      implicit none
*
* Begin:
      call PRG_manager ('enter', 'Routine_NAME', 'CLASS:OBJECT_NAME%MODALITY')
*
* Get all required objects:
      call get_object ('CLASS:DEFAULTS')
      call get_object ('CLASS1:OBJECT_NAME1%MODALITY')
      call get_object ('CLASS2:OBJECT_NAME2%MODALITY')
*
* Build the object:
      if(local_debug)then
*       print required objects
      end if
*
*     Code to build object
*
      end if ! if(.not.Object_current(ObjNum))
*
* Check if print of object is requested (always print for a debug):
      if(Object_print(ObjNum).or.local_debug)then
*       code to print object
      end if
*
* Check if saving of object is requested:
      if(Object_save(ObjNum))then
*       code to save object
      end if
*
* End of routine Routine_NAME
      call msg_print ('exit', 'Routine_NAME', 'CLASS:OBJECT_NAME%MODALITY')
      RETURN
      END subroutine Routine_NAME
      end MODULE class_name
```

4.5 Program Management Tools

The next major element of the backbone or infrastructure of the code is the program management tools. Program management tools are a collection of tools designed to help the programmers write the code. Their purpose is to help in the development of the code. The program management tools consist of error checking tools, as well as debugging tools, both useful to the programmer during the development stages. In addition the tools have a timing function built in which can be used for code optimization and can build a dependancy matrix which could have important applications for future code developments.

The *program manager* is a subroutine that is central in the overall program scheme. In general, every routine calls the *program manager* when it first enters and just before it exits. There are some exceptions to this, for example routines that could be called millions of times during the execution of the code will leave out the call to the *program manager*, since the overhead would be to much. The main purpose of the *program manager* is to keep track of information about the code's execution which are deemed useful to the programmer. As has been noted, every object in the code has a unique object number. The *program manager* uses this object number in many of its features. For instance, when a trace of the codes execution is required the program manager uses the object number to print the desired information.

The program manager is not essential to the execution of the code. It is for developmental purposes only and can be removed in a finished "number crunching" version of the code. This removal would involve some global substitutions since the *program manager* is used in every routine in the code. This removal should increase the efficiency of the code.

The *program manager* is intimately connected with the rest of the code infrastructure, especially the *object_list*. The *program manager* uses the information stored the *object_list* for most of its functionality.

The program MUNgauss has two main types of sub-program units, utilities, and object building routines. Object building routines were described in section 4.3, they perform the required computations to build an object. Utilities are a portion of the code utilized by the program to perform some task, they do not build an object. These routines are essential to the operation of the program however since they are not objects they do not have an object number associated with them. With this in mind the *program manager* has to be able to deal with both kinds of sub-program units.

To deal with utilities and objects, the *program manager* subroutine is divided into two main parts, one for each sub-program unit. Figure 4.5.1 shows a simplified flow chart of the *program manager* subroutine. It can been seen that the first step is to decide whether it is dealing with an object or a utility. When working with a utility the *program manager* is only concerned with performing a trace. The main part of the *program manager* is the object half of the subroutine. There are five main parts of the *program manager* each of these are shown in Figure 4.5.1. There is error checking, debug, timing, dependency and trace parts of the routine. A description of the implementation and function of each of these parts follows.

4.5.1 Error checking

The first feature in the program manager is the error checking portion. This checks to ensure that the *program manager* is working with the correct information. The check is between the object passed into the *program manager* from the building routine (which calls the program manager) and the object that is stored in the *object_list* that corresponds to the current *ObjNum* (object number). This check is simply implemented using a string comparison of each object name. Figure 4.5.2 shows the error check in *program manager*.

Figure 4.5.2: Sample error check.

```
* Error Checking to ensure the correct ObjNum is being used
      class=Local_Object_Name(1:index(Local_Object_Name,':')-1)
      name=Local_Object_Name(index(Local_Object_Name,':')+1:len_trim(Local_Object_Name))
      modality=' '
      if(index(Local_Object_Name,'%').ne.0)then
       name=Local_Object_Name(index(Local_Object_Name,':')+1:
     .                        index(Local_Object_Name,'%')-1)
       modality=Local_Object_Name((index(Local_Object_Name,'%')+1):
     .                            len_trim(Local_Object_Name))
      end if
*
      if (Routine_Name .NE. Object(ObjNum)%routine(1:len_trim(Object(ObjNum)%routine)))
     .  then
        write(uniout, *) 'Routine Names do not match ', Routine_Name, ' and ',
     .                  Object(ObjNum)%routine(1:len_trim(Object(ObjNum)%routine)),
     .                  ' upon ', EnterOrExit, ' Object Number ', ObjNum, ' Index ',
     .                  ObjectArrayIndex
        stop
```

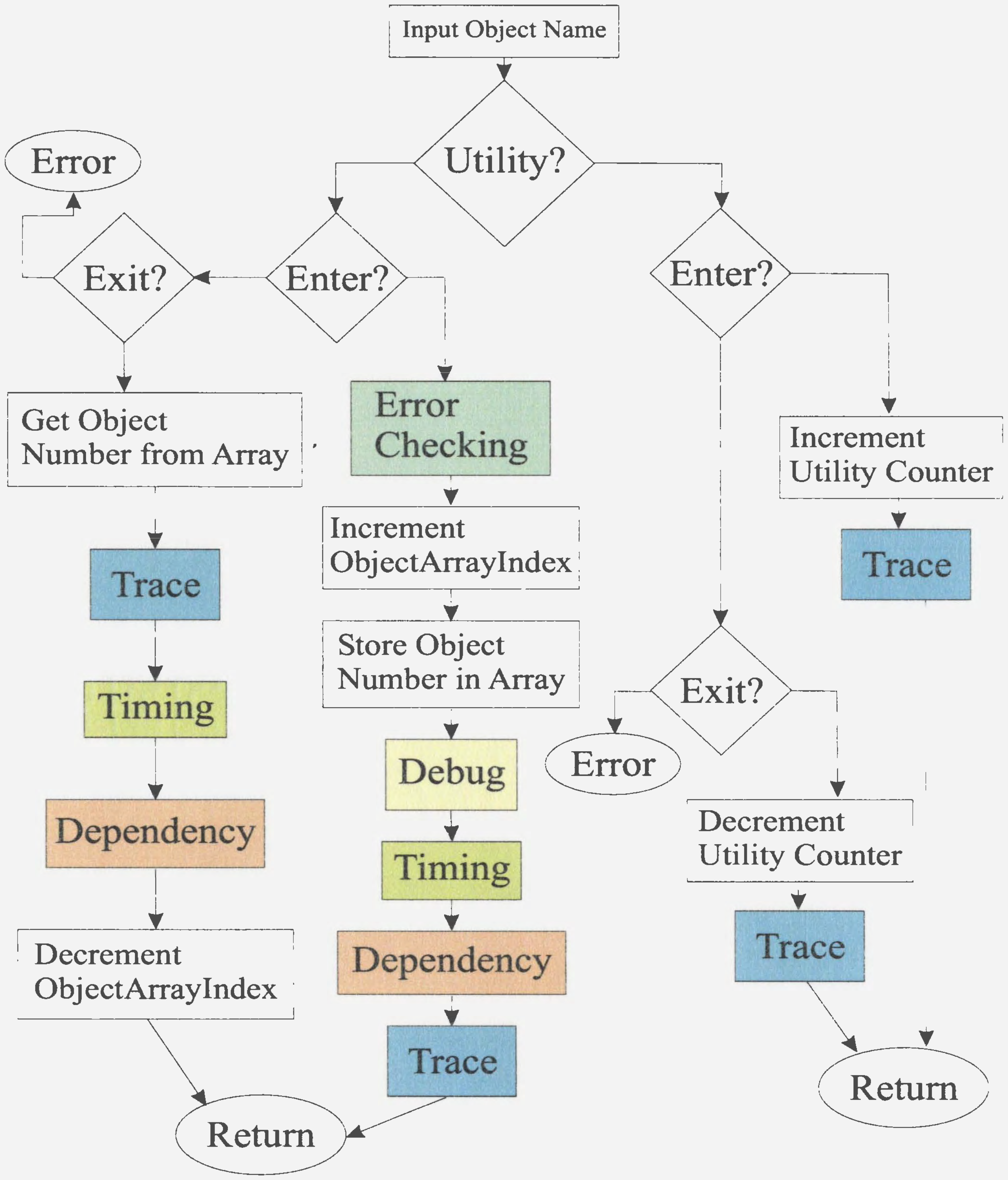

Input Object Name
Utility?
Error
Exit?
Enter?
Enter?
Get Object Number from Array
Error Checking
Increment Utility Counter
Trace
Increment ObjectArrayIndex
Trace
Store Object Number in Array
Timing
Exit?
Error
Debug
Dependency
Decrement Utility Counter
Timing
Decrement ObjectArrayIndex
Dependency
Trace
Trace
Return
Return

Figure 4.5.1: Simplified flow chart for *program manager*. Each of the five main functions are shown in different colors, error checking, debugging, timing, dependency, and tracing.

4.5.2 Local Debug

Debugging is one of the most difficult and most important part of code development. There are many tools available to help debug programs. However, they are often difficult to learn and use, give ambiguous information regarding the error, and are often not suited to very large codes. For the development of MUNgauss, a set of debugging tools were built into the code. Each routine has (or at least can have) sections that contain useful debugging information to help the programmer locate errors. These sections are contained within an "if" statement that is controlled by a local debug variable. It is the responsibility of the *program manager* to determine if local debugging was requested by the programmer and to set the value of the local debug variable (`Local_Debug`) to true or false as required.

As stated in Section 4.2, the *object_list* includes an array (in implementation #1) or a variable in the derived type (implementation #2) that stores if the local debugging of an object is requested. The request is an input from the programmer, via the menu of the program. When the *program manager* is called it checks if debugging was requested by simply using the *ObjNum* to check the appropriate value in the *object_list*. It then sets the value of `Local_Debug` to the correct value (true or false). This value is then passed on to the routine (through the module) and the debugging portion of the program is executed.

4.5.3 CPU_timing

Knowing the time it takes a subroutine to execute can be very useful for a programmer during code development. It can help pinpoint areas of the code which take the most time and allow the programmer to work on those areas to help the overall performance of the program. The *program manager* contains tools to gather the execution time of subroutines in MUNgauss. The manner in which *program manager* does this is by using a call to a function CPU_TIME. Unfortunately this function is not standard Fortran 90, however it is part of the Fortran 95 standard[7]. The basic procedure of collection timing information should not change, the only part that may change is the call to the function and possibly the manipulation of the results for printing.

Figure 4.5.3: Timing within the program manager

Entering portion of Program Manager:

```
* CPU_Time Computation
        if (Object_cputiming(ObjNum)) then
          call CPU_TIME(Begin_time)
          BeginTimeArray(ObjectArrayIndex) = Begin_time
        end if !LCPU_Time
```

Exiting portion of Program Manger:

```
* Computation time printing
        if (Object_cputiming(Object_Number_list(ObjectArrayIndex))) then
          call CPU_TIME(end_time)
          EndTimeArray(ObjectArrayIndex) = end_time

          write(uniout, *)  Object_routine(ObjNum)(1:len_trim(Object_routine(ObjNum))),
     .         ' took ',EndTimeArray(ObjectArrayIndex) -
     .         BeginTimeArray(ObjectArrayIndex), ' seconds',
     .         ' Object Number ' , Object_Number_list(ObjectArrayIndex),
     .         BeginTimeArray(ObjectArrayIndex), EndTimeArray(ObjectArrayIndex)
        end if !LCPU_Time
```

When the *program manager* is called, it checks to see if cpu time was requested for that object, by checking the corresponding element in the `Object_cputiming` array. If timing is required, the *program manager* places a call to `CPU_TIME` function and stores the result in a `BeginTimeArray` at the next element in that array. When the calling routine calls the *program manager* upon exiting, the *program manager* again calls `CPU_TIME` and obtains a value for the end time of the subroutine. It is then just a matter of subtracting the beginning time from the ending time to obtain the elapsed execution time for the subroutine. Figure 4.5.3 shows the two pieces of timing code.

4.5.4 Dependency

An interesting addition to the *program manager* is the creation of a dependency matrix. A dependancy matrix is a matrix that contains information on which objects depend on other objects. In the *program manager* functionality has been created for the program to construct the first order dependancy matrix for the program MUNgauss on the fly, or as the code executes. This information can be printed at the end of the execution if requested by the programmer/user.

Why is the dependency matrix of interest? Firstly if can give the programmer an idea of how the code is being executed and how objects relate to one another. This kind of information can help the programmer gain a higher level of understanding of how the program operates. This can be very useful in large scientific codes, as they can be quite complex, any assistance in understanding the codes operations is helpful.

Figure 4.5.4: Dependency matrix portion of program manager.

```
Entering portion of Program Manger:

* Build the First Order Dependency Matrix.
        if (Ldependency) then
          if (.not.allocated(First_Order_Depend)) then
            allocate (First_Order_Depend(Nobjects,Nobjects))
          end if
          DependCounter = DependCounter + 1
          if (DependCounter .gt. 1) then
            First_Order_Depend((Object_Number_list(ObjectArrayIndex-1)), ObjNum)=.true.
            First_Order_Depend(ObjNum, ObjNum) = .true.
          else if (DependCounter .eq. 1) then
            First_Order_Depend(1:NObjects,1:Nobjects) = .false.
            First_Order_Depend(ObjNum, ObjNum) = .true.
          end if !DependCounter
        end if !Ldependency

Exiting portion of Program Manager

        if (Ldependency) then
          DependCounter = DependCounter - 1
        end if !Ldependency

        ObjectArrayIndex = ObjectArrayIndex - 1
```

The second use of the dependency matrix is work for future code development. Parallelization of programs is becoming a very widely used technique to improve a code's performance. Knowledge of how objects depend on each other is a good first step in designing a parallel algorithm of a program. The goal is to design the program in such a way that it will first build the dependency matrix, on the fly, and then use that matrix to determine a level of parallelization. This kind of "automatic" parallelization will be difficult to obtain, however the dependency matrix could be an important start.

4.6 How it all works together

The four main portions of the programs infrastructure do not work independently. They interact in a very special way that reaches many of the goals of well-written codes discussed in Chapter 2. The overall picture of how each of the pieces of infrastructure interact to produce a program can be seen in Figure 4.6.1. Here it can been seen that the user asks for a particular object (or piece of information, such as the energy of a system at a particular level of theory). The code then calls *get_object* to get that object, *get_object* will then select the correct *building routine* which will proceed to build the desired object. The route independence discussed in section 1.2 is illustrated here, it can be seen that the building routine will "get" an object that the desired object depends on. By calling *get_object* the code determines its own path of execution. The *build_object_list* and the *program manager* are not included in Figure 4.6.1 since most of there work is done "behind the scenes". The object list created by *build_object_list* is stored in memory and an available repository of information about each object in the program. It is key to the functioning of the program, but is only executed once at the start of the program, then just sits there providing information to the rest of the program.

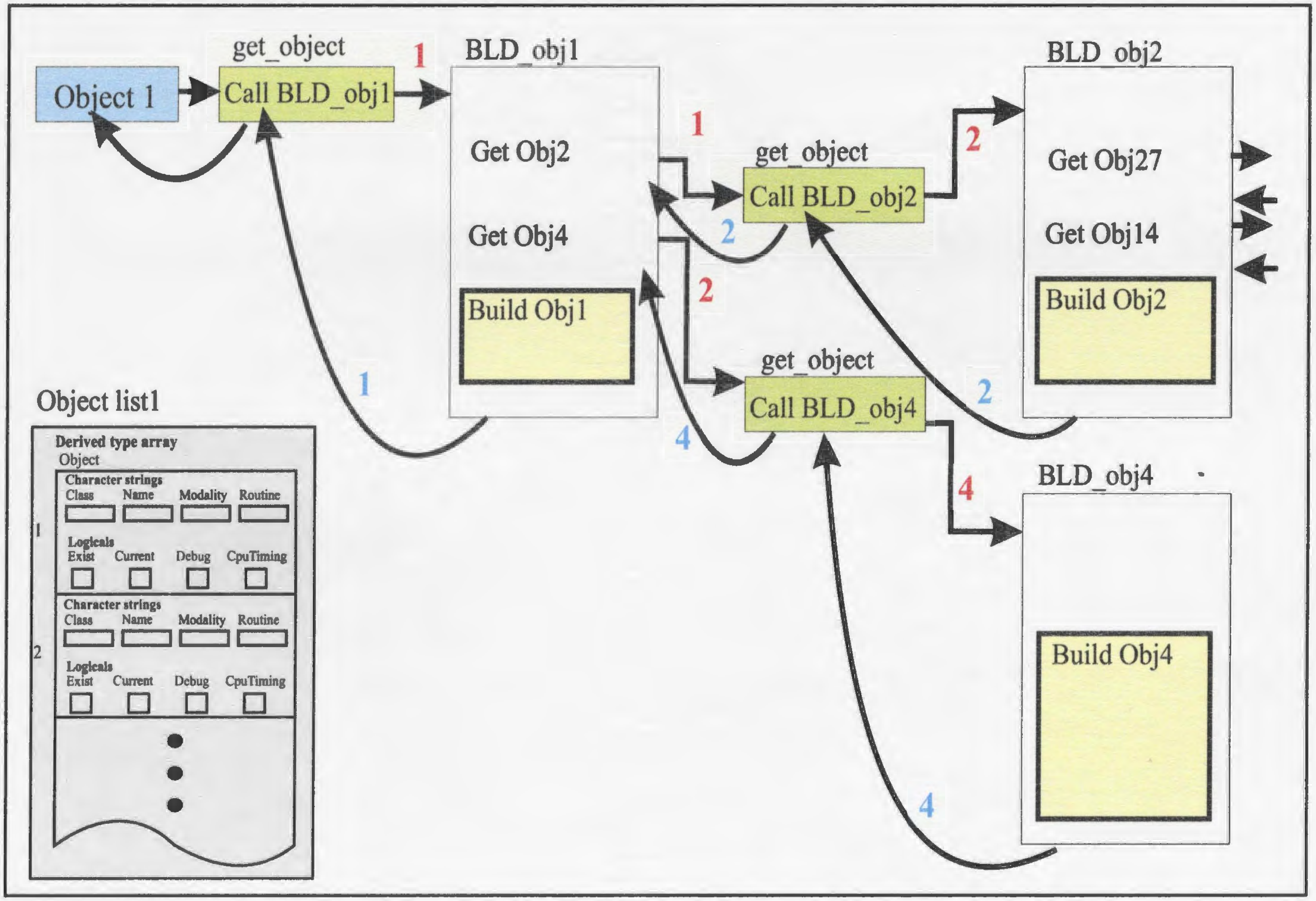
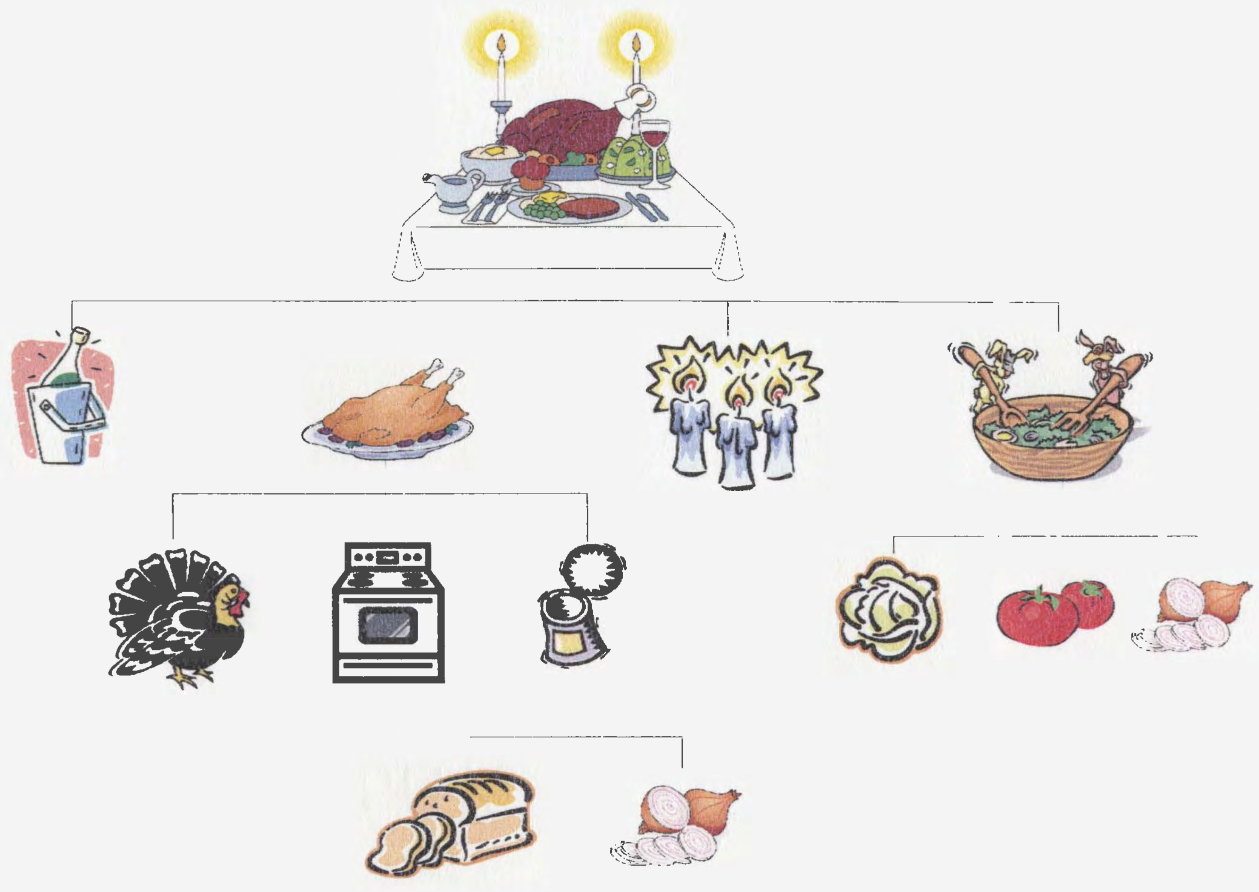


Figure 4.6.1: Sample code execution. Note the route independence, route is determined dynamically. The numbers indicate the value of the Object Number at that point in the execution. Red values are set by *get_object* going down the tree of execution (right) while the blue numbers are set by the *program manager* coming back up the tree (left).

The *program manager*, as previously noted, is called at the beginning and end of each and every routine or function in the program. Every routine shown in Figure 4.6.1 does call program manager, however it is not included for clarity. The program manager acts mainly as a record keeper of the program. The *program manager,* keeps track of which routine called which routine, and when. It can track the time the program takes in each routine. It also acts as a prompter to cause the routines to perform debugging features requested by the user. The *program manager* also keeps track of the object number and makes sure that it is always correct when used by any other routine. As the code goes further down the tree of execution (further to the right in Figure 4.6.1) the object number changes whenever *get_object* is called. However when the code starts to return up the levels, the object number would be that of the last object which is built. The *program manager* resets the object number to the correct value when it is called at the end of a building routine. This allows other routines access to the correct object, via the object number.

Following the logic of this kind of programming approach may be a little confusing. An analogy may help. Lets say you want to make (build) a christmas dinner. Well if this programming approach was used you would simply ask for the object "Christmas_dinner". *Get_object* would call the building routine "BLD_Xmas_dinner". This routine would "know" that object "Christmas_dinner" needed objects "Bottle_wine", "Cooked_Turkey", "candles" and "Salad" to be built. Once "BLD_Xmas_dinner" has each of these objects it could build a Christmas dinner. So it calls *get_object* for "Bottle_wine", *get_object* calls the

Figure 4.6.2: Illustration of "BLD_Xmas_dinner" routine.



BLD_Bottle_wine and returns the object "Bottle_wine". "BLD_Xmas_dinner" would then ask *get_object* for the object "Cooked_Turkey". *Get_object* would then call "BLD_Cooked_Turkey".

Now BLD_Cooked_Turkey needs "Turkey", "Oven" and "Stuffing" to build the object "Cooked_Turkey". It then goes through the same process of calling *get_object* to obtain all the objects in needs to build "Cooked_Turkey". Once BLD_Cooked_Turkey finishes it returns the object "Cooked_Turkey" to BLD_Xmas_dinner. BLD_Xmas_dinner continues this process to obtain "Candles" and "Salad" objects. This example may seem a little silly but is does illustrate how the program uses objects and that each *build object routine* just needs to "ask" the program for the objects it requires to build the desired object. BLD_Xmas_dinner does not care how the object "Salad" was built, it just needs the object "Salad" to be created and be current. The programs infrastructure ensures that when a building routine asks for an object it is given a current object.

Chapter 5

Implementation

5.1 Introduction

The preceding four chapters described an approach to writing a large scientific program. From beginning concepts of a "purpose statement of the code" through a process of identifying objects, classes, etc. and on to building an infrastructure upon which the code can be placed, the approach has been presented. However up to this point, no actual scientific code has been written. The bulk of this work was to create the infrastructure, clearly define the protocols, and help organize things in such a manner as they can be easily maintained, but at the end of the day a piece of scientific code is needed. This chapter aims to describe the steps involved in adding a piece of scientific code to the program created. As mentioned, MUNgauss is an evolving program, it has been in development for 20+ years and continues to change. The process described below has been done (in one way or another) to every part of MUNgauss. Many portions of the program are somewhere in the middle of the conversion process, however it is important to point out that the code is still functional. The process described in Section 5.2 allows for incremental change. Once the infrastructure has been created, the old code need only be modified very little to get it to "run", however some fine tuning would have to be done to get it to conform to the design protocol described in Chapter 3. That part of the conversion is the most time consuming part.

5.2 Adding new objects to the program

The procedure for adding new functionality to the program is a straight forward one. There are a few points where some thought will be required and some juggling may occur, but for the most part adding new features is a simple manner of following the steps described below.

1. Decide what functionality to add, and what objects will be added.

Often the programmer will end up only adding one object to get the desired result, however sometimes multiple objects will have to be combine to achieve the desired functionality. Give the object(s) a name and determine the class in which it belongs. Looking back at the table of classes in Chapter 2 can help make this decision. This initial placement of the object is sometimes very easy, as it will obviously fit into a class, but sometimes placement can be tricky and adding a new class may have to be considered. Here are some things to keep in mind:

1.1 Keep things that have something in common together (i.e. comes from the same theory, calculates things similarly)

1.2 The idea of creating separate "packages" was a major goal of the project, thus make sure the separation is maintained between packages.

1.3 Object names have to be unique, and meaningful.

1.4 Does a modality apply to this object, if so this plays a key role in determining how it is integrated into the code.

Once these decisions have been made all the characteristics of the object should be able to be defined, as shown in Figure 5.2.1. Figure 5.2.1 shows the "Dummy Class" of the BLD_object_list routine discussed in Chapter 4. These characteristics identify the object.

Figure 5.2.1: Template of object characteristics to be placed in the BLD_object_list routine.

```
* Dummy Class
      Nobjects = Nobjects + 1
      Object(Nobjects)%class = '?'
      Object(Nobjects)%name = '?'
      Object(Nobjects)%modality = '?'
      Object(Nobjects)%routine = '?'
```

2. Placing the object into the infrastructure.

With the object identified it must now be inserted into the infrastructure of the program

2.1 First, add the object to the *object list*. As identified in Chapter 4, the object list is a master list of all the objects available in the code. Within the routine "BLD_object_list", find the class that the object belongs and insert the new object, following the template seen in Figure 5.2.1. Doing so will give the new object a unique object number and allow the other features of the infrastructure to work with it.

2.2 The next step is to place the call to the object building routine within the "*get_object*" routine sub-structure. Recall, the *get_object* routine is a group of select cases based first on modality, then class, then object name. In MUNgauss, to help organize this hierarchy of select cases, the implementation makes use of "include" files. Therefore to add the call, to the building routine of the new object, the appropriate include file (called case_MOD_objects, where 'MOD' is the modality of the new object) must be edited. In that file, find the class of the object (or add a new class), and insert the `case` and `call` to the building routine, remembering to stay in alphabetical order.

At this point the building routine and thus the object, is part of the infrastructure of the program and can be called by any routine in the program, or can be requested by the user via the input menu. The only problem is that the building routine has not been created.

3. Write the object building routine.

Generally one is adding either a new, never before created object or adding a piece of older code that now needs to be integrated with the new program. If creating a new object, then simply follow the protocols described throughout this thesis, making sure to follow the template given in Chapter 4 (Figure 4.4.1). Make sure to 'use' the appropriate modules and to include the *program manager* features. If other objects are needed simply call *get_object* and 'use' module for that object and the information will be available. Ensuring the proper practices of variable naming, documentation, error messages, etc. and the object can be created in no time (well the actual algorithm to build the object must be written).

Converting an older piece of code so it can be used in the newly designed program is a step-wise process. Starting by adding a few of the new features, such as the *program manager*, the routine can be made to work with very little effort. The major challenge comes from truly converting old F77 code to F90. As with any integration of old code, when adding old code to the program, the programmer must ensure it has the required information available to it, to build the object. This means a call *to get_object* may be necessary to make any required information available. Often variable names have changed, and this merging could become time consuming, however once complete the code will run without major changes.

Once the code is working, the program can call for that object from anywhere in the code. However, likely the old code will not be well optimized or well documented or follow many of the design protocols of this thesis. Therefore time should be invested in rewriting the routine to reflect the new protocols. This is another often time consuming part, but it will pay off in a few months when a programmer has to go back and edit that piece of code and they have no idea how it works, or what it does.

5.3 Implementing a BSSE Code

As an example of a new object that could be added to MUNgauss, it was decided that the Boys-Bernardi counterpoise correction for Basis Set Superposition Error (BSSE) functionality would be added.

In theory the binding energy, ΔE_{INT}, due to the interaction of species A and B will be given by

$$\Delta E_{INT} = E^{*}_{AB} - (E_A + E_B)$$

where E^{*}_{AB} is the energy of the complex between A and B, and E_A, E_B is the energy of the A and B respectively, in their relaxed geometry and its own basis functions. This is true for an infinite basis set, however with finite basis sets, the functions on A will improve those on B and vise versa, during the optimization for the complex. Therefore the energy, ΔE, will be incorrect by a factor of δ_{BSSE}, known as the basis set superposition error[5,21].

The Boy-Bernardi correction for this error is estimates by

$$\delta_{BSSE} = E(A—*) + E(*—B) - (E_A^{\dagger} + E_B^{\dagger})$$

Where E(A—*) and E(*—B) are the energy of the monomers A and B with the basis set of AB and in the geometry of the complex. $E_A^{\dagger}$ and $E_B^{\dagger}$ are the energy of monomers A and B in the geometry of the complex but using its own basis set.

This correction is then subtracted from the ΔE_{INT} to get the total, corrected ΔE_{INT}(no BSSE).

$$\Delta E_{INT}(\text{no BSSE}) = \Delta E_{INT} - \delta_{BSSE}$$

$$= E^*_{AB} - [E(A—*) + E(*—B)] - (E_A - E_A^{\dagger}) - (E_B - E_B^{\dagger})$$

In practice $E_A \approx E_A^{\dagger}$ and $E_B \approx E_B^{\dagger}$ therefore

$$\Delta E_{INT}(\text{no BSSE}) = E^*_{AB} - [E(A—*) + E(*—B)]$$

So to go through the above procedure, first a new object had to be identified. That is to say the characteristics of the object (class, name, modality and routine) had to be defined.

```
Object(Nobjects)%class = 'ENERGY'

Object(Nobjects)%name = 'BSSE'

Object(Nobjects)%modality = 'RHF'

Object(Nobjects)%routine = 'BLD_ENERGY_BSSE'
```

Next the call to the routine 'BLD_ENERGY_BSSE' must be added to the *get_object* substructure. Therefore the include file 'case_RHF_objects' was edited to contain the class 'ENERGY' and the call to the building routine. Figure 5.3.1 shows this addition.

Figure 5.3.1: Addition to *get_object* substructure to add call to BSSE object building routine.

```
...
      case ('ENERGY')

         select case (Object)
         case ('BSSE')
           call BLD_ENERGY_BSSE
         case ('COMPONENTS')
           call E_JandK
         case default
           write(uniout,*)'No such object "',Object(1:len_trim(Object)),
     .                    '" for class "',class(1:len_trim(class)),'"'
           stop'No such object'
         end select
...
```

Then the template (Figure 4.4.1) was edited to reflect the BSSE object. The actual building routine does not build the BSSE object since time did not permit it. However, the purpose of this exercise was to show the ease with which an object could be added to MUNgauss. In its current state the routine simply returns the negative of the RHF energy. With some algorithmic changes this would produce the correct value.

With the building routine in place, the Makefile had to be edited to ensure the new functionality was compiled. Then the object could be called like any other object in MUNgauss, from any routine or from the menu with the command,

OUTPUT object = ENERGY:BSSE%RHF

When executed with this command the output is

```
E_interaction BSSE-free = 197.154475
BLD_ENERGY_BSSE: NOT CORRECT ENERGY!!
BSSE not available at this time
```

for a sample molecular complex of HF with HF.

Figure 5.3.1: BSSE object building routine.

```
      SUBROUTINE BLD_ENERGY_BSSE
*************************************************************************
*     Date last modified: December 12, 2002               Version 1.2   *
*     Author: Darryl Reid and R. A. Poirier                             *
*     Description: Compute the BSSE free energy                         *
*************************************************************************
* Modules:
      USE program_manager
      USE program_defaults
      USE global_scalars
      USE constants
      USE type_energies

      implicit none
*
* Work arrays:
*
* Local scalars:
      double precision :: E_BSSE_int,E_BSSE_AB,E_BSSE_Astar,E_BSSE_starB
      logical Ldebug
*
* Begin:
      call PRG_manager ('enter', 'BLD_ENERGY_BSSE', 'ENERGY:BSSE%RHF')
      Ldebug=Local_Debug
*
* Get Energy for A---B complex
      call get_object ('MO:COEFFICIENTS%RHF')
      E_BSSE_AB=ENERGY_RHF%total
* Modify molecule
* Get Energy for A---* complex
      E_BSSE_Astar=ENERGY_RHF%total
* Modify molecule
* Get Energy for *---B complex
      E_BSSE_starB=ENERGY_RHF%total
* Compute BSSE free interaction energy
      E_BSSE_int=E_BSSE_AB-E_BSSE_Astar-E_BSSE_starB
      write(uniout,'(a,f12.6)')'E_interaction BSSE-free = ',E_BSSE_int
      write(uniout,'(a)')'WARNING> BLD_ENERGY_BSSE: NOT CORRECT ENERGY!!'
      write(uniout,'(a)')'BSSE not available at this time'
      stop'BSSE not available at this time'

*
* End of routine BLD_ENERGY_BSSE
      call PRG_manager ('exit', 'BLD_ENERGY_BSSE', 'ENERGY:BSSE%RHF')
      RETURN
       END
```

5.4 Conclusions and future work

Redesigning the major infrastructure of MUNgauss to work in a more modular way and to take advantage of many of the new features of Fortran 90 has greatly improved the manageability of the program. The design protocols that have been implemented have improved the codes readability and has given the code a consistent look and feel, which have made code development much more programmer friendly. The new infrastructure has provided a good backbone for adding additional functionality and the process to add the functionality is a straightforward one.

The decision to use Fortran 90 as the language for this project proved to be a wise one. Fortran 90 allowed the conversion of old MUNgauss code to be a gradual process without long periods of downtime. Many of the new features of Fortran 90 allowed for better memory management (allocatable arrays) and cleaner code (modules, select cases, etc.).

The addition of a *program manager* routine, along with the creation of a unique object number, has afforded many useful features. Timing routines, debugging the program, tracing execution and error detection are all now available through the *program manager*. In addition the *program manager*, provides many of the needed tools for future work. The creation of a dependency matrix could eventually lead to parallelization of MUNgauss. The timing tools can help programmers identify inefficient portions of the program and guide them towards optimization of the entire program.

MUNgauss will continue to evolve. It is a dynamic program that is continually improving its functionality. The new protocols and design has made this process much more programmer friendly and hopefully will act to accelerate this evolution.

References

1. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, Inc., Englewood Cliffs, 1988.

2. Cooper Redwine. *Upgrading to Fortran 90.* Springer-Verlag New York, Inc., 1995.

3. F. Colonna, L. Jolly, R.A. Poirier, J.G. Angyan, G. Jansen. OSIPE - a tool for scientific programming in FORTRAN. Comp. Phys. Comm. 81 (1994) 293-317.

4. A. Szabo and N. S. Ostlund. *Modern Quantum Chemistry; Introduction to Advanced Electronic Structure Theory.* McGraw-Hill, Inc., 1996.

5. I. N. Levine. *Quantum Chemistry: Fifth Edition.* Prentice-Hall, Inc., Upper Saddle River, 2000.

6. R. Davies, A. Rea and D. Tsaptsinos. *Introduction to Fortran 90, QUB.* Retreived January 9, 2002 from, http://www.pcc.qub.ac.uk/tec/courses/f90/stu-notes/F90_notesMIF_1.html

7. Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith and

Jerrold L. Wagener. *Fortran 95 Handbook, Complete ISO/ANSI Reference.* The MIT Press, 1997.

8. Larry R. Nyhoff and Sanford C. Leestma. *Introduction to Fortran for Engineers and Scientists.* Prentice Hall, Inc., 1997.

9. A. C. Marshall, J. S. Morgan, J. L. Schonfelder. *Fortran 90 Course Notes.* University of Liverpool, 1997.

10. ACES II is a program product of the Quantum Theory Project, University of Florida. Authors: J.F. Stanton, J. Gauss, J.D. Watts, M. Nooijen, N. Oliphant, S.A. Perera, P.G. Szalay, W.J. Lauderdale, S.A. Kucharski, S.R. Gwaltney, S. Beck, A. Balková D.E. Bernholdt, K.K. Baeck, P. Rozyczko, H. Sekino, C. Hober, and R.J. Bartlett. Integral packages included are VMOL (J. Almlöf and P.R. Taylor); VPROPS (P. Taylor) ABACUS; (T. Helgaker, H.J. Aa. Jensen, P. Jørgensen, J. Olsen, and P.R. Taylor).

11. CADPAC: The Cambridge Analytic Derivatives Package Issue 6, Cambridge, 1995. A suite of quantum chemistry programs developed by R. D. Amos with contributions from I. L. Alberts, J. S. Andrews, S. M. Colwell, N. C. Handy, D. Jayatilaka, P. J. Knowles, R. Kobayashi, K. E. Laidig, G. Laming, A. M. Lee, P. E. Maslen, C. W. Murray, J. E. Rice, E. D. Simandiras, A. J. Stone ,M.-D. Su and D. J. Tozer.

12. "General Atomic and Molecular Electronic Structure System" M.W.Schmidt, K.K.Baldridge, J.A.Boatz, S.T.Elbert, M.S.Gordon, J.H.Jensen, S.Koseki, N.Matsunaga, K.A.Nguyen, S.Su, T.L.Windus, M.Dupuis, J.A.Montgomery J.

Comput. Chem., 14, 1347-63(1993).

13. Gaussian 98 (Revision A.11.3), M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, V. G. Zakrzewski, J. A. Montgomery, Jr., R. E. Stratmann, J. C. Burant, S. Dapprich, J. M. Millam, A. D. Daniels, K. N. Kudin, M. C. Strain, O. Farkas, J. Tomasi, V. Barone, M. Cossi, R. Cammi, B. Mennucci, C. Pomelli, C. Adamo, S. Clifford, J. Ochterski, G. A. Petersson, P. Y. Ayala, Q. Cui, K. Morokuma, P. Salvador, J. J. Dannenberg, D. K. Malick, A. D. Rabuck, K. Raghavachari, J. B. Foresman, J. Cioslowski, J. V. Ortiz, A. G. Baboul, B. B. Stefanov, G. Liu, A. Liashenko, P. Piskorz, I. Komaromi, R. Gomperts, R. L. Martin, D. J. Fox, T. Keith, M. A. Al-Laham, C. Y. Peng, A. Nanayakkara, M. Challacombe, P. M. W. Gill, B. Johnson, W. Chen, M. W. Wong, J. L. Andres, C. Gonzalez, M. Head-Gordon, E. S. Replogle, and J. A. Pople, Gaussian, Inc., Pittsburgh PA, 2001.

14. Q-Chem 2.0: A high-performance ab initio electronic structure program,J. Kong, C. A. White, A. I. Krylov, C. D. Sherrill, R. D. Adamson, T. R. Furlani, M. S. Lee, A. M. Lee, S. R. Gwaltney, T. R. Adams, C. Ochsenfeld, A. T. B. Gilbert,G. S. Kedziora, V. A. Rassolov, D. R. Maurice, N. Nair, Y. Shao, N. A. Besley, P. E. Maslen, J. P. Dombroski, H. Daschel, W. Zhang, P. P. Korambath, J. Baker, E. F. C. Byrd, T. Van Voorhis, M. Oumi, S. Hirata, C.-P. Hsu, N. Ishikawa, J. Florian, A. Warshel, B. G. Johnson, P. M. W. Gill, M. Head-Gordon, and J. A. Pople, J. Comput. Chem. (2000) 21 , 1532-1548.

15. Spartan version 5.0. Wavefunction, Inc. 18401 Von Karman Avenue, Suite 370.

Irvine, CA 92612 U.S.A.

16. David A. Case, David A. Pearlman, James W. Caldwell, Thomas E. Cheatham III, Junmei Wang, Wilson S. Ross, Carlos Simmerling, Tom Darden, Kenneth M. Merz, Robert V. Stanton, Ailan Cheng, James J. Vincent, Mike Crowley, Vickie Tsui, Holger Gohlke, Randall Radmer, Yong Duan, Jed Pitera, Irina Massova, George L. Seibel, U. Chandra Singh, Paul Weiner, and Peter A. Kollman (1997), AMBER 5, University of California, San Francisco.

17. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations, J. Comp. Chem. 4, 187-217 (1983), by B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus.

18. HyperChem(TM), Hypercube, Inc., 1115 NW 4th Street, Gainesville, Florida 32601, USA

19. W. N. Celmaster. Modern Fortran Revived as the Language of Scientific Parallel Computing. *Digital Technical Journal.* Vol. 8, No. 3, 1996.

20. An Overview of WG5 and its work. Sept. 1999. Retrieved on Dec. 2002, from http://www.nag.co.uk/sc22wg5/overview.html

21. M. Watkins. *York Centre for Laser Spectroscopy; Electronic Structure Methods.* Department of Chemistry, The University of York. Retrieved Dec. 2002 from, http://www.york.ac.uk/res/ycls/theory/notes.shtml