An Introduction to Object-Oriented Analysis and Design

Andrew Moncrieffe

X52.9267

TABLE OF CONTENTS

Preface 9
Intended Audience 9
How the Book is Organized 9
Trademarks 10
Comments 11
Introduction 12
History 13
Chapter 1 17
Introduction to Object-Oriented Analysis, Design and Programming 17
Evolution of Software Engineering 17
Procedural Language Issues 18
Object Oriented Development 19
Comparison of Procedural and Object-Oriented Methods and Languages 20
Object-Oriented Analysis, Design and Programming Explained 21
Analysis 21
Design 23
Object-Oriented Capabilities and Benefits 24
Basic Concepts of Object-Oriented Development 24
Sample Project 26
Chapter Summary 27
Exercises 28
Chapter 2: Classes and Objects 29
Classes and Objects 29
The Meaning of the Word "class" 29
Identification of Typical Classes 30
Class Semantics 32
So, What is an Object? 32
Objects as Class Instances 33
Identification of Typical Objects 33
Object Features 33
What Makes Languages and Methods Object-Oriented? 34
Procedural and Object-Oriented Language Comparisons Revisited 34
Main elements of OO paradigm 34
Abstraction 35
Encapsulation 38
Hierarchical Relationships 41
Modularity 46

Persistence ... 48
Benefits of OO Development ... 50
Interfacing with Non-Object-Oriented Systems ... 51
Identifying Classes ... 51
Behavior Analysis ... 52
Domain Analysis ... 52
Use-Case Analysis ... 53
Informal English ... 54
Structured Analysis ... 54
Finding Key Abstractions ... 54
CRC Cards ... 55
Sample Project ... 56
Requirements ... 56
Chapter Summary ... 58
Exercises ... 59
Chapter 3 ... 60
Class Structure ... 60
"Design-time" and "run-time" Defined ... 60
What is Class Structure? ... 61
Abstract Classes ... 64
Class and Object Interactions ... 66
How Classes Determine the Behavior of Objects ... 67
Introduction to Class Modeling using UML ... 67
History of UML ... 68
UML Notation ... 68
Benefits of Class Modeling ... 71
Modeling Activities ... 72
Sample Project ... 72
Analysis ... 72
Chapter Summary ... 79
Exercises ... 80
Chapter 4 ... 81
Class Relationships and Interactions ... 81
Class Hierarchies ... 81
Class Associations ... 89
Which Relationship do we Choose, When? ... 91
Interfaces vs. Implementation ... 92
What is the Interface of a Class? ... 92
Implementing a Class' Functionality ... 92

Encapsulation and Information Hiding 93
Sample Project 96
Chapter Summary 103
Exercises 104
Chapter 5 105
Object Structure and Relationships 105
What is an Object? 105
Structure of an Object 105
Instance Fields 106
Class Fields 106
Methods 107
Object Initialization 108
Constructor Usage 109
Object De-initialization 110
Objects and Access Levels 111
Class-Object Relationships 111
Objects and Inheritance 111
Subclass Initialization 113
Object Interactions and Relationships 114
Modeling Object Behavior at Run-Time 114
Static vs. Dynamic Modeling 122
Sample Project 123
Chapter Summary 129
Exercises 130
Chapter 6 131
Designing with Classes and Objects 131
Overview 131
Design Guidelines 132
Abstraction 134
Refining Class Selections 134
Design Goals 143
Additional Design Factors 145
Design Elements 148
Sample Project 153
Constructors 159
Destructors 163
Checkpoint 163
Persistence and Data Management 169
Student-Related Data 174

System Functionality and Report Requirements 179
Implementing Data Management Methods 181
Class Details Revisited 182
User Interface 186
Summary 201

Chapter Summary 205

Exercises 206

Chapter 7 207

System Development Processes 207

What is a Software Development Process? 207

The Software Development Process 207
Conceptualization and Requirements Gathering 208
Analysis 209
Design 215
Additional Development Phases 219
Why do we Need a Process? 221

Chapter Summary 225

Exercises 226

Chapter 8 227

Creating and Using Object Oriented Software Interfaces 227

Interfaces vs. Implementation 228

Applications in OO Design 228

Polymorphic Behavior and Interfaces 230

Interfaces in UML 231

Support in OO Development 232

Interfaces in Distributed Systems 233

Sample Project 235

Chapter Summary 237

Exercises 238

Chapter 9 239

Object-Oriented Software Architecture 239

What is Software Architecture? 239

Object-Oriented Architectural Elements 240

Designing with Components 241
Using Components 242
Components and Distributed Systems 244

Components in UML 246
Sample Project 249
Architecture 250
Chapter Summary 253
Exercises 254
Chapter 10 255
Object-Oriented Methodology in the Industry 255
Requirements Gathering 256
Code Reuse 257
Components and Reuse 259
Code Maintenance 259
Components and Maintainability 260
Object-Oriented Technology at Work 261
Rational Unified Process 261
Rational Rose 262
Object-Oriented Databases 262
Chapter Summary 264
Appendix 1 265
Use Cases 265
Use Case Models 266
Appendix 2 268
Brief UML Reference 268
Introduction 268
UML Diagram Types 270
Appendix 3 287
Object-Oriented GUI Design Elements 287
Glossary 292
Bibliography 295
Index 297

TABLE OF FIGURES

Fig 1.1 Early computer programming 14
Fig 1.2 Shared memory access in typical procedural languages 18
Fig 1.3 Memory access with objects 19
Fig 2.1 Class "Computers" 31
Fig 2.2 Class "Modes of Transportation" 32
Fig 2.3 Abstraction 36
Fig 2.4 Abstraction 37
Fig 2.5 Abstraction 38
Fig 2.6 Encapsulation 41
Fig 2.7 Inheritance 42
Fig 2.8 Inheritance 43
Fig 2.9 Inheritance 43
Fig 2.10 Inheritance 44
Fig 2.11 Composition 46
Fig 2.12 Modularity 48
Fig 2.13 Options for saving a document 50
Fig 3.1 UML Notation 69
Fig 3.3 UML Class Diagram 71
Fig 4.1 UML Class Diagram 84
Fig 4.2 Associations 91
Fig 4.2 Student Diagram 97
Fig 4.3 Student Diagram 98
Fig 5.1 UML Sequence Diagram 116
Fig 5.2 Annotated UML Collaboration Diagram 119
Fig 6.1 Database Abstraction 151
Fig 6.2 Parameterized class 153
Fig 6.1 Student class relationships 168
Fig 6.2 System class relationships 169
Fig 6.3 Entity-Relationship diagram for student data 173
Fig 6.4 Additional entities 174
Fig 6.5 Student-related focus 184
Fig 6.6 System-related focus 185
Fig 6.7 Main menu 188
Fig 6.8 File sub-menu 189
Fig 6.9 Adding a new student 190
Fig 6.10 Adding a Typical student 191
Fig 6.11 Adding a Faculty student 192
Fig 6.10 Adding a Transfer student 193
Fig 6.11 Find Student menu item 194
Fig 6.12 Search criteria 195

Fig 6.13 Search Results 196
Fig 6.14 View student details 197
Fig 6.16 Tools menu 199
Fig 6.16 Help menu 200
Fig 7.1 Analysis 214
Fig 7.2 Design 218
Fig 7.3 Implementation 220
Fig 7.1 Task Sheet 223
Fig 7.2 Gantt chart 224
Fig 8.1 UML 231
Fig 8.2 Interface illustration 232
Fig 8.3 CORBA IDL example 235
Fig 9.1 UML components 246
Fig 9.2 Component interactions 247
Fig 9.3 Component interactions 247
Fig 9.4 Component interfaces 248
Fig 9.5 Multiple interfaces 248
Fig A1.1 Use case example 266
Fig A2.1 Some GUI elements 289
Fig A2.2 Some more GUI elements 290
Fig A2.3 Simple Windows application 291

Preface

Intended Audience

This book is intended for students seeking an introduction to the object-oriented way of thinking and how that is achieved through analysis and design activities. It is not mandatory that you have prior experience programming in an object-oriented language such as C++ or Java. It is expected that you have some familiarity with high-level languages, as throughout the book, many references are made to familiar high-level language constructs. As such, familiarity with languages such as BASIC, C, COBOL, FORTRAN, PL/1, Pascal, etc., is sufficient. However, any familiarity with object-oriented languages should reduce your learning curve.

How the Book is Organized

The basic concepts of object-oriented analysis and design are covered in Chapters 1 to 6. Chapters 7 to 9 introduce more advanced concepts such as the development process, object-oriented architecture and distributed computing. Throughout the book, UML is used for class and object modeling.

Chapter 1 contains an introduction to object-oriented concepts starting with tracing the evolutionary steps of software engineering. Chapters 2 to 5 introduce the concepts of classes and objects, progressing through discussions of more complex relationships. An example development project is introduced in Chapter 1. At the end of each of Chapters 2 to 6, the Chapter's material is applied to the example, evolving it from the requirements through analysis and design. Here, the focus is on the applying the concepts introduced in each chapter. Chapter 6 discusses the elements of design and the evaluation of design. This chapter also discusses "connecting" user interfaces to object models. At the end of the chapter, these are

applied to the example. While not a book on programming, where appropriate, there are examples in both C++ and Java, in addition to pseudocode.

In Chapter 7, we take all the activities done so far in the previous chapters and formalize them into an overall lifecycle partitioned into phases. The discussion is focused on which activities are allocated to which phases and why.

Chapter 8 discusses interfaces, as "independent" abstract constructs, not just as a description of the public operations defined in an abstraction. Interfaces, as used in distributed computing are reviewed also. The chapter ends with the definition of interfaces in UML for the example.

In Chapter 9, we discuss software architecture and distributed systems. The chapter provides a definition of architecture and discusses various architectures. Component architectures are also discussed. At the end of the chapter, we evolve our example's architecture to be component based.

Chapter 10 discusses how the attributes of the object-oriented methodology address some of the issues facing IT managers. It ends with a brief discussion of some products available.

Trademarks

Various products of various companies are mentioned throughout the book. These names are trademarks of the companies, as follows:

Java is a trademark of Sun Microsystems, Inc. in the U.S. and other countries. The Java technology is owned and exclusively licensed by Sun Microsystems, Inc.

Microsoft, Windows, Windows NT, Project and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries or both.

Rational Unified Process and Rational Rose are trademarks of Rational Software Inc. in the U.S. and worldwide.

Other company, product and service names may be trademarks or service marks of others.

Comments

Comments, criticisms, suggestions or any other feedback is welcomed. Please send email to the address below:

mracmny@gmail.com

Introduction

Though not a recent innovation strictly speaking, object-oriented development is now enjoying immense popularity. This popularity has been bolstered by the availability and maturity of object-oriented tools such as languages and environments. This popularity has also been aided by the presence of the World Wide Web.

At the same time, more and more corporations are faced with the challenging problem of integrating diverse platforms, as they seek to gather critical information for client and decision support systems. As a result, we focus not only on the elements of the object-oriented methodology, but also on how to integrate non-object-oriented systems into object-oriented development efforts.

The use of the World Wide Web (and related technologies) as a delivery medium for computing has caused a major shift in how applications are designed and deployed. In addition to developing object-oriented systems, more and more corporations are developing distributed systems as well. Many products have come to market, based on this paradigm shift. It is important then, to extend the discussion of object-oriented development to include component based development.

Even with all of these advancements, the old adage still holds true: in order to know where you're going, you must know where you've been. With that in mind, a brief exploration of the evolution of software engineering is in order, to give some insight into why object-oriented development evolved (and from what) in the first place.

In our world today, we all want everything yesterday. Development projects are much the same. In many cases, we are faced with management and market pressure to deliver systems ASAP, which means yesterday. Regardless, we need to develop sufficient

understanding of the fundamentals before we apply to our analysis and design efforts. To that end, we discuss in some detail, the formalities and theories behind the activities required to develop object-oriented software and follow that discussion with application to a class example. The "formality" will provide a basis that is somewhat portable. Languages that support object-oriented development may have different syntaxes, but the fundamentals should apply, regardless of language.

In a similar vein, it is important to make sure we agree on the semantics of the words we use. Many people refer to systems as object-oriented. However, as we will see, not all "object-oriented" systems are indeed object-oriented. We examine what it means to be object-oriented. There are similar issues with the terms "analysis" and "design". In some cases, we've used those terms for years, without stopping to consider exactly what we're doing in each case. As before, we need to have the same understanding of the meanings of the words we're using, in order to prevent ambiguity and misunderstandings.

As with other paradigms, productivity has been and continues to be an issue. One of the requirements for acceptance is how quickly developers et al "get up to speed".

Like anything else, to obtain the most from object-oriented paradigm and methodologies, there must be a basic understanding of the concepts. This necessitates viewing problem-solving in a completely different way than one might be used to. In some cases, it will require learning completely new languages, in addition to the new concepts.

History

To put all of this in the proper context, a brief history lesson is in order. In the beginning, there was the mainframe, and it was good. It was also huge, requiring its own room, complete with specific cooling mechanisms. In one sense, programmers were at the mercy of the mainframe, as at this time, the interactions between programmers and the mainframe were based on the language of the mainframe, i.e. zeroes and ones. Imagine how difficult it must have been to create a trivial program, let alone anything complicated. The I/O devices were rudimentary at best. Computer programs involved directly submitting programming instructions via switches on a rudimentary panel (or panels). This was the interface to the computer – no mice and keyboards. The programs were essentially setting or

re-setting switches on or off, translating into commands for the computer. Of course, we now have the advantage of decades of improvements on computing, but this is how it was done in the beginning. Just think of how "interesting" it must have been to code and debug a program back then, with no monitors, keyboards, mice, IDE's (Integrated Development Environment) etc.

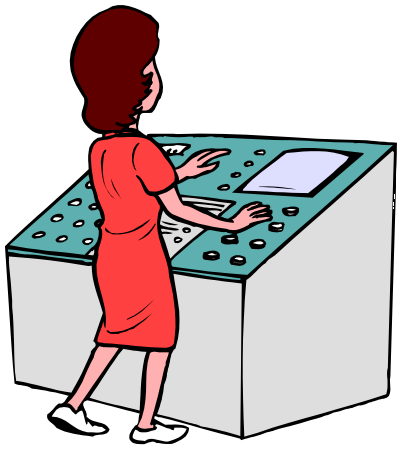


Fig 1.1 Early computer programming

Though primitive and obviously tedious, this method allowed programmers to program the computer to solve problems. Prior to this, they were not able to employ and leverage computing solutions. It is also clear that with this level of sophistication, the computer programming methods were only suitable for solving relatively simple problems. As you can imagine, this method would not (and did not) scale well as problems became more complex. Thus, over time, the industry realized that another paradigm, or way of thinking, was needed, because the problems that were being presented were becoming increasingly complex, outstripping the capabilities of the solutions (and programs).

This led to the introduction of first-generation languages (1GL), such as COBOL[1]. These languages provided a more easily adapted (for humans anyway) interface between humans and the computer. This interface was the high-level, text based language. This also meant the advent of compilers and interpreters. These were programs that would accept text files with each line having an English-like structure and translate it into binary code, which the computers understood. This allowed for an order of magnitude leap forward in terms of the complexity of the problems that could now have computing solutions. High-level languages also gave programmers the ability to declare

[1] Common Business Oriented Language

variables. A variable represents a location in memory. High-level languages allowed programmers to refer to locations in memory by symbolic names. This made it easier to manipulate memory using meaningful names and not being limited to manipulating memory addresses directly all the time. We take these things for granted as it is commonplace now. At the time, it was a significant leap forward.

High-level languages evolved to second-generation languages (2GL), which added the ability to subdivide programs into subroutines or procedures. This was the advent of procedural languages. The arrival of the subroutine meant that "divide and conquer" could be applied to larger problems, some of which may have been prohibitively large before. This went a long way with regard to the complexity of problems that could now be effectively solved. These constructs allowed hierarchical decomposition of problems into more manageable components, each of which could be further subdivided.

Hierarchical decomposition is also termed "algorithmic decomposition". It is the cornerstone of the top-down design methodology that so many of us were taught in our beginning courses in Computer Science. With hierarchical decomposition, complex problems, orders of magnitude greater than undertaken (and solved) previously, were being dealt with, somewhat more routinely. This became more commonplace as the procedural paradigm evolved further. Another effect of this division of labor was the utilization of modularity in computing. Sections of a program's code (subprograms) could be used, and re-used.

Further evolution saw the progression to third-generation (3GL) and fourth-generation (4GL) languages. As they evolved, high-level languages provided other constructs, such as data types. A language defined a set of types that were considered primitive because they were inherently supported by the language. This meant that programmers could choose individual types that were appropriate for their programming efforts. They could decide that they needed integer numbers only for some value and once this declaration was made, the compiler would take care of the amount of memory that needed to be allocated for that particular type. This freed the programmers from such tasks as well. Languages also allowed users to create their own data types by combining and/or renaming the primitive types. By utilizing abstract data types (ADT's), users are able to define types that were abstractions of elements of the problems they were trying to solve. The programmers could use language elements from the requirements, making their code easier to

understand and the solutions easier to conceptualize. These allowed the aggregation of primitive types (i.e. integers, characters, etc.) in a way that was more meaningful to human designers and coders. It also allowed for more readable and organized code. Examples of user-defined types are structs in C, records in Pascal, etc. This progression also saw the appearance of support for modular programming and data manipulation.

So, if procedural programming was so great, why was there a new paradigm introduced, i.e. object oriented development? Why was this new paradigm invented in the first place? What problems, existing or perceived, were the inventors and designers attempting to solve?

Chapter 1

Introduction to Object-Oriented Analysis, Design and Programming

Object-oriented analysis, design and programming evolved to address shortcomings in other methods of software development. As more complicated problems were undertaken, the early methods of software development were proven to be insufficient. The object-oriented way of thinking evolved to help in this area.

This chapter introduces the fundamental concepts of object-oriented analysis (OOA), design (OOD) and programming (OOP). It also describes how object oriented languages differ from procedural languages.

The text is titled "Object Oriented Analysis and Design". Our main concern in this chapter will be the concepts, tools and guidelines of object-oriented methodology. We will explore the analysis and design activities as they relate to the overall object-oriented system development lifecycle. However, in order to do this, we must establish the context in which we will interpret various terminologies. So, to paraphrase, in order to know where we're going, we have to know where we've been.

Evolution of Software Engineering

Computers and Computer Science has now been around for a while. We utilize various tools and techniques on a daily basis, taking some things for granted. In order to put object-oriented development in the proper perspective, we should look briefly at what led to the development of object-oriented approaches.

Procedural Language Issues

Here are some general issues with procedural languages. Please note: these are not absolute, and are thus open to dispute. Why? The answer is procedural programmers have attempted, with varying levels of success, to deal with these issues, within the constraints of the particular language, environment and approach. Some have been more successful than others due to the particular language and the "best practices" that have been invented to reduce some of these issues. We will highlight three of these, as follows.

1. Programs (consisting of subroutines and modules) have unrestricted access to shared (common) data. Procedural languages have a separation between data and the procedures that manipulate the data. So encapsulation, with regard to data, (i.e. the concept of "private" data) is not robust for procedural languages. Even if there is data in a `record` (in Pascal) or a `struct` (in C), the data in these structures is publicly accessible, meaning it can be assigned to any other variable of the appropriate type and similarly, any variable of the appropriate type may be assigned to it. There is no access restriction on this data, even though it is obviously important to the overall structure. In addition, the data could be remotely manipulated (via pointers, etc.).

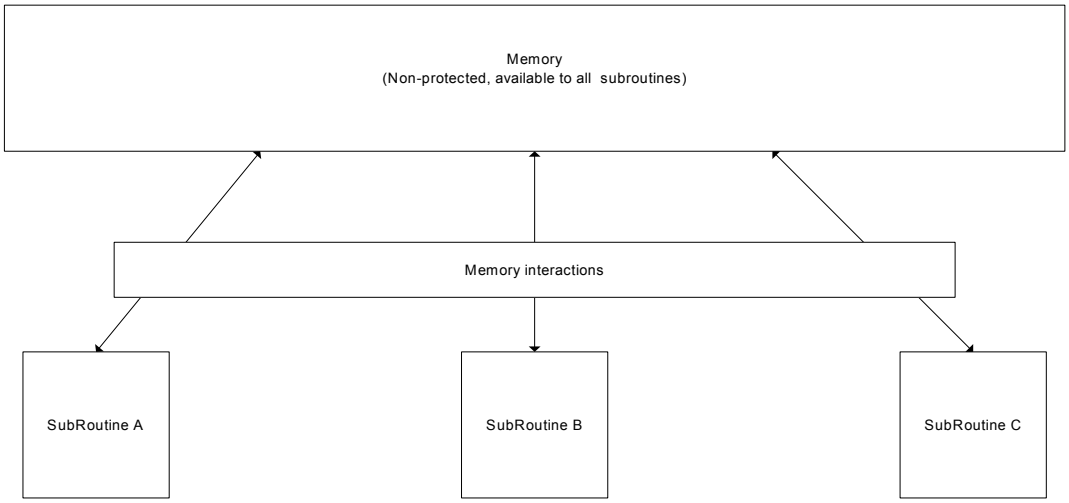


Fig 1.2 Shared memory access in typical procedural languages

2. Code reuse, except for copying the code of particular procedures and modules, was largely un-realized.

3. Programmers were unable to apply concepts based on elements present in real life, such as leveraging the relationship hierarchies between items, whether data elements or algorithms. They could create "static" types to represent items in the real world, but that's where it stopped.

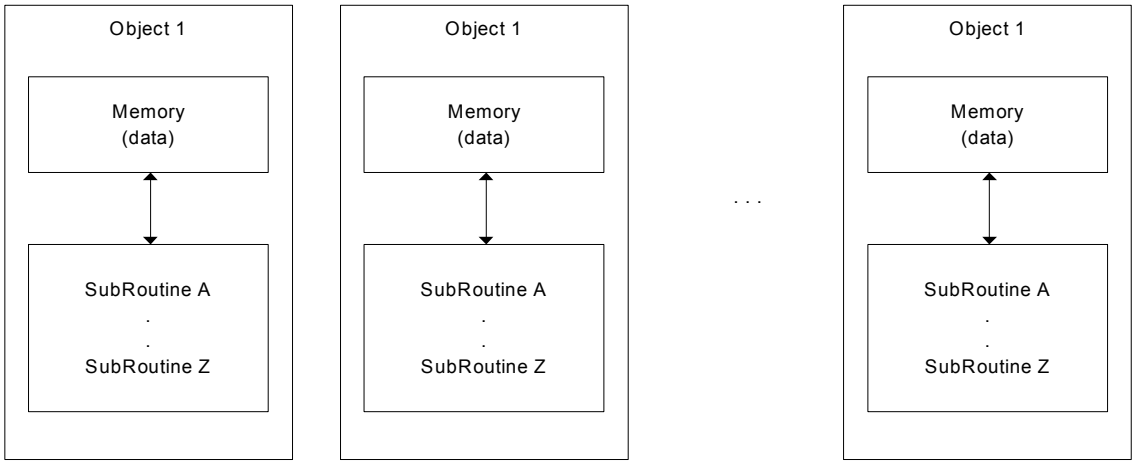


Fig 1.3 Memory access with objects

Object Oriented Development

Object-oriented development has been around since the 1960's. Object-oriented languages are not new. The concept of the "object" was first introduced in Simula67. Simula was developed for use in creating simulations of real-world systems. Many of these systems were highly complicated, involving many moving parts. Simula introduced programmers to objects and classes. This makes Simula very important in our current discussion. Objects in Simula were allowed to have their own behavior and data its objects represented real (i.e. physical) objects.

Object-oriented development has recently become very popular in IT shops. In fact, this approach has probably become more widely adopted due to its close association with Web development (primarily due to Java being object-oriented).

Why has there been this (relatively) slow adoption? There are many reasons, a few of which are as follows: There's a significant amount of existing and fully operational software that is not based on object-oriented development methodologies or use object-oriented languages. This software is also, in many cases, absolutely mission-

critical, in addition to relatively reliable. In addition, there's much more available expertise and familiarity with procedural methods and languages. This is related to the point above. We also have to factor in organization's reluctance to change. This change sometimes involves, among other things, significant retraining/learning curve of new paradigm for existing resources and management. This is compounded by the opportunities for misunderstanding that exist regarding object-oriented methodologies.

Comparison of Procedural and Object-Oriented Methods and Languages

Smalltalk appeared after Simula. Smalltalk came along in the 1970's. It is a pure object-oriented language as every facet of the language uses objects. For example, even types that are primitives (non-object-oriented) in other languages, are represented by objects in Smalltalk. As a result, it is impossible to write a program in Smalltalk that is not object-oriented.

C++ was developed after Smalltalk. Although Smalltalk existed before C++, C++ is credited as being the first mainstream object-oriented language. C++ maintains backward compatibility with C. Unfortunately this means, it is possible to write non-object-oriented C++ code. As a result, while C++ does provide native object-oriented constructs in the language, it is not a pure object-oriented language.

Java (from Sun Microsystems) was originally designed for consumer electronics such as set-top boxes. At its origin, it was called Oak. Oak was not a commercial success for Sun. Oak was based on C++ to the extent that it kept the most important aspects of that language and discarded the most troublesome. It was also designed with portability and security in mind. As a result, pointers (among other items) are not offered in Java. With the advent of the World-Wide Web, Sun renamed Oak Java and created the HotJava browser. Java 1.0 was officially released in 1996. Some in the industry argue that since Java has non-object oriented primitive types, it is not a pure object-oriented language either.

In addition to these, Microsoft's suite of object-oriented development languages now includes Visual Basic.Net and C#, both of which provide object-oriented features.

Object-Oriented Analysis, Design and Programming Explained

Now that we have an understanding of the "why", let's go further and look at some specific terminologies. In this text, we will discuss object-oriented analysis, design and programming. Let's look at each one individually.

Analysis

What is meant by Analysis? Analysis is one of the elements, i.e. phases of an overall software development process (lifecycle). In Analysis, we create the high-level models of the system, based on requirements. These models are based on an understanding of what functionality the system is to provide. From these requirements, we are expected to develop a system that provides all of the necessary functionality. Indeed, we are expected to produce a system that meets the users' expectations of more than just functionality.

The analysis effort involves taking functional requirements and developing a depiction (model) of the system. For object-oriented development efforts, this means preparing an object-oriented decomposition that satisfies the requirements. The activities here help transform the requirements of the system into a design that can be realized by software.

Strictly speaking, in Analysis, we are not concerned with implementation details, i.e. how the system will be implemented. Rather, in Analysis, we need to focus on the functional aspects of a system, i.e. the information conveyed by the functional requirements. As we will see, functional requirements are not the only requirements that may exist for a system.

In Analysis, our goal is to "digest" the requirements and produce a set of documentation on which to base the design. Analysis documentation reflecting the functional aspects of the system, provides a static view of the system (more on this later). The activities of analysis help transform the requirements of the system into a design that can be realized by software. The models of analysis are at a level of abstraction above the physical implementation of the system. The level of abstraction is such that the models could be applied equally well to many different platforms and architectures. Analysis ignores the architectural constraints of the system. The purpose of Analysis is to ensure that some aspect of the system

satisfies each functional requirement. For object-oriented analysis, this documentation will include class diagrams, which show the object-oriented decomposition.

Requirements

A requirement is a description of a feature of a system, with systems typically having many features. An example of a requirement is a description of the functionality to be provided by the system (functional requirements). Another example is a description of the constraints under which the system must operate (non-functional requirements).

In general, a system's requirements are the set of documentation (of one sort of another) that sufficiently specifies the functionality and operations of the system. Requirements may be grouped into two categories, functional and non-functional requirements.

Functional Requirements

Functional requirements are the most popular, by far. At some point or another, it is very likely that we have received functional requirements[2]. Indeed, some of us, at one point or another, may have created functional requirements as well.

Functional requirements describe the actions of the system, i.e. the functionality to be provided by the system. These requirements typically describe the input to and the result of these actions. Another way of saying this is that the functional requirements describe how the system should behave in response to various inputs, whether from users or other systems. Given this, functional requirements include (but are not limited to) use-cases.

Use Cases

A use case[3] is a description of an interaction between users and the system (the term users in the context of a use-case includes people and external systems). This interaction between the user and the system is called a scenario. Each use-case has a primary (i.e. positive) scenario and could have many alternate scenarios that could cover areas such as what to do when exception conditions are encountered in the execution of the primary scenario.

Non-Functional Requirements

As we mentioned before, in addition to the functional requirements outlined above, there are other requirements that must be taken into

[2] The functional requirements are also used to develop the functional specifications of a system.
[3] Use cases are reviewed in more detail in Appendix 1.

consideration when constructing a system. These, as a group, are termed non-functional requirements, as they specify system features other than functionality. Non-functional requirements represent expectations or constraints that affect the system's operation, not what functionality it provides. These requirements include elements such as environmental constraints, performance, usability, availability (robustness, reliability, redundancy, etc.) and security. There are also requirements that specify hardware and installation/deployment. As with functional requirements, various forms of documentation may comprise the set of non-functional requirements for a system.

Design

Practically speaking, it is difficult to examine Design[4] completely separately from Analysis as they are tightly coupled. In practice, they are not like the "waterfall"[5] model would suggest. In the "waterfall" model, each phase[6] of a development project is completed before the next one is started. In fact, phases of a development project tend to be more iterative, which does add to the project management challenge. Indeed, in many cases, the design activities occur almost in tandem with those of analysis.

In the Design phase, we take the high-level models from the Analysis (class diagrams, etc.) and make them more concrete by factoring in the environment, constraints, non-functional requirements, cost, time-to-market, etc. The output of Design is a set of models that are the basis for writing code. Design is a step further along toward implementation.

In Design, we are specifying how the elements of the system that provide the functionality and satisfy constraints (non-functional requirements) will be implemented. In the Design phase, we refine and add more detail to our analysis models. We also try to depict the behavior of the system at run-time, via various diagrams that capture how the system as a whole (or pieces) behave over time and what interactions exist. We are able to do this because we have more information at our disposal, having gone through analysis. We make decisions such as which technologies will be used, what platforms will be used, etc. No doubt, these decisions will be heavily influenced by our non-functional requirements.

[4] Design is discussed in Chapter 6.
[5] "Waterfall" refers to the way projects are typically depicted in project plans, i.e. divided into sequential project phases where one phase is completed before the next phase begins
[6] A phase of a project represents a major unit of work for a project and is comprised of individual tasks

Object-Oriented Capabilities and Benefits

Benefits

Obviously, given the progression of procedural languages, from 1st generation to 4th generation languages, there are many proponents, users and uses for procedural languages. However, these shortcomings, are real and have far-reaching effects which grow in proportion to the size of the problem to be solved and many would also argue in proportion to the size (and number) of the teams participating in developing the solution.

Let's look at unrestricted access to shared memory as an example. Two subprograms need to manipulate data (not local to each subprogram). Each subprogram has the ability to read and write data, thereby changing values in that particular data area. However, there are no restrictions on which memory areas are accessible and which are not. Errant code in one sub-program may overwrite the memory area or areas used by the other, unbeknownst to it. This is different from just changing the value of global data.

If you extend this simple example by assuming teams of programmers would be participating in the problems solution, then you can see that problems such as this could lead to many hard-to-find bugs in the software.

Capabilities

The quest to find ways, i.e. development methodologies and languages, to deal with issues such as these led to object-oriented development. The inventors of OO sought to improve the overall process of producing defect-free, robust code. Many of the issues listed above were thought of as weakening the ability to produce defect-free code. In addition, there were processes developed, which "wrapped" the OO paradigm to further guide developers in creating object-oriented software.

Basic Concepts of Object-Oriented Development

In object-oriented way of thinking, systems are comprised of collaborating "objects". Each of these objects "knows" what it has to do – it has a clear and distinct purpose, in addition to a clear and distinct lifetime. "Objects" are created as needed (in an orderly fashion) and provide a service through a well-defined interface. In addition, they are destroyed in as orderly a fashion as that in which they were created.

Briefly, in abstract terms, an object is a tangible entity that exhibits some well-defined behavior, has an overall and specific purpose, has a definable state and has a particular identity. Each of these will be explored further. An object also includes its data (attributes) and the means to manipulate this data (methods). Objects are also separate and distinct from each other.

Thus, object-oriented thinking (and thus, object-oriented development) is based on thinking in terms of objects and their interactions with each other. The overall methodology is based on recognizing and providing support for the following areas:

- Abstraction
- Hierarchical relationships
- Encapsulation
- Modularity
- Persistence

In order for object-oriented development to be useful, these elements (present to varying degrees in procedural languages) must also be present in object-oriented languages. As we progress, we'll explore how objects-oriented development addresses each of the "procedural language" issues listed earlier, using these elements.

Definitions

Following from above, Object-Oriented Analysis (OOA) is an analysis of the requirements that is based on object-oriented thinking. This is an analysis that yielded the object-oriented decomposition, as opposed to the top-down hierarchical decomposition of structured analysis. Similarly, Object-Oriented Design (OOD) is design that is also based on object-oriented thinking. In the design phase, we are concerned with making the models developed in the analysis phase more concrete and refined, ready for development. Object-Oriented Programming (OOP) is the development of programming code based also based on object-oriented thinking, but also uses an object-oriented language and environment (C++, Java etc.).

Sample Project

A picture is worth a thousand words. As an accompanying thought, an example is worth much also. So, as we progress through each of the 10 sessions, we will apply the knowledge in gained thus far to analyzing a set of requirements and designing a solution. As this is not a programming course, we will stop short of implementing the solution. However, many of the decisions that we will need to make in our design stages are dependent on the target platform(s). We will look at our design decisions in the context of different architectures, to see what influences are present. Whenever assumptions are made (and there will be some), we will clearly identify them. The actual exercise will be introduced in the next chapter.

Chapter Summary

- Analysis is the activity of taking the functional requirements and creating a model of the system. Object-oriented Analysis takes the functional requirements and produces an object-oriented decomposition (unlike structured decomposition).

- Design is the activity of specifying how to implement the elements of the system that provide functionality and satisfy constraints. Design takes the high-level models from Analysis and makes them more concrete by factoring in non-functional requirements as well.

- A requirement is a description of a feature. Functional requirements describe the actions of the system. Non-functional requirements reflect system constraints other than functionality such as performance, usability, etc.

- Object-Oriented Programming is the activity of creating an object-oriented program using an object-oriented programming language.

- Object-oriented methodology is not new. It has been around since the 1960's, starting with Simula67.

- The object-oriented methodology is based on recognizing and providing support for Abstraction, Hierarchy, Encapsulation, Modularity and Persistence.

Exercises

1. Based on your experiences, list any additional shortcomings of the procedural approaches to system development

2. Define and give examples of abstraction, modularity and typing in the context of procedural development

Chapter 2: Classes and Objects

Classes and Objects

Classes and objects are central to anything prefaced by the term "object-oriented", which, of course, includes object-oriented analysis, object-oriented design and object-oriented programming (OOA, OOD, OOP). In addition, as we will see later, the understanding and correct choice of what should (or shouldn't!) be a class is similarly central to having a good object-oriented design.

The Meaning of the Word "class"

As you probably have guessed, based on your prior understanding and usage of the word, a "class" is a group of items based on a set of shared and similar characteristics, which are exhibited by all members of the group. The characteristics of the class determine the expected behavior and features of its members. A class may also (loosely) be thought of as a set of elements.

There are some obvious parallels between set theory and the definition of a class. In set theory, we have sets, subsets and elements. Sets and subsets would most closely correspond to classes and subclasses,

respectively. An element of a set would most closely correspond to an object or instance of a class[7][8].

Identification of Typical Classes

Before we can create an object-oriented model of a system to be constructed, we must identify what specific building blocks we'll use. These building blocks are the classes that will be involved in our system. The selection of classes is critical to our progress. Let's examine how we'll do this.

Let's define a class called "Humans". Since, by definition, a class represents a grouping where all elements share characteristics, we would expect that our class Human to have characteristics (features and behavior) such as:
Representing all items considered human
Oxygen used for breathing
High degree of intelligence[9]
Ability to communicate
Ability to interact via our 5 senses
Etc.

It is clear that this is not an exhaustive list of human characteristics.

Let's define a class called "Automobiles". As above, membership in this class could be defined by the following:
Having a means of propulsion, such as an engine
Having a steering mechanism
Having a braking mechanism
Having accommodation for the driver
Having a mechanism to transform the output of the engine into movement (forward and backward), i.e. transmission, propeller shaft, differential(s), etc.

Obviously, again, this list of characteristics is not exhaustive. Equally obvious is the fact that these characteristics are very high-level characteristics. There are specialized versions of automobiles that exist – cars, trucks, construction equipment, farm equipment, etc. They can also be further specialized, i.e. for cars we could have

[7] In case you were wondering, procedural languages just have no class.
[8] This parallel between sets and classes is not absolute. For instance, with classes, there's nothing that parallels the intersections, unions and complements. However, the analogy of a set is a useful one in attempting to understand the relationship between objects and classes.

[9] May be an arguable point for some.

sedans, coupes, etc. However, you will notice that all of these, in general, share the high-level characteristics outlined above.

Let us think about the class "Computers". Such a class (see Fig 2.1) would be a very large one covering all computing devices. We would expect a member of the set of Computers to be a computing device, allow inputs, return outputs, have the ability to be issued programming instructions, have the ability to execute programming instructions, etc. This could be visualized as follows:

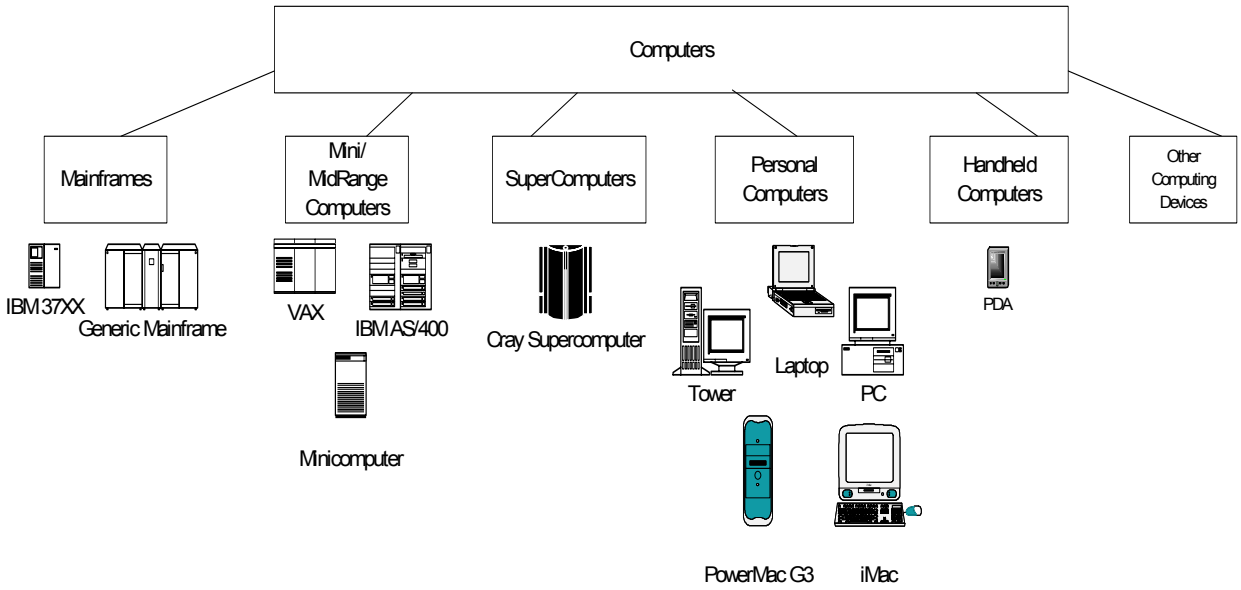


Fig 2.1 Class "Computers"

This diagram shows the class "Computers" and some members of the class. The class is large enough to be subdivided into additional classes (subclasses). Each of these classes is also large enough to be subdivided. In any case, some of the members of the classes are listed. Each member of the subclass is also a member of the class "Computers".

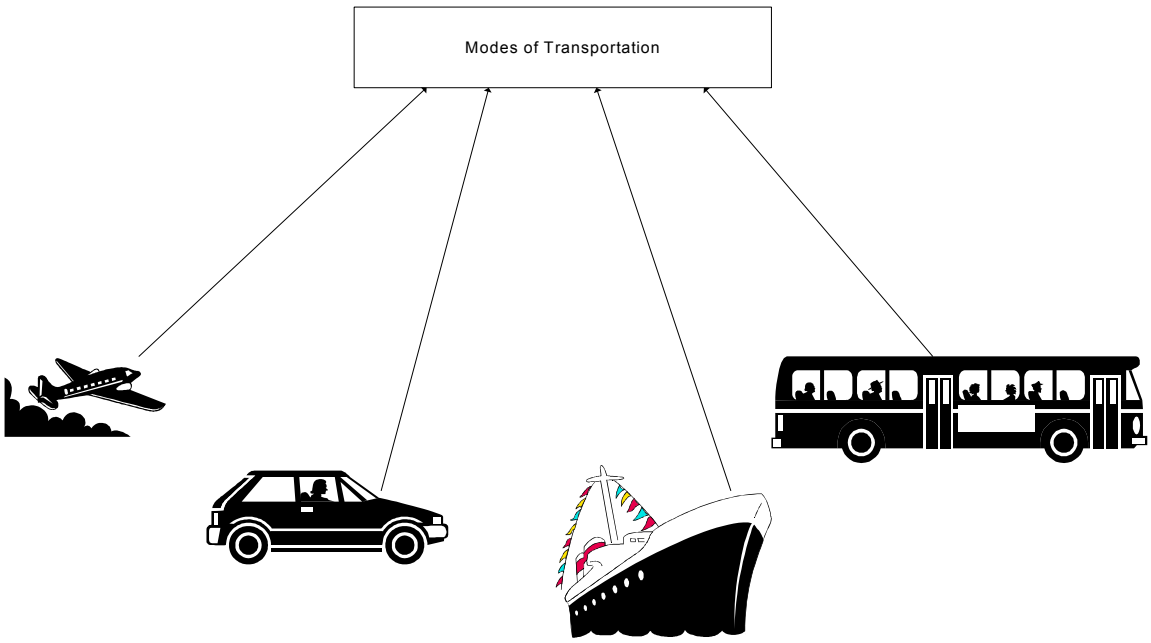


Fig 2.2 Class "Modes of Transportation"

Class Semantics

The description of a class, which outlines the behavior expected of each of its members, is, in effect, a contract between the class (and its designer(s)) and the people that interface with its members. A class may also be described as a blueprint for (and of) its members. Let's put this another way. When a class is defined, you are describing a set of attributes and behaviours that each and every member of that class will share. You should be able to predict what a member of a particular class will be able to do, based on the definition of the class. The phrase "what a member of the class will be able to do" is due to the semantics of the class. The semantics are the rules defined in each class and followed explicitly by each member.

In the prior section, we gave examples of classes and discussed what characteristics members would have. These characteristics are dictated by the semantics (or rules) of each class. As designers, we will define classes and rules.

So, What is an Object?

Above, we've repeatedly used the term "members" of a class. What are the members of a class? If we go back to our two examples earlier, the members of the class Human would be people, each one of which possesses human characteristics, i.e. features and behavior.

Each person (individual) exhibits the general set of characteristics that we've described and that are exhibited by all humans.

For automobiles, the members of that class, for the purpose of this discussion, are specific cars, trucks, etc.

Objects as Class Instances

Each member of a class is termed an instance of the class. An object is an instance of a class. From our earlier discussion, an instance of a class is a specific item that is representative of the class. Another perspective is to say that an instance of the class is one item that is completely and accurately described by the class' definition. This definition, as mentioned above, can be viewed as a contract between the class' designers and its clients (users).

Identification of Typical Objects

In the example above, we said the class Human could represent all people. Each person would exhibit the general set of characteristics that apply to all human. Each person could also be viewed as individual objects of the class Human. The class Human would effectively be the "blueprint" for each of the person "objects" that exists at any given time.

Object Features

As a specific instance of a class, objects may be described as having the following attributes:

Identity

In order for us to use an object, we must be able to uniquely identify one object from another. So, each object (instance of a class) must have a specific and unique identity. If we use our example of class Humans, of which people are the members or objects, each person has an identity. In many cases, fingerprints are used to verify someone's identity. Fingerprints are a universal and unique human attribute. There are other means of establishing identity as well, albeit not necessarily unique, such as a person's name. The social-security number is unique, but it is not universal.

Behavior

Each member of a class will exhibit the behavior outlined in the description of the class, based on the semantics of the class. Thus, it will be "well behaved", i.e. it will do what is expected, nothing more and nothing less. The *behavior* of an object is also referred to as its *functionality*.

State

As a specific instance of a class, an object will have a particular state at a given point in time. The state of an object is determined by assessing various factors (values or attributes) at that specific point in time. For example, the state of a car at a given point in time may involve assessing factors such as how much gas is in the tank, the condition of its brakes, tires, engine, etc. The state of an object is dynamic – it changes as time goes on.

So, an object is a specific instance (member) of a class, with the class providing the blueprint.

What Makes Languages and Methods Object-Oriented?

A language is considered object-oriented if it supports the major elements of the object-oriented paradigm. This means is provides language constructs for Abstraction, Encapsulation, Hierarchy and Modularity. Each of these is discussed below. If a language doesn't support one of these (typically inheritance), it is termed object-based. Languages such as C++ and Java support all of the attributes above and thus are termed "object oriented". Previous versions of Microsoft Visual Basic[10] were "object-based", as they did not fully support Hierarchy (as we'll describe below).

Procedural and Object-Oriented Language Comparisons Revisited

Procedural languages support and facilitate the implementation of subprograms (procedures and functions). These subprograms exist as a product of the decomposition of a problem into its sub-parts, i.e. successively refined subprograms (top-down decomposition). Object-oriented programming languages support (and facilitate) the implementation of classes, which are the products of the object-oriented decomposition of a problem.

Main elements of OO paradigm

In Chapter 1, we compared procedural and object-oriented approaches. We said that there were some advantages that the object-oriented approach had over procedural approaches. We saw

[10] The current version of Visual Basic includes full support for inheritance, thereby making it object-oriented.

that using objects gets us out of the potential problems of shared, unprotected memory. We also saw that objects are members of classes, which are groupings that we define. These (and other) benefits are obtained by using the object-oriented approach. The object-oriented approach provides these advantages because of its inherent support for the following elements:

1. Abstraction
2. Encapsulation
3. Hierarchy
4. Modularity
5. Persistence

A description and example of each of these is below.

Abstraction

Whenever someone delivers something, i.e. draws you a picture – some sort of diagram or gives you a text document that doesn't focus on the details, i.e. high-level, they are using abstraction. With abstraction, we are able to categorize items (areas of a problem space, for example) into manageable chunks, each of which is easier to deal with than the whole. We can then determine how these chunks interact to provide the entire solution. Abstraction is one of the ways we deal with complexity on a daily basis. An abstraction allows us to focus on the significant areas of a problem (or system), without focusing on all of the details of the problem (or system) all at once. This is what makes those "chunks" manageable. We are able to visualize the problem more easily, as we are not trying to see everything at once, only the parts of interest and relevance.

Using abstraction, we can identify the classes that we use to describe a system. Each of these classes will have a semantics that govern the role of its objects in the overall system. With abstraction, we can define these classes and include enough information, without having to include every possible bit of data for the class. We'll see more of this later.

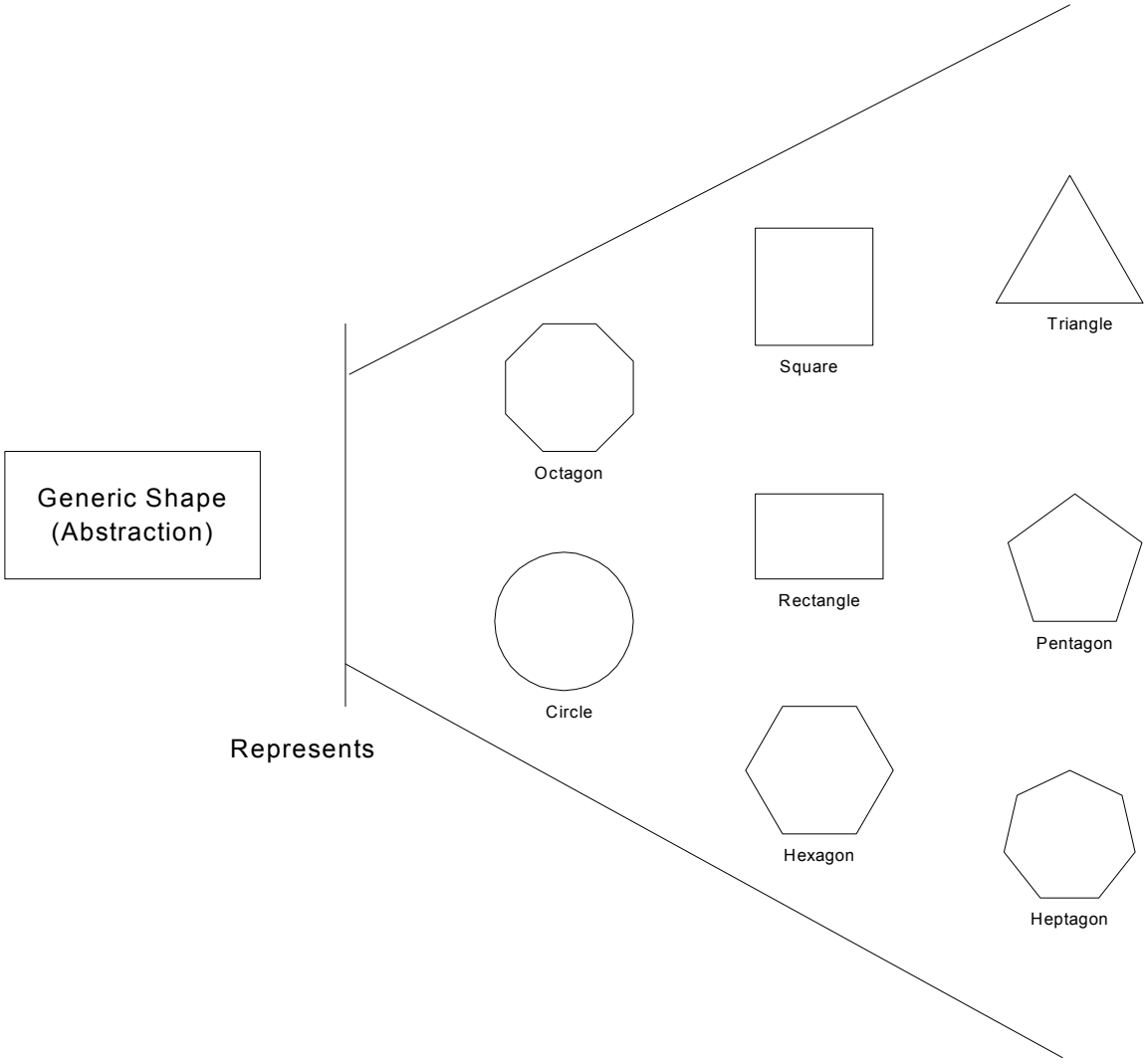


Fig 2.3 Abstraction

In this example of abstraction, we see the generic shape on the left that represents the shapes on the right. Because of Abstraction, we can focus on a subset of the details of the shapes on the right. The subset that we would focus on are those traits that are shared by all of the shapes on the right. An example of such a trait is its perimeter[11]. Another example of such a shared trait is the area of a shape.

[11]The set of shapes on the right-hand side of the diagram should more accurately be called 2-dimensional shapes. This makes the use of perimeter more appropriate.

Fig 2.4 Abstraction

In fig 2.4, we see another example of abstraction. The toy horse on the left is a greatly simplified (not the least of which is that it is inanimate) view of a real horse, as pictured as right. It has some features of the real horse (mane, tail, head, four legs, two eyes, etc. It obviously does not many of the features of real horses, but, at this level of abstraction, provides an adequate representation[12].

[12] It is critical to identify the appropriate level of abstraction. In this case, given the expected audience and use of the rocking-horse, these features it does share, though few, are adequate. Many factors will have to be considered when determining the correct level of abstraction.

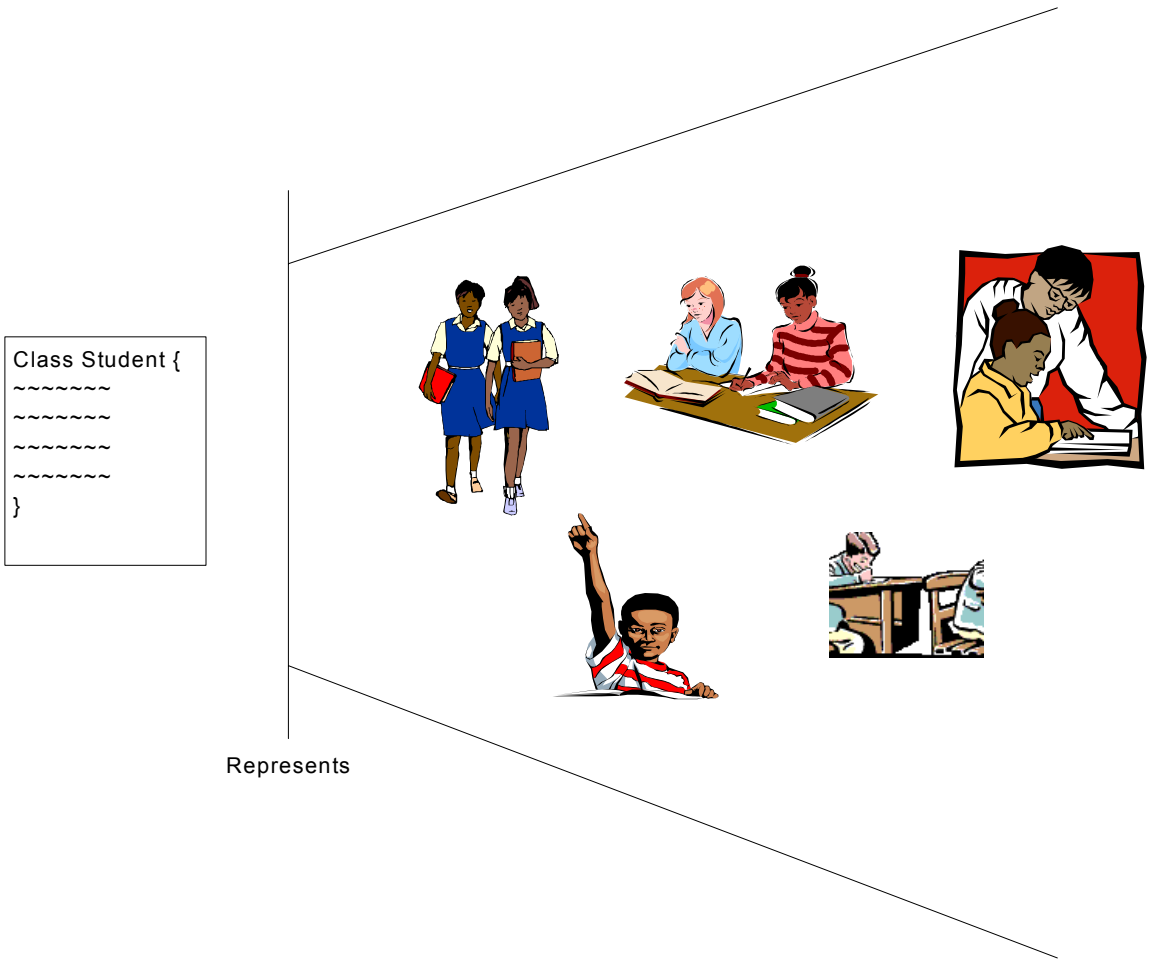


Fig 2.5 Abstraction

In this example, we are highlighting the fact that our abstractions will be captured as class definitions. In these class definitions we include the common data elements and common operations that we need to capture. As before, these have to be at the correct level of abstraction.

Encapsulation

Earlier, we described a class as a grouping of items based on a shared set of characteristics. While accurate, we will add a few things to the definition.

We discussed some of the shortcomings of procedural languages. One of these was that we were unable to restrict access to areas of memory, meaning there could be unauthorized access to memory that we were expecting to use. In addition, we discussed the implementation being open as well, with the possibility of major consequences to minor implementation changes.

In the OO paradigm, classes consist of data and the means to manipulate that data. The class owns both of these items. The data owned by a class may be referred to as its properties or attributes. The means to manipulate this data is via functions, collectively known as methods. The functionality provided by these methods determines the behavior (as above).

In addition, how these methods are implemented, is kept on the "inside" of the class, i.e. hidden from the outside. In addition, the attributes of a class are not necessarily visible from the outside either, so it is more difficult for memory to be overwritten or values changed inadvertently.

We may extend our earlier examples of automobiles as follows:

Let's introduce a class Car. The attributes of class Car may be as follows:
Amount of Fuel
Speed
Etc.

Some of the operation of the Car class might be as follows:
Go Forward
Stop
Go Backward
Engine on
Engine off
Steer
Etc.

Most drivers do not know or care how, or what makes a car go forward, backward, stop, turn, etc. They just want to know that it does, and that it does so consistently, and according to expectations. The interface between humans and cars consists of the steering wheel, pedals, gearshift lever, ignition key, gauges, etc. For our example, it is via this interface that the commands to go forward, etc. will be delivered. The ability to "hide" how the car actually implements the command to go forward, backward, etc. is provided by encapsulation.

Similarly, during the interaction between classes (and objects), there should be no need to access the implementation of a method.

To provide encapsulation, OO languages provide control over the visibility of attributes (data) and methods[13]. We will explore the access levels and what they mean later on.

The view of a class, from the "outside", is termed its interface. This is the set of the publicly accessible operations of the class[14].

Why is Encapsulation Important?

The key concept in Encapsulation is that components (i.e. classes, modules, etc.) all have information that is on the inside and is kept hidden from outside eyes. The areas that are hidden may be those that might be changed more frequently than the external view needs to be aware of.. A data structure or specific calculation routine are examples. Say we have a component that exports[15] an operation `Sort`(). We may want to change the implementation of the `Sort`() routine to take into consideration different variables such as how large a set we're sorting, what elements are to be sorted, i.e. in an array, as a dynamically linked list, etc. Depending on these variables, we could use trivial, off-the-shelf algorithms or specially customized, complex algorithms. The selection of the algorithm is not a detail that needs to be visible to users of the component. In addition, if our `Sort`() operation needed other operations that were only used with `Sort`(), those should also be hidden. With Encapsulation, both are hidden from view. We are able to hide the implementation of operations, as well as complete operations, as necessary. When you hide the implementation, this has a direct impact on reliability, as certain changes are now controlled.

Given the capabilities of Encapsulation, the selection of which operations are exported is a critical activity.

[13] The data is not considered local data even though it is inside an object. Local data (and methods) are defined inside a subroutine or function. In addition, those local elements are only visible inside that subroutine or function. The term "local" is reserved for these items.
[14] Of course, this will also be the set of publicly accessible operations of all objects of this class also.
[15] Makes publicly available.

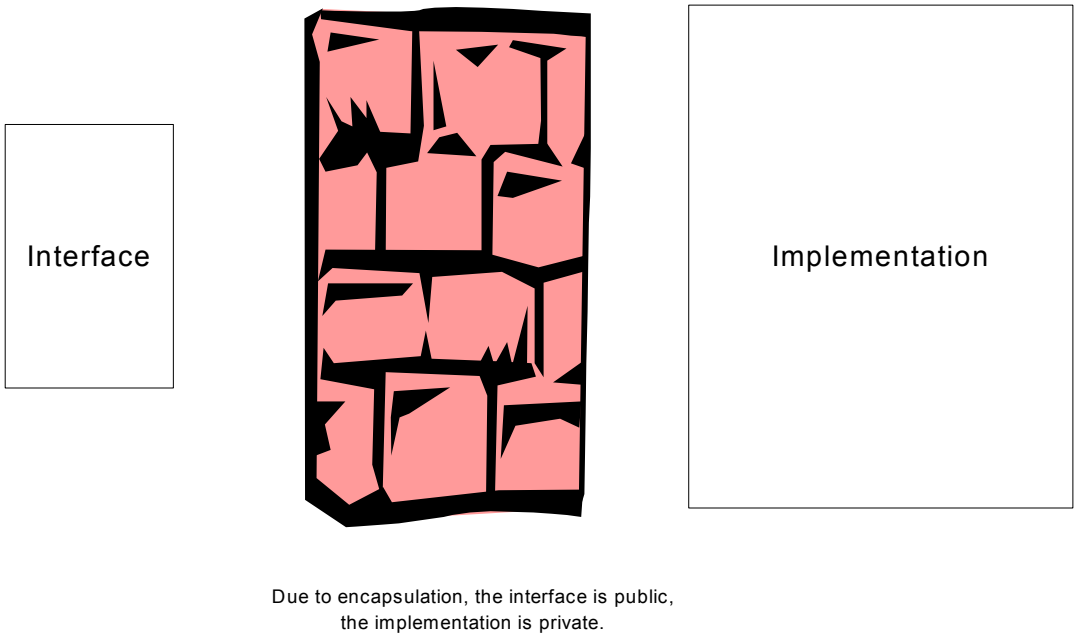


Fig 2.6 Encapsulation

Hierarchical Relationships

In this context, Hierarchy represents a structure that is an ordering of items based on a relationship that is intrinsic to the structure.

Before, we used the class Automobile to describe shared characteristics of a particular class. We also mentioned that we have further specialized examples of automobiles. For example, we have Cars, as a type of automobile, Sedans as a type of car, etc. Each one is successively specialized, but they all share characteristics (features and behavior) of the initial class Automobile. This is an example of inheritance.

Inheritance

There are two types of inheritance – single and multiple. We will concentrate on single inheritance.

Single Inheritance

Think of your family tree and choose either the women or men in your family tree only. Starting with your grandmother or grandfather, your family tree (a portion of it anyway), might look like this:

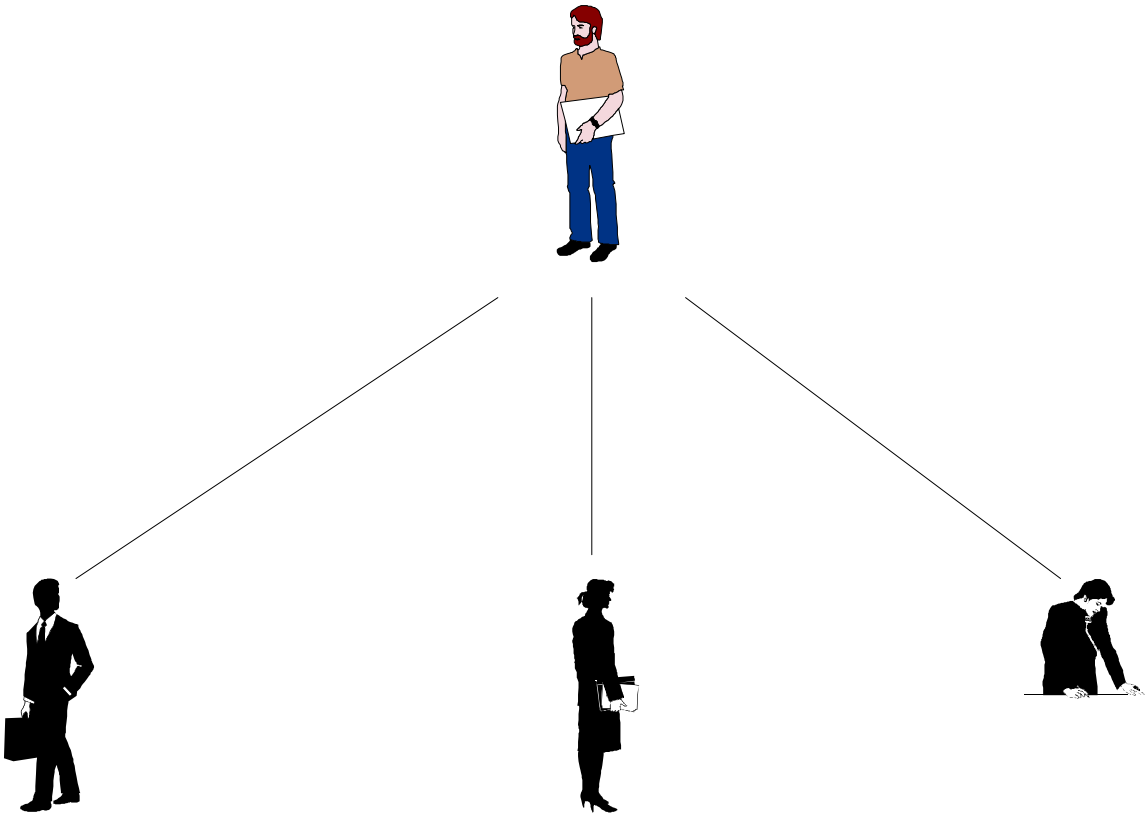


Fig 2.7 Inheritance

You inherited traits, i.e. characteristics, from your grandparents. Of course, each of them inherited traits from each of their ancestors successively.

Let us look at another example – Printers:

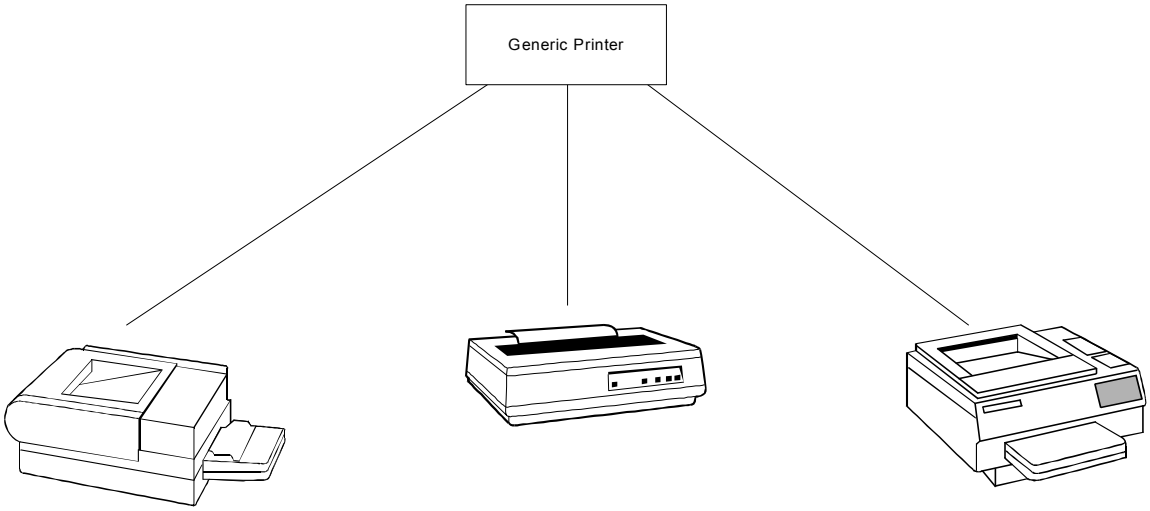


Fig 2.8 Inheritance

Inheritance is an example of a class relationship. Inheritance denotes an "is-a" relationship (unlike composition, to be discussed later). The ability to say something is-a kind of something else implies characteristics and behavior, as before. If we go back to class Human, we can extend the class and construct a class hierarchy as follows:

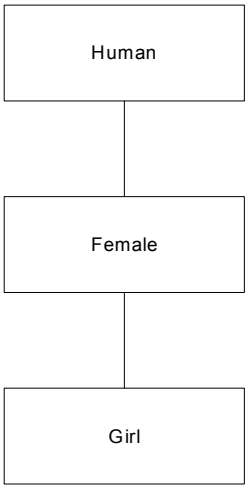


Fig 2.9 Inheritance

We can see from the diagram that Girl inherits from Woman and Woman inherits from Human, etc. As a result, we would expect that Girls would share the characteristics of Women and humans.

The root of our tree, i.e. the first class from which we inherit is called the superclass. Each inherited class is called a subclass.

Multiple Inheritance

The case where we inherit from more than one distinct superclass is termed Multiple Inheritance[16]. If we added back the other half of your family tree, we would accurately see that you inherited traits from your mother and father, not just one (as implied in our example earlier). So, you are a member of (is-a) your mother's side of the family, just as you are a member (is-a) of your father's side of the family.

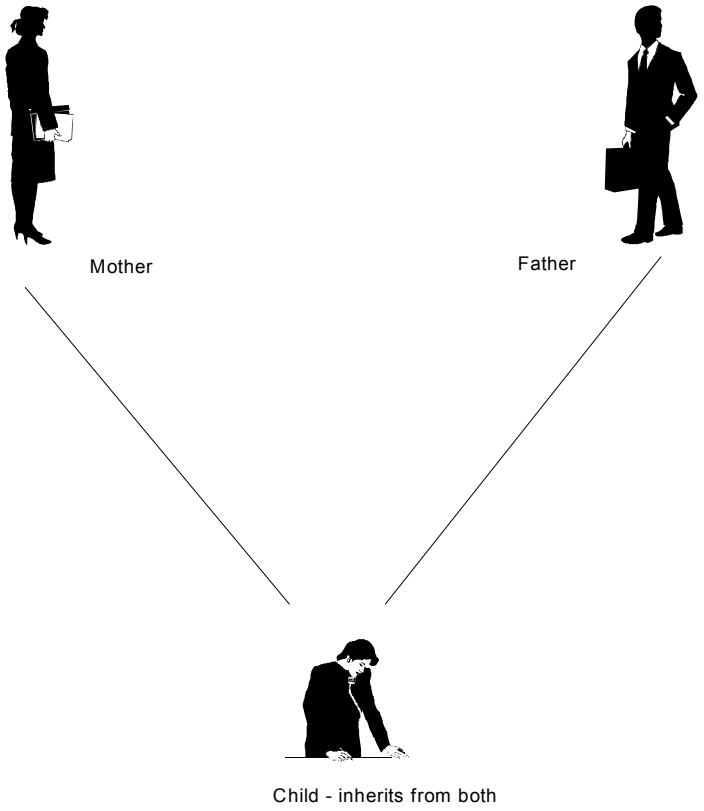


Fig 2.10 Inheritance

Aggregation and Composition

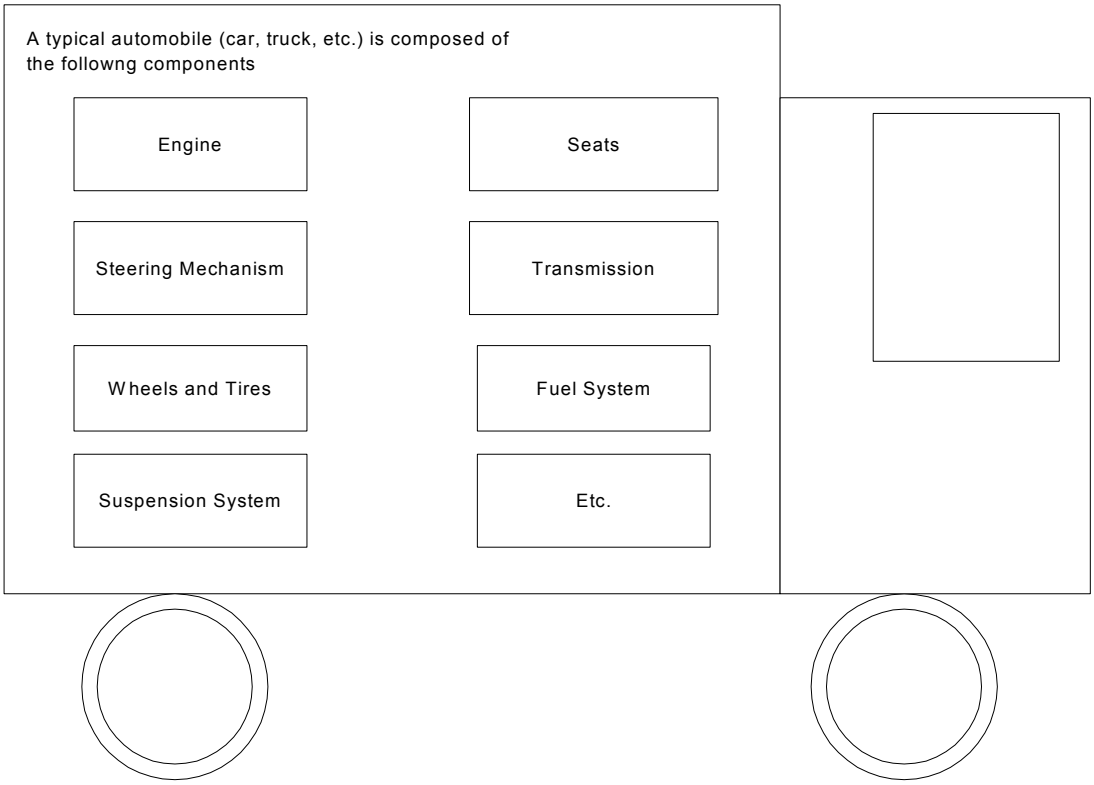
Composition and Aggregation are other examples of hierarchical relationships. Unlike inheritance (is-a), a composition or aggregation relationship is characterized by "has-a". Composition and aggregation express possessive, ownership or containment relationships. This means, a container, such as a paper bag, can be modeled using an aggregation or composition relationship.

[16] It should be mentioned that whereas support for inheritance is a requirement, support for multiple inheritance is not. Indeed, there are object-oriented languages, like Java which do not support multiple inheritance. Further discussions of this are in Chapter 6.

With Aggregation, in order for an object to be valid[17], it is not mandatory to have all of the "parts" present. With Composition, in order for an object to be valid, all of the parts are mandatory. A car is a good example of composition, as follows:

A car has (contains):
An engine
Transmission
Seats
Wheels and Tires
Etc.

In effect, a car is the sum of its parts, and cars have many parts. Obviously, we cannot say an engine is a car, or a transmission is a car, etc. The appropriate way to characterize the relationship between cars, engines, transmissions, etc. is to say a car has an engine, has a transmission, etc. In addition, a "valid" car is one that has all of the requisite parts.



[17] Valid refers to the abstraction. If the semantics of the abstraction do not allow any missing parts, but call for an aggregate, then we would use Composition. If the semantics call for an aggregate, with no rules as to missing parts, we would use Aggregation.

Fig 2.11 Composition

Fig 2.12: A city

A city is an example of aggregation. With aggregation, the whole, i.e. the aggregate, is still valid whether the number of elements varies or not. So, a city of zero, fifty, one hundred or one million residents is still a city.

Why is Hierarchy Important?

The support for Hierarchy has a direct impact on code reuse. This is because we can directly leverage previously developed code, in the development of our system.

Modularity

Modularity is the ability to decompose or partition a system into a set of collaborating components or structures. These structures could be one or more files that contain code. A module is defined as a structure that has data and operations defined on that data. In object-oriented development, a module could be one class, or a group of classes.

Various languages support modularity. In such languages, modules allow the definition of data that is visible throughout the module, but not visible outside the module. In addition, they allow the control of operational visibility also. This means some operations defined within the module will only be visible inside the module, not from outside. All

object-oriented languages (and quite a few non-object-oriented languages) support modularity. As we shall see later on, the notion of a "package" is also centrally related to modularity.

Why is Modularity Important?

Modularity is important for a number of reasons. We can employ modular designs to leverage reusability. For example, we can create modules that represent areas of the program that may be reused. For example, we can create generic sorting modules, for instance. By separating areas of our program's functionality into modules, we can more easily reuse some of that functionality than if we did not use modules. This is true for both object-oriented and structured approaches.

With modularity, we are able to separate areas functionally dissimilar areas of our program, which yields greater clarity, reliability, etc. and associated benefits. For example, we can group functional areas such as user-interface, input/output, data management, etc. into separate modules[18]. While these modules will interact and have many interdependencies, the fact that they are separate will lead to overall system reliability, as we can develop, test and certify and maintain each separately.

For our discussion here, these components correspond to classes and objects[19]. As before, each object is a specific instance of a class. An object is concrete, while a class is abstract. So, even though we would decompose a system into a set of classes, it is the interaction of the objects, instanced from these classes that constitute the operational system. The ability to have an operational system comprised of cooperating objects (at run-time) is due to Modularity.

[18] Depending on the complexity of the system, it may be necessary to have many modules dedicated to each functional area.
[19] As we will see in chapter 8, the term "component" has a specific meaning as well.

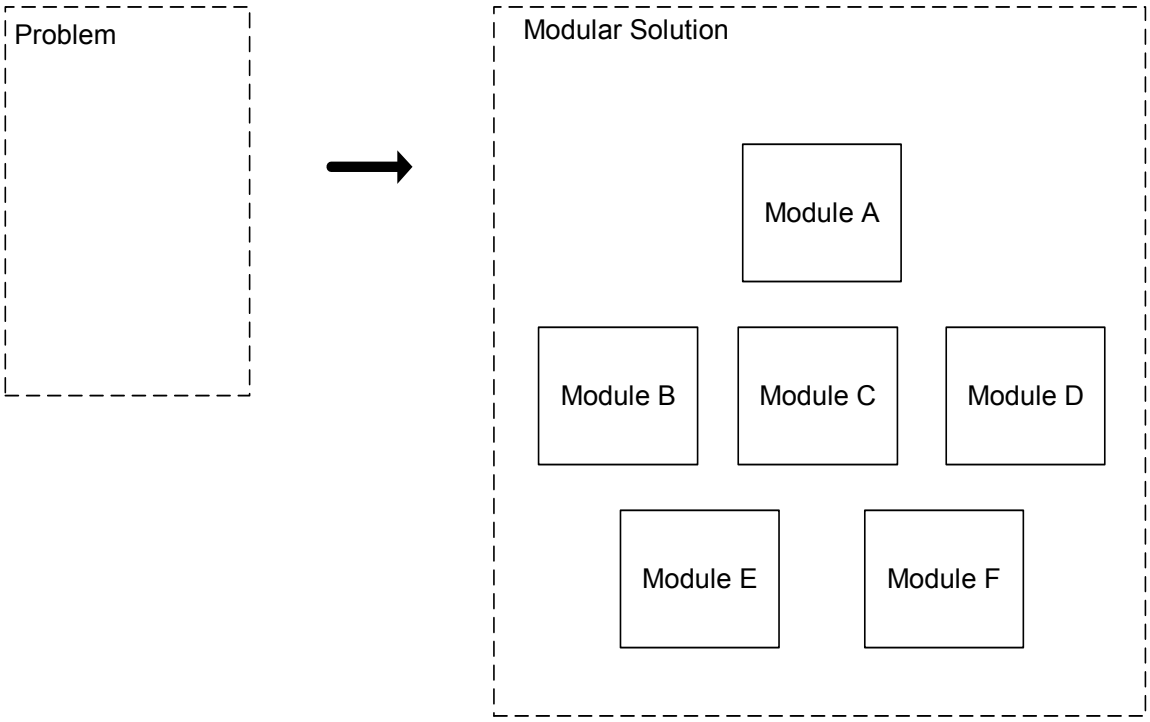


Fig 2.12 Modularity

This diagram illustrates the modular decomposition of a given problem[20].

Persistence

In general terms, it is critical to be able to preserve values between sessions, i.e. from one time of usage to another. In object-oriented development we work with object (at run-time) and we need the ability to "save" the value of our objects from one session to the next. For example, when we create a word-processing document, we save the document, which translates the memory representation of the document into a representation of the document on some other storage device[21][22]. We are most interested in storing the current state of the document. This includes the text of the document, in addition to any cosmetic changes styles we applied, etc. When our word-processing program loads our saved document, we need it to present our document in the exact state it was in when we saved it. Being

[20] The illustration of the modular decomposition does not include links between modules so as not to imply how the modules communicate. A modular system will have cooperating modules, with various interactions and interdependencies.
Whereas the definition of module is general enough to fit both object-oriented and non-object-oriented circumstances, object-oriented modules differ in structure that non-object-oriented modules.
[21] Storage devices may be disk drives, CD-ROM, etc.
[22] Serialization is another term used to describe translating from one form into another. Typically, serialization does not describe how the data is persisted.

able to save and restore our documents state is fundamental to how we use programs such as word-processing programs. Imagine having to complete every document at one sitting because there was no way to save incomplete work! It is worth mentioning that the format of the saved document is outside the scope of the discussion. Whatever the format is, we must be able to take a document saved in that format and do whatever we need to do to display a document ready for editing.

This ability to save and restore state is also fundamental to object-oriented development. It is the state of an object that we are concerned with persisting. The state of the object is dynamic and is based on the effects of various operations on the object that were invoked, up to this point[23]. Object-oriented systems are composed of objects interacting at run-time. If we end then resume a session, we need to have a mechanism that allows us to save the state of our objects at the end of the session and load our saved state at the resumption of our session.

As we will see in Chapter 6, in our discussion of design, we may employ various methods to persist object. The current crop of object-oriented languages provides varying levels of built-in persistence. For those with weaker support, we may employ additional methods to achieve full persistence.

[23] We are limiting the effect on an object's state to the execution of operations, as public data elements in objects are discouraged. We discuss this further in Chapter 6.

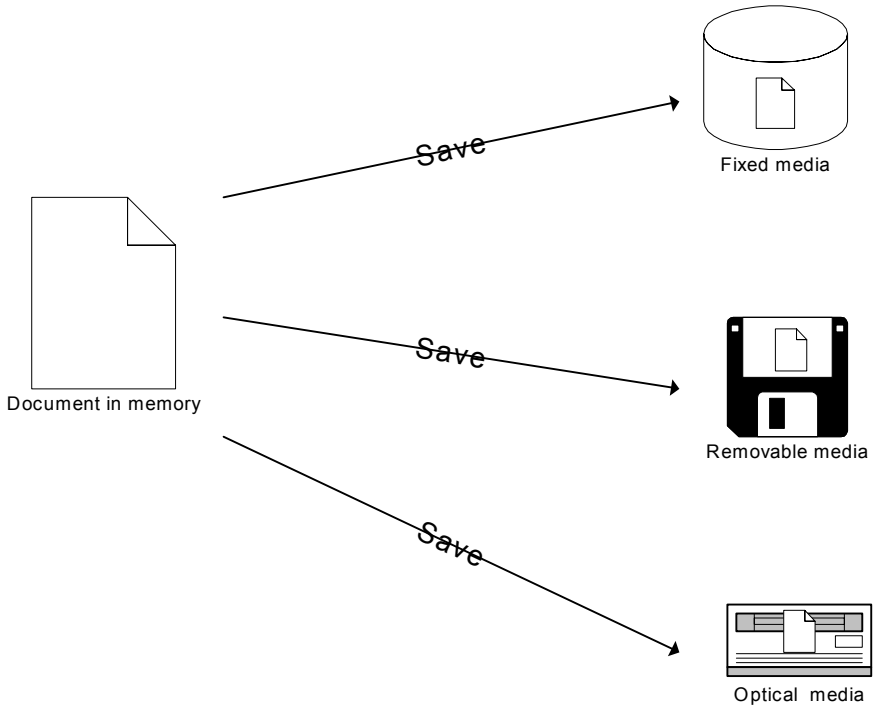


Fig 2.13 Options for saving a document

Benefits of OO Development

Earlier, in the recap of Chapter 1, we discussed some of the drawbacks of procedural languages. The features of the OO paradigm directly address these issues. For example, Encapsulation allows the separation of the use of a procedure (method) and it's implementation. With Encapsulation the designer (of the method) is given a tool to control its use and visibility. Abstraction allows the use of words that reflect the real world that is being modeled. Hierarchy allows the leveraging of relationships in a way that was not possible before. In so doing, it promotes code reuse as well.

From this chapter's discussion, we can expect the following, based on the major features of the OO paradigm:

Abstraction – the ability to create a realistic model, which will make it easier to render solutions, as these models are based on the language (and definitions) of the problem space.

Encapsulation – the ability to identify data and methods, grouped together into one element (class), with the added ability to determine who has access to the data and methods.

Modularity – the ability to model an operational system based on cooperating object, closer to reality than the earlier hierarchical decomposition would allow.

Hierarchy – the ability to take advantage of shared characteristics between groups of classes, adding further specialization (subclasses) as necessary.

Persistence – the ability to preserve the state of objects across sessions

As with most things, the correct application (or usage) of these items will determine how many of the apparent benefits you are able to enjoy.

Interfacing with Non-Object-Oriented Systems

In this chapter, we are seeing a new way of viewing the solution to problems (object-oriented vs. procedural). However, in the real world, we are not always afforded the opportunity to "start over from scratch". In many cases, our solutions involve interfacing with existing systems such as mainframes. In other cases, we need to utilize data that resides in a variety of formats in various databases, files, etc. As we develop our object-oriented toolkit, we will revisit these issues and discuss possible solutions.
The elements of the object-oriented approach give us many alternatives. For example, we could use Abstraction to create classes that represent (i.e. abstract or hide) the legacy systems. With Encapsulation, the details of the legacy system would be hidden "behind" the definition of the class.

Identifying Classes

As listed before in the Analysis phase, we are attempting to discover classes and objects. There are various approaches to doing this. The key is that they are derived from the requirements of the problem domain. Remember, the system we will ultimately design has to satisfy our requirements.

In one example (Shlaer/Mellor), classes and objects usually come from

Tangible things	Cars, telemetry data
Roles	Mother, teacher

Events	Interrupts
Interactions	Loan, meeting

Another perspective (Ross), based on database modeling yields the following:

People	Humans who carry out some function
Places	Areas set aside for people or things
Things	Physical objects
Organizations	
Concepts	
Events	

Here is yet another set of sources for potential objects (Coad/Yourdon):

Structure	Is-a/part-of relationships
Other Systems	External systems with which the application interacts
Devices	
Events remembered	
Roles played	
Locations	
Organizational units	
Subject Areas	Higher level of abstraction: group of classes

Behavior Analysis

Another school of thought uses Behavioral Analysis, which is the focus on primary behavior as source of classes and objects. This is closer to conceptual clustering as we are forming classes based on groups of objects displaying similar behavior. Conceptual Clustering is the approach whereby classes are generated by formulating conceptual descriptions then classified according to the descriptions. A derivative is to use classifications based on behaviors viewed as function points. A function point is one end-user business function. A business function represents some kind of output, inquiry, input, file or interface. Thus, a business function represents any outwardly visible and testable behavior of the system.

Domain Analysis

On a larger scale, we also have Domain Analysis, which is an attempt to identify the objects, operations and relationships that domain experts deem important about the domain (of all applications). This is

applicable to all systems in a domain, as opposed to some of the other approaches, which are more applicable to individual systems.

Domain Analysis seeks to identify the classes and objects that are common to all applications within a given domain, such as securities trading, etc.

Domain analysis helps by pointing you to key abstractions that have proven useful in other related systems.

Some of the steps of Domain Analysis are as follows:

- Construct models based on consulting with domain experts[24]
- Examine existing related systems and represent using common format
- Identify similarities and differences between systems by consulting with domain experts
- Refine strawman[25] to accomodate existing systems
- DA may be applied vertically: across similar applications or horizontally: related parts of same application.

Use-Case Analysis

The previous practices depend heavily on the experience of the analyst (domain expertise). With use-case analysis, this shifts to the experience of the end-user. This practice may be coupled with our earlier approaches as well.

As mentioned above, a use-case[26] is a particular scenario that begins with some user of the system initiating some transaction or sequence of interrelated events.

Users/domain experts/development team members enumerate scenarios fundamental to the system's operation. These will collectively describe the system functions of the application (part of the requirements gathering phase).

Analysis then proceeds by studying each scenario, outlining the objects that participate in each scenario, each object's responsibilities

[24] Note: "Domain expert": experienced user.
[25] A "strawman" is a model, prepared with the understanding that all details are possibly unavailable at the time of its creation.
[26] Use cases are discussed in more detail in Appendix 1.

and how objects collaborate with each other. A clear separation of concerns among all abstractions is crafted.

The scenarios captured in the use cases may also be used as the basis of system tests.

Informal English

In this practice, we write an English description of the problem then underline all nouns and verbs (Abbott). The nouns represent candidate objects and the verbs represent candidate operations on them. This is useful, because it forces the developer to work in the vocabulary of the problem domain. However, it is not rigorous, because of the ambiguities in and impreciseness of the English language (verbs may be derived from nouns and vice versa).

Structured Analysis

Many of us are familiar with the process (and results) of structured analysis. Many tools and methods support structured analysis. The idea with starting with structured analysis is to reuse such artifacts by creating a "bridge" to the object-oriented way of thinking.

Candidate objects may be derived from the following:
External entities
Data stores
Control stores
Control transformations

Candidate classes may be derived from:
Data flows
Control flows

Data transformations we assign as either operations on existing objects, or as the behavior or "agent" objects.

Finding Key Abstractions

Key abstractions are very important to our overall process. A key abstraction is an abstraction that is to be included in our models and subsequent design. There are potentially many candidate classes that may be reviewed. Not all candidate abstraction s will necessarily become classes. As we examine each one, we will keep those that

have merit, i.e. represent the real-world objects we want to model. These key abstractions are the abstractions that will become classes.

The primary value of a key abstraction is that it gives boundaries to our problem - highlight things in our system relevant to our design. The identification of key abstractions allows us to be specific about behaviors, hence the notion of boundaries. Key abstractions will always be domain specific, i.e. specific to the domain of the problem.

The identification of key abstractions uses two mechanisms: discovery, i.e. obtaining the abstraction from the requirements, and invention, i.e. adding abstractions that were not explicit in the requirements. We may recognize key abstractions through interactions with domain experts and review of the requirements. If they talk about it, then it may be an important abstraction. Through invention, we may add abstractions that are not part of the problem domain, but should be part of the solution. Many of the abstractions identified out of invention are useful in design or implementation.

One of the most powerful ways of identifying key abstractions is to look at the problem and see if there are any abstractions that are similar to the classes and objects that already exist.

CRC Cards

CRC cards are a simple, yet effective way of helping to understand how a particular candidate class would fit into your overall picture. CRC means Class/Responsibilities/Collaborators.

A CRC card is an index card (typically 3inches by 5 inches). With the name of the class written at the top of the card, on one side, list the responsibilities of the class. On the other, list all of the other classes that are collaborators with this class.

Sample Project

Here is our "problem". We will work through the following example.

Requirements

Design a program to manage student information based on the following criteria:

There are three types of students: **Typical**, **Faculty** and **Transfer**. Typical students are regular college students, about which the following information is typically captured – Name, Address, ID and Major.
Some students may also be faculty of the college. In this case, what subject they teach is captured. Members of faculty are also given a discount based on their years of service, 10% for up to 5 years, 20% for 6-10 years, 30% for more than 10 years. Only faculty members of this college are eligible.
In addition, some students may be temporary transfers from another college. In this case, we need to know their home college and it's address to be able to return their grades at the end of the semester.

All students over the age of 55 are eligible for a 15% discount (in addition to other discounts if possible).

Students may be full-time or part-time. Full-time students are those with 10 credits or more. Part-time students are those with less than 10 credits.

The current classes are as follows:

- Abstract Algebra 4 Credits
- Calculus 4 Credits
- Intro to Computing 4 Credits
- Advanced Computing 4 Credits
- Object Oriented Programming Using C++ 4 Credits
- English 3 Credits
- Spanish 3 Credits
- Chemistry 3 Credits
- Physical Education 1 Credit
- Art History 2 Credits

The college charges $100 per credit per semester.

The current majors are as follows:

- Math
- Computing
- English
- Chemistry
- Undeclared

Periodically, the college will add new classes and majors to these lists. Students may have at most two majors.

The system must be capable of the following:

- Adding a new student's information
- Searching and displaying a student's information
- Deleting a student
- Changing/assigning classes and credits to students
- Changing/assigning a student's major
- Changing/assigning a student's type
- Changing/assigning a student's status, i.e. full-time or part-time according to the rules above
- Producing reports as follows:
  Sorted list of full-time students (all information)
  Sorted list of part-time students (all information)
  Number of students of each type (typical, faculty and transfer)
  For each type of student, a sorted list of student names and addresses
  For each type of student, a reversed list of student names and addresses
  List of all students, their majors and number of credits
  A sorted list of all students based on their cost for the semester

Notes:

- The system will maintain student data in a relational database
- The system should allow the entry of complete information at time of addition
- Implement a simple, straightforward user interface[27]
- Design for "real-world" use - employ error handling where appropriate

[27] We will discuss user interface objects in Appendix 2

Chapter Summary

- A class is a group of items based on a set of shared and similar characteristics that are exhibited by all members of the group.
- Each class has a set of semantics or rules that govern its behavior.
- An object is an instance of a class.
- Objects have identity, behavior and state.
- Abstractions allow us to focus on the relevant characteristics of the real world object it represents. It is a simplification.
- Encapsulation allows us to separate the interface from the implementation.
- Hierarchy allows us to leverage the similarities that exist between classes and enables use of previously developed code.
- Modularity allows us to partition a system into a set of collaborating components.
- Persistence allows us to preserve the values of objects.
- We may use various techniques to identify classes such as Behaviour Analysis or Domain Analysis.

Exercises

1. Create an example demonstrating each of the following: inheritance, composition, aggregation and association.

Chapter 3

Class Structure

We have now developed a basic understanding of the OO paradigm, i.e. way of thinking. We will now go forward and delve more deeply into the details of classes and objects, by continuing our discussion of class structure and interactions.

The definition of a class can be abbreviated as follows:

> *A class is a structure that contains data and methods that manipulate that data.*

Data, in this definition, represents the data contained in a class, what we've also referred to as values, attributes or properties or fields. Methods are the functions (or procedures) that are defined within the class as members of the class (also referred to as member functions). Methods operate on the data defined in the class. Methods are the only functions that directly operate on the data in the class without explicit permission.

"Design-time" and "run-time" Defined

For us, "Design-time" refers to the activities during the Design phase. In the Design phase, some new classes may be defined, while those existing (as of the Analysis) are refined as necessary. Class relationships are identified and exploited as appropriate. At "run-time", we have objects collaborating to provide the functionality of the system.

What is Class Structure?

Let us define a class that represents a general shape. Class `Shape` would then be a generalization for all shapes, such as circles, squares, trapezoids, triangles, hexagons, etc. Based on the definition above, as represented in pseudo-code, we can construct a very general class definition for `Shape` as follows[28]:

In C#:

```
abstract class Shape
{
/* Data */

     double    area;


/* Methods */
     public abstract double CalculateArea()[29];
               /* no implementation */
}
```

The attribute *area* is defined as a number. For our purposes, number could be either integer or real (including double and long, depending on the size of our shapes [1]).

We've also defined a method `CalculateArea()`, which for our purposes, returns a value (of type integer), which represents the calculated area of our shape.

As we said before, we intend to let class `Shape` be a generalization of all shapes.

Let's also see what this class definition would look like in two popular object-oriented languages:

In C++:

```
class Shape{
    int area;
    public virtual int CalculateArea()=0;  /* function
                    prototype only – no implementation */
```

[28] The examples are in pseudocode, C++, C# and/or Java.
[29] public, as used here, will be defined later

```
}
```

In Java:

```
abstract class Shape{
     int area;
     public abstract int CalculateArea();
     /* no implementation */

}
```

Let's go further and define two additional classes, *circle* and *rectangle*. As shapes, a circle and a rectangle share properties in common, as we can say a circle is a shape and a rectangle is a shape also. We can rewrite this, using language from last class as follows:

- A circle *is-a* shape
- A rectangle *is-a* shape

This use of *is-a* is a depiction of a particular type of relationship, Inheritance, which was discussed as part of Hierarchy (one of the main elements of the OO paradigm).
As class designers, we want to exploit these classifications where beneficial. So we want to exploit the similarities between circles and shapes, and between rectangles and shapes - definitely candidates for inheritance.

Let's define class *circle* as follows:

In C#:

```
class Circle : Shape[30]
{
/* data
     const double PI = 3.14159;  /* rounded */
     double radius;

/* methods */
     double CalculateArea()
     {
     }
}
```

[30] In C#, the colon ":" in the class definition, as used here, indicates inheritance, with the immediate superclass appearing to the right of the colon, the subclass to the left.

```
class Rectangle : Shape
{
/* data */

     double length;
     double width;

/* methods */
     double CalculateArea()
     {
     }

}
```

Since we're using inheritance, we do not need to redefine area – we inherit it from our *superclass* Shape. A superclass is a direct ancestor of a class. So in this example, *Shape* is the superclass for both circle and rectangle. We refer to circle and rectangle as subclasses. This means that circle and rectangle are specializations of shape. With circle and rectangle, we're no longer referring to all generic shapes – we're now referring only to shapes that conform to the definition of circle and rectangle.

For class Circle, we've added a new field called *radius*. The value of this field will be needed for us to calculate the area of a circle.

For class Rectangle, we've added two new fields – *length* and *width*. These are necessary for the calculation of the area of a rectangle and are self-explanatory.

Note – with these changes, even though circles and rectangles are both shapes, we can see that circles and rectangles are quite different.

Let's discuss what we need to do with the superclass method `CalculateArea()`. As you've seen from above, we have redefined `CalculateArea()` in both of our subclasses.

The reason for this becomes clearer when we consider exactly how we calculate the area of circles and the area of rectangles. They are not the same.

For circles, area = PI * $(\text{radius})^2$

For rectangles, area = (length * width)

Obviously, any implementation of `CalculateArea()` other than these for Circle and Rectangle would be incorrect.

So, we need to implement the specific steps in `CalculateArea()` for Circle and Rectangle.

Implementing specific functionality in a subclass method, where the method name is the same as in the superclass is known as ***method overriding***. This means we are able to implement, in each derived subclass, the appropriate steps for calculating the area of that specific shape.

Here are important questions for us to answer:

a) How would we implement `CalculateArea()` for the generic *shape* class?
b) Why would we implement `CalculateArea()` for class *Shape*?

In the context of class Shape, i.e. what we understand via the generic description of the class, Shape represents generic shapes, no specific shapes. In order to calculate the area of a particular shape (note the word particular), we need to know the specific shape for which we need to calculate the area. This means an implementation of `CalculateArea()` in the superclass Shape has no meaning in this context. So the answer to question b) is: we wouldn't implement `CalculateArea()` in the superclass at all.

So why include `CalculateArea()` in the superclass at all? Well, all shapes have an area, with the way to calculate that area being specific to that particular shape. So, as designers of the class, we wanted to make a provision in the superclass for the calculation of a shape's area. We want to make sure that all classes that inherit from shape implement a method that calculates that shape's area with a method named `CalculateArea()`.

Abstract Classes

Let's back up a moment. We just said that there was no implementation of `CalculateArea()` in the superclass, because such an implementation, in this context, would make no sense. However, we have declared the method `CalculateArea()` in the superclass. As a result of this, class Shape is considered an ***abstract*** class. An abstract class, such as class Shape, is one in which there is no implementation for at least one declared method. Methods for which

there is no implementation are also considered abstract (pure virtual in C++; defined as abstract in Java and C#). In Java and C#, the declaration of abstract methods requires adding the word "abstract" to the method declaration.

Abstract classes are very interesting to work with. Like all classes, they serve as a blueprint (from our earlier discussion). However, unlike other classes, they cannot be instanced, i.e. objects of abstract classes cannot be instantiated. So, in our example, we would not be able to create any generic shape objects. In addition, any class that inherits from an abstract class has to provide an implementation for its abstract methods or it will also be considered abstract.

So to recap, shape is our superclass or base-class. It is the root of our class-hierarchy. Class circle and class rectangle are subclasses of class shape. The relationship is characterized by is-a:

Circle is-a shape
Rectange is-a shape.

However, bear in mind that though circle and rectangle both inherit from shape, the following relationship is not valid, i.e.:

Circle is-a rectangle (wrong!)

This is obviously incorrect.

So each derived shape class knows how to calculate it's own area. This ability is important to object oriented thinking.

Let's examine another subclass *square*. Class square is a specialization of class rectangle. A square is a rectangle with the length and width being equal. This example is a bit simplistic, but it will serve to illustrate a point regarding overriding methods.

Class Square (inherits rectangle) is described as follows:

```
class Square : Rectangle
{
/* data */

/* methods */
}
```

We have declared neither data nor methods. This is because, in our simplistic example, the data and methods declared for rectangle are appropriate for square as well. The idea here is this: though we are able to override the `CalculateArea()` method of class Rectangle and we are able to add new fields to class Square, we will do so when it's appropriate. In this example, the way we calculate the area of a rectangle is the same as the way we calculate the area of a square. In both cases we use (length * width). So we can use the data and methods, as defined in the superclass (in this case class Rectangle) and they are still appropriate. Compare this to the earlier situation regarding the implementation of `CalculateArea()` in class shape. In this case, we do not need to override `CalculateArea()` for class Square.

Class and Object Interactions

As we discussed earlier, objects are instances of classes. How do we represent objects in OO languages (or in our pseudo-code)? In reality, objects in a programming language are declared in the same way as variables, with the difference being the type (in this case class is synonymous with type). So let's assume we have declared the following variables (objects):

```
Shape objShape;              /* illegal!  Remember - cannot
                              create instance of
                              shape because Shape is
                              abstract */

Circle objCircle;

Rectangle objRectangle;

Square objSquare;
```

We can draw parallels between classes and types and objects and variables as follows:

Each new class we define introduces a new data *type*. A class is an example of a user-defined type. A class defines a kind or type of objects. In fact, the values of a class are objects. What does this mean? Let's review primitive types for a moment. Let's use a primitive type as an example. Examples of primitive types are types such as integers and characters (int, char, etc. in C++/Java). Let's use the character type for our example. The character type represents

all Unicode or ASCII characters (depending on language). However, a particular character value represents one character.

For example:

```
char c = 'A';
```

In Java, C# or C++, this represents one member of the set of all characters, i.e. only the character 'A'.

Similarly, an object represents one example (or instance) of all objects represented by a particular class definition.

How Classes Determine the Behavior of Objects

As mentioned above, objects are instances of classes. Another perspective is that a particular object (instance) is "one-of" the set of all objects defined by the class. This is an important point. The definition of a class provides the blueprint for an object. An object cannot have data or methods not provided for (i.e. defined) in the class of which it is an instance.

The behavior of an object is based on the functionality of its methods. Of course, the methods are defined in the class definition. So, the class, which is the blueprint for the object, determines, based on this blueprint, what the behavior of the class will be.

Introduction to Class Modeling using UML[31]

Some important events occurred in the software industry in the mid 1990's. One of the more important among them was the unification of the work by Booch, Rumbaugh and Jacobson, now known as the "Three Amigos". Booch, Rumbaugh and Jacobson are among the pioneers of object methodology. Each was pursuing different areas separately. Their "coming together" has yielded the modeling language now known as UML, the Unified Modeling Language. In addition to UML, there is also a complete methodology that has been created for software developments, which uses UML to depict its artifacts.

[31] Appendix 2 is a brief UML reference

UML is a general purpose modeling language designed to specify and document the products (i.e. artifacts) of software systems. With UML, we are able to visually describe the structure and behavior of object-oriented systems.

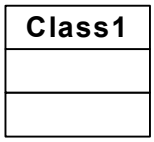
History of UML

As mentioned earlier, UML is a unification of previous modeling methods. These earlier methods included Booch, OOSE (Object-Oriented Software Engineering – Jacobsen) and OMT (Object Modeling Technique – Rumbaugh). Development of UML began in 1994 when Grady Booch and Jim Rumbaugh of Rational Software began unifying the Booch and OMT methods. In the fall of 1995, Ivar Jacobsen joined the unification efforts and merged in OOSE. Further inputs from several other companies were accepted and UML 1.1 was submitted to the OMG (Object Management Group) for adoption in 1997.

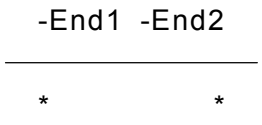
UML Notation

Using UML, we can represent classes, objects and their respective relationships. This list of icons is certainly not exhaustive, but it introduces us the icons used to represent classes and objects, such as those we've designed so far. As we progress through the chapter, we will add more icons and model more complicated structures.

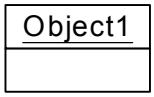
Classes and Objects in UML



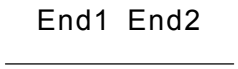
This icon represents a generalized class in UML. The class name is first and is bold and is above the first horizontal line. Attributes and operations may be added above each of the following horizontal lines,respectively



This horizontal line represents a link (i.e. an association) between two classes (note: in practice, line not necessarily horizontal)

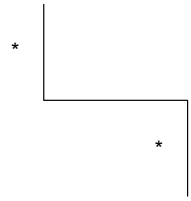


This icon represents an object, with the object's name underlined and above the horizontal line

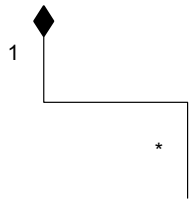


This line represents a link between objects (note: line not necessarily only horizontal)

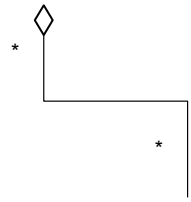
Fig 3.1 UML Notation



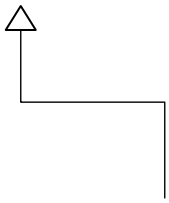
Depicts binary association, i.e. between two classes.



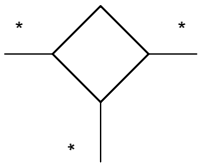
Depicts a composition relationship (filled diamond)



Depicts an aggregation relationship (clear diamond)



Depicts an inheritance relationship. The arrow points from the subclass to the superclass



Depicts an N-ary association, i.e. an associative relationship between many classes.

Fig 3.2 UML Notation

Let us revisit our shape example from before using UML Notation:

Class Diagram in UML

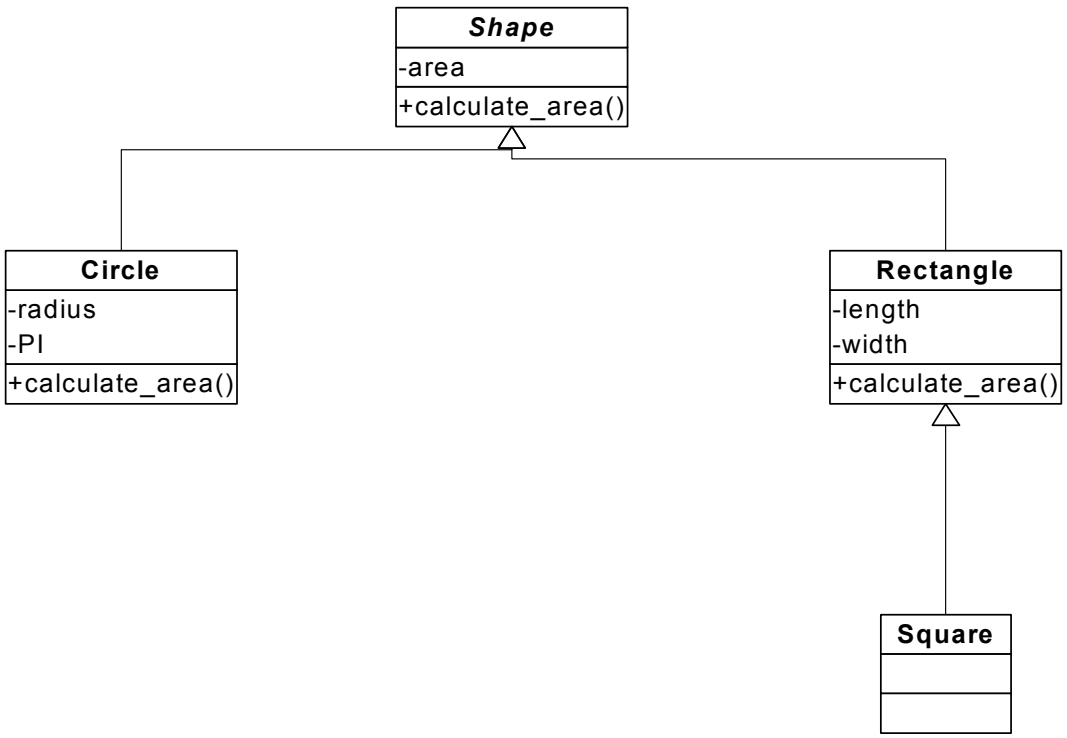


Fig 3.3 UML Class Diagram

In each case, the arrows indicate the direction of the inheritance relationship. In addition, the diagram includes the character 'A' in the inverted triangle (in the Shape class icon) to communicate that Shape in an abstract class.

Benefits of Class Modeling

The fundamental reasons to use modeling notation is for analysis and communication. Modeling is the act of creating a visual representation of the structure and behavior of a system. It allows (at a certain level of abstraction) the communication of certain concepts more clearly (and easily) than the alternatives. The spoken (natural) language, regardless of which one, is imprecise, and thus sometimes is not as useful when it comes to more complex concepts. At the other end of the spectrum is code. Code is very precise – it represents the very detailed instructions you're giving the machine to execute, via

translation or directly. While being very precise, it is also too detailed. So models are used when a certain level of precision is required, or needs to be conveyed, but when you also do not want to be lost in all of the details. Another way of looking at it is that modeling allows someone to obtain an overall view of the system, one that may be comprehensive and detailed, but which is not bogged down by the specific implementation details particular to each language and environment.

Modeling Activities

As above, we will use modeling notation for communication. We will need to communicate various aspects of our system, during each phase of development. We will begin with class diagrams for communicating class structure. As we progress, we will employ other UML diagrams to describe various perspectives of object behavior.

Sample Project

Analysis

Identifying the abstractions (classes) is the key to solving this problem. This activity will be iterative and we'll make small steps of progress, as discussed earlier.

Let's start with the informal English method. Let's identify possible abstractions by underlining nouns, as follows:

> *Design a program to manage student information based on the following criteria:*
>
> *There are three types of students: Typical, Faculty and Transfer.*
> *Typical students are regular college students, about which the following information is typically captured – Name, Address, ID and Major.*
> *Some students may also be faculty of the college. In this case, what subject they teach is captured. Members of faculty are also given a discount based on their years of service, 10% for up to 5 years, 20% for 6-10 years, 30% for more than 10 years. Only faculty members of this college are eligible.*
> *In addition, some students may be temporary transfers from another college. In this case, we need to know their*

home college and its address to be able to return their grades at the end of the semester.

All students over the age of 55 are eligible for a 15% discount (in addition to other discounts if possible).

Students may be full-time or part-time. Full-time students are those with 10 credits or more. Part-time students are those with less than 10 credits.

The current classes are as follows:

- *Abstract Algebra 4 Credits*
- *Calculus 4 Credits*
- *Intro to Computing 4 Credits*
- *Advanced Computing 4 Credits*
- *Object Oriented Programming Using C++ 4 Credits*
- *English 3 Credits*
- *Spanish 3 Credits*
- *Chemistry 3 Credits*
- *Physical Education 1 Credit*
- *Art History 2 Credits*

The college charges $100 per credit per semester.

The current majors are as follows:

- *Math*
- *Computing*
- *English*
- *Chemistry*
- *Undeclared*

Periodically, the college will add new classes and majors to these lists. Students may have at most two majors.

From this exercise, we have the following nouns:

Program
Student
Information
Typical
Faculty
Transfer
College
Name

Address
Id
Major
Faculty
Subject
Discount
Semester

Obviously, there may be additional potential classes for us to use if we use other methods.

How do we determine whether or not these potential or candidate classes make sense in the context of the problem to be solved?

Let's restate the problem. What we are asked to do is design a system to "manage" a group of three types of students. Let's see what the requirements said:

Your system must be capable of the following:

- *Adding a new student's information*
- *Searching and displaying a student's information*
- *Deleting a student*
- *Changing/assigning classes and credits to students*
- *Changing/assigning a student's major*
- *Changing/assigning a student's type*
- *Changing/assigning a student's status, i.e. full-time or part-time according to the rules above*
- *Producing reports as follows:*
  Sorted list of full-time students (all information)
  Sorted list of part-time students (all information)
  Number of students of each type (typical, faculty and transfer)
  For each type of student, a sorted list of student names and addresses
  For each type of student, a reversed list of student names and addresses
  List of all students, their majors and number of credits
  A sorted list of all students based on their cost for the semester

Notes:

- *The information in the system is for the current semester only – no history*
- *The system should allow the entry of complete information at time of addition*

So, by "managing" our group of three types of students, our system must be capable of providing the required functionality listed above.

In reviewing our list of candidate classes (our nouns), we don't have a candidate class "System". Let's add one, giving us the following:

Program
Student
Information
Typical
Faculty
Transfer
College
Name
Address
Id
Major
Faculty
Subject
Discount
Semester
System

Given our understanding of the problem, let's investigate each abstraction's semantics by employing CRC cards, or their equivalent. So, imagine you have 3x5 cards, on each of which you have written one of the nouns extracted earlier. On one side we will put the class' responsibility and on the other side, we will put the class' collaborators.

Let's view each of these separately:

Program

Responsibilities

Hmmm. Something called "program" seems to refer to what we're trying to design. So the responsibilities of "program " would be to provide the functionality per the requirements.

Collaborators

With the responsibilities of Program define as above, it would seem that it would interact with many or all of the classes included in the design. But, as we have not investigated the other classes, let's defer this until later on.

Student

Responsibilities

This class represents all students in our college. This seems to be central to our design. What are the responsibilities? A student is responsible having zero, one or two majors, a selection of one or more classes, etc.

Collaborators

From the information we've gathered so far, the Program class, in addition to majors and classes would be collaborators.

Information

Responsibilities

What would this class represent? That is not clear. In re-reading the requirements, it seems we should have put "information" with "student".

Collaborators

None.

Typical

Responsibilities

This class would represent one group of students. Upon further review, it seems the responsibilities of this class are very similar to those of the student class outlined above.

Collaborators

Same as Student.

Faculty

Responsibilities

This class also represents one group of students. The responsibilities of this class are also very similar to those of the student class outlined above, with the exception of the following: In representing students that are also faculty, this class must also keep track of what subjects are taught as well

Collaborators

Same as Student.

Transfer

Responsibilities

This class also represents one group of students. The responsibilities of this class are also very similar to those of the student class outlined above, with the exception of the following: In representing students that are also transfers, this class must also keep track (via college name and address) of which college these students have come from.

Collaborators

Same as Student.

College
Responsibilities
In the context of the problem, this class would represent colleges. Thus, this class would be responsible for keeping names and address and any other information relevant to colleges, given in the problem.
Collaborators
Transfer students

Name
Responsibilities
This would represent a student's name. Not enough information was given in the problem to determine how a name should be represented. We will have to make an assumption, as follows: first name, last name, middle initial and title (Mr., Mrs., etc.)
Collaborators
This class would collaborate with all classes that need names, such as the student, typical, transfer and faculty student classes.

Address
Responsibilities
This would represent an address. Enough information was not given in the problem to determine how an address should be represented. We will have to make an assumption, as follows: street, city, state and zip.
Collaborators
All classes requiring an address, i.e. student, typical, transfer and faculty student classes, in addition to the college class.

ID
Responsibilities
This class would represent a student's ID.
Collaborators
Student, typical, transfer and faculty student classes.

Major
Responsibilities
This would represent a student's major. Enough information was not given in the problem to determine how a major should be abstracted. So we may assume that a major would only have a name.
Collaborators
Student, typical, transfer and faculty student classes.

Faculty

Responsibilities

In the context of the problem, this class would represent all members of faculty of a college. Not enough information is available.

Collaborators

The College class.

Subject

Responsibilities

This class would represent all subjects available to students. Based on the requirements, this class would contain the subject's name and number of credits.

Collaborators

Collaborators could include Student, Typical, Transfer and Faculty "student" classes, in addition to the Program class.

Discount

Responsibilities

This class would represent all discounts available to students.

Collaborators

Collaborators could include Student, Typical, Transfer and Faculty "student" classes.

Semester

Responsibilities

This class would represent all semesters.

Collaborators

Collaborators could include Student, Typical, Transfer and Faculty "student" classes, the College class and the Program class.

System

Responsibilities

This class would be responsible for providing the functionality outlined in the requirements. In addition, something has to get the ball rolling. The system class would be responsible for this as well.

Collaborators

Collaborators could include Student, Typical, Transfer and Faculty "student" classes, etc.

Chapter Summary

- A class provides the blueprint for an object, as an object is an instance of a class.
- UML is a general purpose modeling language designed to specify and document the products of software systems.
- Class modeling helps us in analysis and allows us to communicate better.

Exercises

1. Create class diagrams for each of the examples of inheritance, composition, aggregation and association done for the previous sessions' assignment.

Chapter 4

Class Relationships and Interactions

Class Hierarchies

Last chapter, we discussed the structure of classes. We added a new definition of a class as follows:

> *A class is a structure that contains data and methods that manipulate that data.*

Based on this definition, we looked at how we would implement sample classes and extended our discussion to include *inheritance*. To review, the inheritance relationship is based on similarities between supertypes (i.e. superclasses) and subtypes (i.e. subclasses). These similarities are described by "is-a".

Examples (from last time):

> A circle is-a shape
> A rectangle is-a shape

Subclasses inherit methods and data from superclasses (i.e. parent classes or *base* classes).

Taken as a whole, this is an example of a *class hierarchy*. A class hierarchy represents relationships between classes. The hierarchy may be an inheritance hierarchy (as in our previous example), but there are other hierarchies as well, such as Composition, Aggregation and Association. Each is discussed below.

Inheritance and Polymorphism

Inheritance defined

Inheritance is the relationship (as in our previous example) where one class is the superclass and the other(s) inherit from, or extend the functionality defined in that class. Again, this relationship is characterized by "is-a", also as above.

Polymorphism defined

Polymorphism literally means "many forms". The relevance of this becomes clearer below.

In the last chapter, we discussed abstract classes. A class is described as abstract if it contains at least one abstract method. An abstract method does not have an implementation in the class in which it is declared. A subclass that inherits from an abstract base class, but does not provide implementations for the abstract methods is itself an abstract class.

In our example from last chapter, we had the following:

```
abstract class Shape
{
/* Data */

     double    area;


/* Methods */
     public abstract double   CalculateArea();

}
```

```
class Circle : Shape
{
/* data
     const double PI = 3.14159;  /* rounded */
     double radius;

/* methods */
     public double CalculateArea()
     {
          //implementation omitted
     }
}

class Rectangle : Shape
{
/* data */

     double length;
     double width;

/* methods */
     public double CalculateArea()
     {
          //implementation omitted
     }
}

Class Square : Rectangle
{
/* data */

/* methods */
}
```

Class Diagram in UML

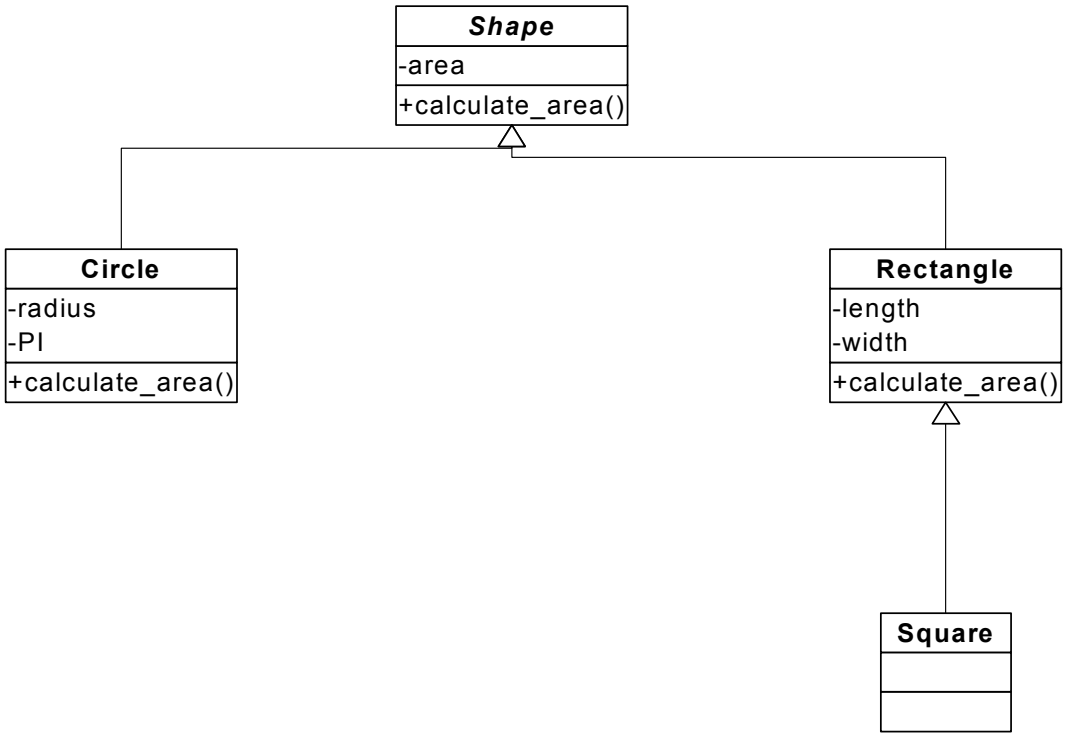


Fig 4.1 UML Class Diagram

In our discussion, we determined that we would not be able to provide a meaningful implementation for `CalculateArea()` in class Shape. This is because, in our example, the steps to calculate the area of a circle and of a rectangle are very different (in this example, Square is a subclass of Rectangle, but the methods of calculating the area is the same for both). This can also be seen in the additional attributes that were introduced in class Circle and in class Rectangle. So, because there is a declaration but no implementation for `CalculateArea()` in class Shape, the class is an abstract class.

We said that inheritance was the "is-a" relationship. As before, we can say,

A circle is-a shape
A rectangle is-a shape
A circle is-a rectangle // Incorrect!

We also discussed that classes and objects are related. An object is an instance of a class. In programming languages, a class corresponds to a *type*. More specifically, a class corresponds to a *user-defined type*. Correspondingly, in programming languages, an object would be a variable declared of that type (in general).

To illustrate our example, let us introduce the concepts of **pointers** and **references** (address-of).

A pointer is a special variable that holds the memory address of a variable. In C++, we are not allowed to create instances of abstract classes, but we are allowed to create pointers of that type. This may sound contradictory, i.e. to what are we pointing if we can't create instances? The answer to this is forthcoming. In our example, we will use the asterisk, placed before the variable name to signify a pointer (*), i.e., here is our base-class pointer[32]:

```
Shape     *ptrShape;
```

Variables are essentially areas in memory, set aside to hold particular values. The size of the area is dependent upon the type of the value it is to hold. This area in memory has an address. In some programming languages, there are operators that allow us to obtain this address. These are reference (or address-of) operators.

So let's also assume that we have created two objects, one of class Circle and one of class Rectangle, as below:

C++:

```
Circle    objCircle;

Rectangle objRectangle;
```

C#:

```
Circle    objCircle = new Circle()[33];

Rectangle objRectangle = new Rectangle();
```

[32] Note: not all languages, object-oriented or otherwise provide support for pointers. Java does not provide support for pointers. In C#, when using pointers, your code is termed "unsafe".

[33] In C# (as in Java), we must explicitly use the "new" keyword to indicate that we're requesting memory for creation of our object. This is not required in C++, as objects are implicitly created when declared.

So, given these declarations (and the inheritance relationship), we are allowed to use the pointer to Shape to "point" to either of the objects of Circle or Rectangle.

In C++:

```
ptrShape = &objCircle;      /* "&" is our address-of
                            operator */

ptrShape = &objRectangle;
```

in C#:

```
ptrShape = objCircle;      /* "&" is not used for
                           address-of in C# */

ptrShape = objRectangle;
```

We are allowed to do this because of the "is-a relationship", i.e.:

A circle is-a shape
A rectangle is-a shape

In programming language terms, Circle is of the same type as Shape and Rectangle is the same type as Shape. As such, these assignments are legal.

Let's make this assignment:
C++:

```
ptrShape = &objCircle;
```

C#:

```
ptrShape = objCircle;
```

With this assignment, we will use this pointer, `ptrShape`, to manipulate the object to which it points. All of our classes have the declaration of `CalculateArea()`. As before, it is not implemented in class Shape. `ptrShape` is of class Shape. It is reasonable to expect that we would be attempting to call the version of `CalculateArea()` in class Shape if we did the following:
C++:

```
double thisarea = ptrShape-> CalculateArea();
```

C#:

```
double thisarea = ptrShape.CalculateArea();
```

Remember, there is no implementation of `CalculateArea()` in class Shape.

Now let's repeat this for Rectangle, as follows:

C++:

```
ptrShape = &objRectangle;

double thisarea = ptrShape->CalculateArea();
```

C#:

```
ptrShape = objRectangle;

double thisarea = ptrShape.CalculateArea();
```

We know that class Circle has an implementation of `CalculateArea()`, as does class Rectangle. In actuality, `ptrShape`, after each assignment, "points" to an object of that type. In fact, `ptrShape` "knows" that it is pointing to a different object in each case. So when we invoke (i.e. call) the `CalculateArea()` method using the `ptrShape` pointer, we get the appropriate implementation of `CalculateArea()`, for each assignment and invocation made earlier. This behavior is an example of **Polymorphism**. Polymorphism describes multiple behaviors, i.e. the selection of the correct implementation of methods, from one source, i.e. our base class pointer.

Here is another example of Polymorphism:

Given the example above, suppose we have a method called `GetTheArea()` defined in some class (or globally), which takes one parameter, a pointer (or a reference in C#) of type Shape. The method would be defined as follows:

C++:

```
double GetTheArea (Shape *ptrShape) {
     return ptrShape->calculate_area()
}
```

C#:
```
double GetTheArea (Shape ptrShape)
{
     return ptrShape.CalculateArea()
}
```

Here, we see that this method could be called with a pointer actually pointing to any of our subclasses, as they all resolve down to type Shape. In this way, we could have this one method able to handle any subclass of Shape. As long as the method called inside, i.e. `CalculateArea()`, is defined in the superclass shape, we have no problems creating and executing a method such as `GetTheArea()`.

Polymorphism is available due to late-binding and method overriding. In describing Inheritance and Polymorphism, we've used classes that have the same methods declared in each of them. When base class methods are re-declared and re-implemented in subclasses, those methods are **overridden**. We say the subclass overrides the superclass method. So, in Circle and Rectangle, we see overridden examples of `CalculateArea()`, as it was first declared in the superclass Shape. With late-binding, the object being referred to becomes the target of the execution – it is not based solely on the type of the reference or pointer (as in our example). So, if we actually had a Circle object, we would want to execute `CalculateArea()` for our object of class Circle, not `CalculateArea()` for Shape, which we could not, anyway. Late-binding allows this determination to happen at run-time, not at compile time, which is otherwise the norm.

Benefits and Drawbacks of Inheritance

Inheritance allows the leveraging of similarities between classes. This could allow us to define functionality in one place, extending it as needed. This in turn, reduces redundancy and can promote more efficient production of code. Also, if the base functionality is "certified", i.e., has been tested and works, extending it lessens the "risk" of the new code, as it were. In addition, changes to superclasses are available to subclasses without necessarily changing the subclasses. Inheritance can make the job of designing, coding and maintaining somewhat easier. Also, the ability to exploit polymorphism is as a result of inheritance.

Inheritance does come at a price, though, if done intelligently, the price may not be great relative to the code reuse and convenience aspects. The run-time environment has to manage all of the facilities needed to provide inheritance and polymorphism. In addition,

whereas changes to superclasses may benefit subclasses, disaster may strike if superclasses are changed in ways which significantly change the behavior of subclasses.

Composition and Aggregation

This relationship is characterized by "has-a" (as opposed to is-a). With Composition and Aggregation relationships, we are describing "container" relationships, i.e. a class contains objects (variables) of other classes.

Composition is another example of an Aggregation class relationship, though it is a much stronger form. Composition implies that the component is an integral part of the whole aggregate. Think of a square. A square has four sides of equal length. A square cannot exist without four sides of equal length. Thus, there exists a strong ownership of the components, by the aggregate structure. We may restate this by saying the following. With Composition, there is a strong coupling between the aggregate class and the elements that are aggregated, whereas with Aggregation, the coupling is much looser.

We can use an automobile as an example of Composition. An automobile is made up (and is the sum) of many parts, such as an engine, seats, a steering wheel, wheels and tires, etc. So if we designed a class Auto, Auto would have to contain these (and other) elements, to provide the expected behavior of a car. Each of these elements are objects, with their own behaviors. However, an auto is not "valid" if any of these parts are missing. For example, imagine a car without an engine!

Differences between Aggregation and Inheritance

Inheritance relationships depict similarities between classes that fit the "is-a" form. Composition/Aggregation relationships depict a relationship between classes of the "has-a" form. The semantics are borne out by the examples above. Either of these forms of hierarchy will support the design goals of code reuse, etc. The decision of which to use must be made based on which relationship may be exploited to greatest value.

Class Associations

Associative Relationships Defined

There is yet another class relationship, Association. With Association, we are describing associative relationships between classes. To demonstrate association, let us ponder a typical sale at a merchant. A

sale is a type of transaction that involves one or more items that were for sale. Let's assume that each of these items, part of the overall inventory, is an individual object. Let's also assume that we have a class Sale, which represents this type of transaction. Then we can see that this type of transaction, a Sale, is associated with the items purchased by the customer, each of which is represented by a particular class.

Cardinality

We can use cardinality to help describe associative relationships. Associative relationships may be 1-1, 1-many or many-many. If we generalize the example above to be one between sales and items for sale, then we see this relationship is 1-many, as we can have more than one item for sale included in one sale transaction.

Here is an example:

```
class ItemForSale
{
     double    price;
     int       numberInStock;
}

class Sale
{
     ItemForSale    item;
     Date           date;
     int            quantity;
     Person         salesperson;
}
```

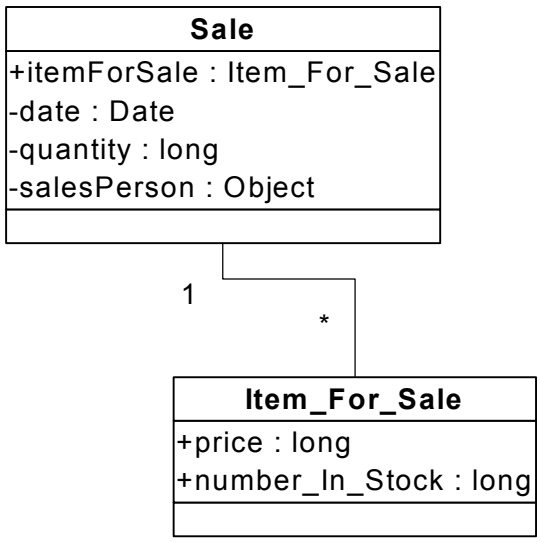


Fig 4.2 Associations

Types of Associations between Classes

Mandatory and Optional Associations

Some associations depict mandatory relationships between classes, others optional. Our example depicts a mandatory relationship between the Sale and ItemForSale classes. Obviously, we cannot have a sale if we have not sold anything! So, for each sale, there must be at least one item for sale that was included in the transaction.

Which Relationship do we Choose, When?

Each of the relationships above lend themselves to certain situations. Inheritance may be useful when that relationship makes sense, i.e. there is a "is-a" relationship between classes. In addition, we could use inheritance when we seek to leverage the similarities between groups. Inheritance also gives us the opportunity to leverage polymorphism. Aggregation, and the stronger form, Composition, allows us to create classes comprised of objects of other classes. Association allows us to define associative relationships between classes. Fundamentally, the type of relationship should depend on the behavior we are trying to implement.

What are the Costs/Benefits of Each?

Each relationship has a cost and a benefit. Some costs are not as quickly noticed as others. For example, in order for Inheritance (and Polymorphism) to work, the programming environment must support late-binding. In late-binding, the resolution of objects is postponed

until run-time, as opposed to at compile-time as usual. This requires additional structures such as vtables[34], etc. This adds overhead to the program's execution, although these structures are highly optimized so as to not add to much. Inheritance does allow code reuse, by allowing the inheritance of the functionality defined in superclasses. In addition, Polymorphism, which is available due to Inheritance, could lead to simplification of the program's code. Inheritance is also an example of tight-coupling – explained later on.

Aggregation and Composition affords us benefits as well. Here, we are able to change the structure of the component parts without affecting the aggregate. This could give us net rewards, as we may avoid wholescale changes to our system, due to a change to one component.

As all of these are available to us, we can use any or all of these, with any combination in designing our object-oriented systems.

Interfaces vs. Implementation

What is the Interface of a Class?

The interface of a class describes what is visible from "outside" that class. The visible members are those that are accessible to other objects. The interface does not refer to what is visible to methods declared inside a particular class.

Another perspective is that the interface of a class specifies the operations of a class that are visible to the outside world, without allowing the outside world the ability to see how those operations are implemented. The interface implies the class' functionality, but has no details about implementation.

Implementing a Class' Functionality

A class' functionality is the based on the definitions of methods within the class. The interface of the class is providing an interface to the functionality of the class, though not necessarily to all methods defined within the class.

[34] In order to support polymorphism, object-oriented languages have the ability to do late-binding. With late binding, certain specific types are not resolved until run-time. Virtual tables (vtables) are used to resolve (bind) types at run-time. This is how a pointer or reference to a subclass object is resolved, even though the pointer or reference may have been defined originally as a pointer or reference to a superclass.

Encapsulation and Information Hiding

Our discussion of interfaces has to start with the discussion of access control.

Earlier in Chapter #2, we stated:

> *In the OO paradigm, classes consist of data and the means to manipulate that data. The class owns both of these items. The data owned by a class may be referred to as its properties or attributes. The means to manipulate this data is via functions, collectively known as methods. The functionality provided by these methods determines the behavior (as above).*
>
> *In addition, how these methods are implemented, is kept on the "inside" of the class, i.e. hidden from the outside. In addition, the attributes of a class are not necessarily visible from the outside either, so it is more difficult for memory to be overwritten or values changed inadvertently.*

Access Level Controls

Part of the overall view of encapsulation is obtained by using access controls. Access controls restrict (or allow) access to the methods and data of a class. In general, there are three access levels for methods and data: public, private and protected.

With a public access level, class methods and data are accessible by other objects and variables. An example follows.

Let's assume we have the following class:

```
class PublicExample
{
     public int mydata;
}
```

Let's also assume we have the following declarations, outside of the class PublicExample:

```
int       thisnumber;
PublicExample  objMyObject;
```

With the public access level, this statement is legal, using our "dot" notation for member access:

```
thisnumber = objMyObject.mydata;
```

This is allowed because of the public access level. The data contained in the object `objMyObject`, is accessible from "outside", i.e. we are able to complete the assignment.

With private access, methods and data so declared may only be accessed from methods declared inside the class. The methods and data declared private are also not accessible by subclasses. So, if we revisit our earlier example, we have the following:

Let's assume we have the following class:

```
class PrivateExample
{
     private int mydata;
}
```

Let's also assume we have the following declarations, outside of the class PrivateExample:

```
int  thisnumber;
PrivateExample objMyObject;
```

With the private access level, this statement is not legal:

```
thisnumber = objMyObject.mydata;          // not legal!
```

This is not allowed because of the private access level. The data contained in the object `objMyObject`, is not accessible from "outside", i.e. we are not able to complete the assignment. However, if we declared methods inside class PrivateExample, we would be able to access `mydata`.

With protected access, methods and data so declared may only be accessed by methods declared inside the class or inside subclasses. So, if we revisit our earlier example, we have the following:

Let's assume we have the following class:

```
class ProtectedExample
{
     protected int mydata;
}

class DerivedClass : ProtectedExample
```

```
{
     //method
     public void PrintMyData()
     {/* does not return a
          value, only prints mydata */
     }
}
```

Let's also assume we have the following declarations, outside of the class ProtectedExample:

```
int       thisnumber;
DerivedClass   objMyDerivedObject;
```

We can invoke the method `PrintMyData()`, as follows:

```
ObjMyDerivedObject.PrintMyData();       // legal
```

If we've implemented this function correctly it will print the value of mydata. This is allowed because of the protected access level. The data contained in the object objMyObject, is not accessible from "outside", i.e. we are not able to access mydata directly.

```
thisnumber = ObjMyDerivedObject.mydata // illegal!
```

However, we are able to access `mydata` from inside the method declared in the derived class because it was declared protected in the base class.

The interface of a class then, is the view from the outside. In general, the public and protected data and methods constitute the interface. Of course, we're only including the protected methods for those cases when we're using inheritance and are accessing such methods and data from subclasses. If we're not using inheritance, protected and private are equivalent.

These access levels, when used with methods allow us to restrict access to the methods themselves, in addition to hiding the implementation of the methods by placing them inside the class.

Why Have Private Class Data?

Here's a quick scenario. Suppose I have a class Person, with, among other data elements, an integer field age. Let's define field age to be public. Now, based on the implied semantics of class Person, we would expect a person's age to be a positive integer (since we've defined it as an integer). Values such as -1, or any negative value

would be invalid. We wouldn't knowingly assign such a value to age, would we? Indeed, we could, and nothing could stop us because the age field is *public*. This is the sort of thing we'd want some validation to catch, before it ended up being assigned to a field. But, again, since the field is public, the assignment will happen and be completed before any validation could occur. Now, the fact that the age has a negative number could cause all sorts of problems for the program, because, the reasonable expectation of how objects of class Person would behave does not include representing folks with a negative age!

This can be prevented by making the age field *private* and defining *public* Accessor[35] methods to provide access to the data. These Accessor methods (typically Get() and Set(), or properties in Microsoft languages) allow you to have the data come in via a method which performs validation before the assignment is complete. If there is an error, it can be dealt with at the point of attempted assignment, not somewhere downstream.

Sample Project

From an analysis of each class' responsibilities, we see that there's a lot of similarity between all four classes. This strong similarity is borne out by the language of the requirements, which refers to these as different "types" of students. That would imply that each type of student is fundamentally a student, just with different attributes. Hmmm. Sounds like we may be able to identify and use our inheritance relationship, i.e. is-a. So, we have the following:

A Typical Student is-a Student
A Faculty Student is-a Student
A Transfer Student is-a Student

[35] So named because they provide "access" to private data

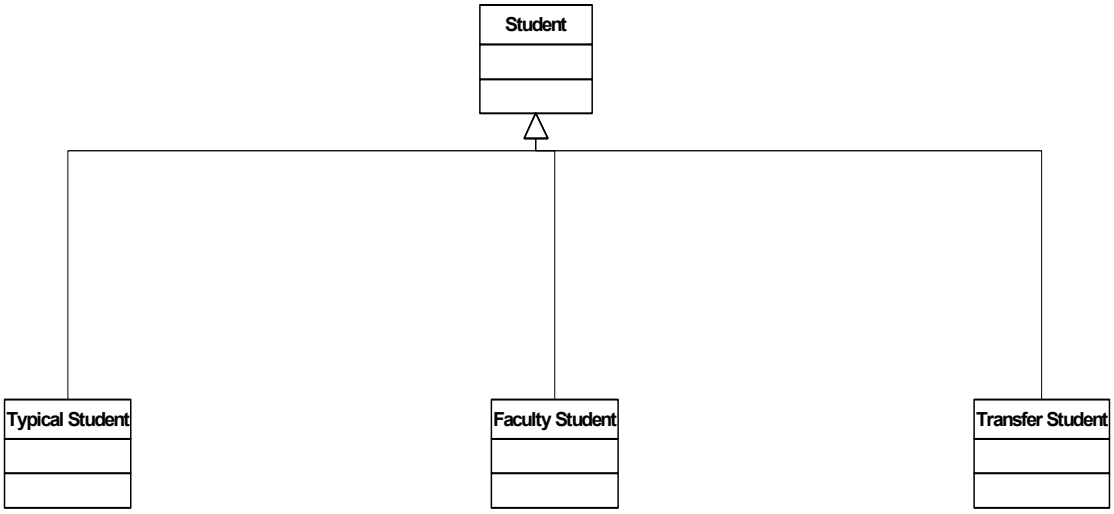


Fig 4.2 Student Diagram

As with everything else, we will have pros and cons to any decision we make. If we use this hierarchy, it would appear that it allows us the flexibility of easily having other types of students in the future, if we need to. Obviously, the need to do this is outside of the scope of the problem.

If we look at the responsibilities of the Student and Typical Student classes, we notice that they are very similar. In fact, based on the requirements, the attributes of the Student class are the same as the attributes of the Typical Student class. This could make the Student and Typical Student classes redundant. So, we could create the hierarchy above, which would give us future flexibility, or we could ignore (i.e. throw out) class Student and have our hierarchy begin with class Typical Student. If we begin with Typical Student, we still have the ability to add new student types later on. The only potential caveat would be that they would inherit the behavior of Typical Student. This hierarchy is described below:

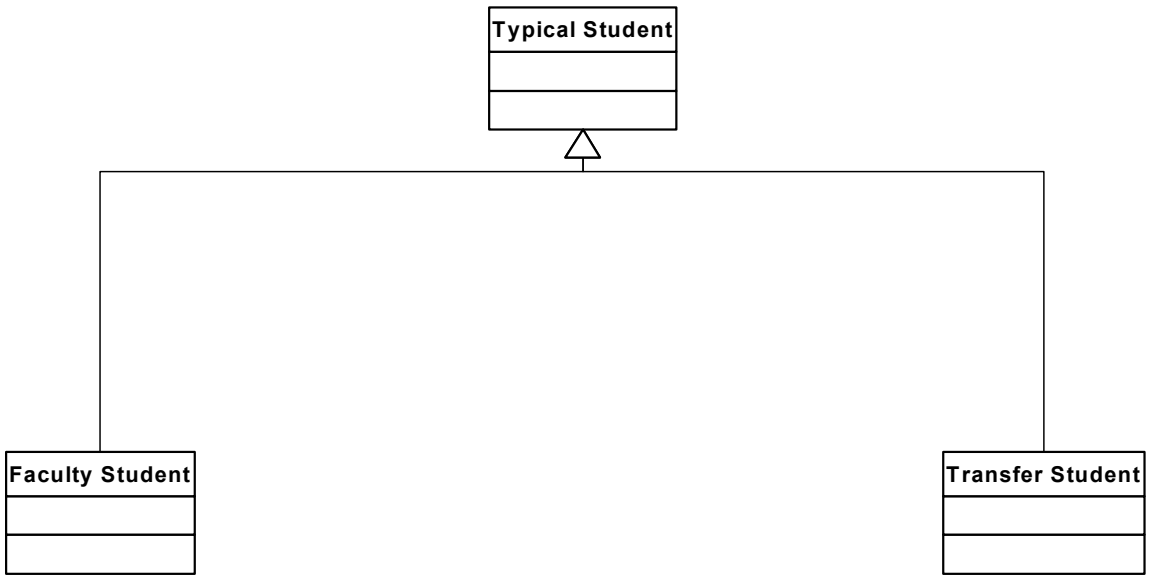


Fig 4.3 Student Diagram

What would happen if we ignored the potential inheritance relationship between the "student" classes? That gives us two options. We could have one class that represents all types of students, or we could have three separate classes, one for each type of student, as follows:

```
Class EveryStudent:
Name                  // for all three types
Address               // for all three types
Major                 // for all three types
Subject taught        // for faculty
Home college name           // for transfers
Home college address  // for transfers
```

If we had one class for each student, we would have to put all of the attributes for all three types of students into one class. While this is possible, it would mean that every time we had an instance of that class, all of these attributes would be available, whether or not they were necessary. This means, faculty student objects would have the home college name and home college address attributes, even though they may never be used for those types of students. In addition, typical student objects would have the two attributes of transfer students and the attribute of faculty students, i.e. subject taught as well, even though these may never be used either. The counterpoint is that the system is supposed to allow changes in student types, so this could be one argument for having all attributes in one class.

Let's also consider what would happen if we needed to introduce another type of student with different attributes? This would force us to have to change our one large class and that may have consequences that may have far-reaching effects on all of the other areas of our system that interact with objects of that class. In fact any change to our class may have far-reaching effects on other areas of our system. So having one monolithic class may not be wise.

Another alternative is to have each type of student be an unrelated class, i.e. not related by inheritance (or any other relationship). This is also possible. This means that while introducing a new student type would not have any effect on the other student classes, it would have far-reaching effects on the rest of the system, as all areas of the system that interact with students would how have to be modified to accept a new student type, in addition to the others. Also, having separate classes could mean increasing the complexity of the system, as now the system has to manage 3 or 4 separate unrelated items, as opposed to 3 or 4 related items, allowing us to leverage inheritance and polymorphism, for example.

We will revisit our decisions and the alternatives as we evaluate our design using the metrics (last chapter).

If we look at the Program and System abstractions and their responsibilities, etc., we see some similarities here also. It appears we have to do for these classes what we did before for the Student and Typical Student classes. We may discard Program and keep System.

If we look at the College abstraction, we see this abstraction represents the name and address of colleges, from which transfer students come. We have a choice here. We could just keep the college information as attributes directly in the Transfer Student class, or we can use this abstraction and use a composition relationship between the Transfer Student class and the College class. What are the criteria for our decision? If we place the information for colleges in the College class, we are acknowledging that we have more than one piece of information regarding colleges and we are allowing for changes to the information we keep on colleges to be made more easily. We are also potentially introducing more complexity, as we will have another class in our system representing colleges. However, as a rule, it is wiser to only include "atomic" values as attributes in a class. The information we need to keep for colleges is not "atomic" because

we need to keep two pieces of information on colleges, i.e. the name and address of the college. So we keep the College class.

We will also keep the Name and Address abstractions. The reasons outlined above for the College class applies here as well. The information we need to keep for names and addresses are not "atomic", so we will use an abstraction for each of these.

In contrast, the information needed for a student's ID seems to be "atomic". We don't have enough information in the requirements to determine that a single value (i.e. "atomic") would not be sufficient. Possible ID's could be a student's social security number or a generated number that is guaranteed to be unique across all students. In either case, we are left with only one value. So we do not need an abstraction for ID. Instead, we will let ID be an attribute of a student.

Let us examine the Major and Subject abstractions. We do not have any information about Major to suggest that it will not be single-valued, i.e. just the name of the major. So we may or may not need an abstraction for Major. For Subject, we know we have two pieces of information for each subject, the subject's name and number of credits. In addition, we also know that each student will be taking one or more subjects. So that implies that we need to keep a list of subjects for each student. Regardless, the same criteria we've used above is relevant here. If we include the Subject information with the attributes of a student, then it causes us to have to manage changes to individual subjects. So we don't need the Major abstraction, but we will keep the Subject abstraction.

The Faculty abstraction would represent all members of the faculty of a college (including those that were students). We could leverage Multiple Inheritance. We have a class representing Students and a class representing members of faculty. Effectively, a class representing both gives the behavior of Faculty Students. However, we do not have any information in the requirements to describe what behavior the Faculty abstraction would have. As a result, we will not utilize this abstraction.

We're left with two abstractions to review, that of Discount and Semester. Even though there may be more than one discount in effect for a student, i.e. a faculty member that is also a student over 55 will get the benefit of both discounts. However, that resulting discount will be single valued. Regarding Semester, based on the requirements, we are only keeping information for one semester

anyway. So that implies this is single-valued also. From this, it seems we do not need either abstraction.

So, of our initial list of abstractions (candidate classes), what are we left with?

Typical Student
Faculty Student
Transfer Student
Home College
Name
Address
Major
Subject
System

Are there any other abstractions that we need? Definitely. Some of these are as follows. Each student may take one or more subjects. This implies we need to maintain a list of subjects for each student. Our choices are to manage this list in the Typical Student class or to create another class to do that for us. For this example, for clarity and consistency, it would be better to have the list of subjects as an abstraction. An object of this class would then be included as an attribute of the Typical Student class.

Students may have up to two majors. We could use the same guidelines we used above and develop an abstraction for a list of majors for each student also.

Per the requirements, the school may add new classes and new majors periodically. This implies that the school (system) needs to maintain a list of all available classes and a list of all available majors, from which students will choose. This then implies two new abstractions, one for each of these lists.

We also need to manage a list of all students. Thus, we will create an abstraction to do this also.

Our list of abstractions is now as follows:

Typical Student
Faculty Student
Transfer Student
Home College

Name
Address
Subject
Major
System
Student Majors
Student Subjects
All Majors
All Subjects
All Students

Chapter Summary

- Inheritance is the relationship where one class is the superclass and others inherit from or extend the functionality defined in that class.

- Polymorphism ("having multiple forms" describes the ability to use superclass references (or pointers) to manipulate subclass references (or pointers) due to inheritance relationship.

- Classes may also have optional or mandatory associative relationships.

- The interface of a class (or object) describes what is visible from outside the class (or object).

Exercises

1. Create an object-oriented model of a directory of files on a computer. Create a class diagram to describe the model.

Chapter 5

Object Structure and Relationships

In the previous chapter, we continued our discussion of class relationships with the discussion of Polymorphism, Composition (Aggregation and Association) and Access Controls. In this chapter, we will discuss the objects and their interactions.

What is an Object?

To recap, an object is a specific instance of a class. As we stated earlier, an object is constructed based on a "blueprint" that is the class specification.

Structure of an Object

A class is a structure containing methods and data. The construction of a class is a design-time activity, i.e. you decide what the elements of your class will be before run-time (execution).

An object is an instance of a class. You may also view an object as one of the members of a class. These two terminologies mean the same thing – an object has to obey the rules set out by the class' design.

Objects are built or created based on a "blueprint", i.e. the class structure. This means at run-time, memory is allocated for each object to be created, based on the fields and methods in the class.

For each object created, there is then, a block of memory to hold fields and methods. While this statement is generally true, in fact, there are some twists we must be aware of.

Instance Fields

As mentioned above, each object has memory allocated for fields. In general, then, each object has its own copy of the fields defined in the class. These fields, for which each object has it's own copy, are called *instance fields*. This also means each instance of the class (or object) has it's own copy.

Example:
If we use the Circle class from before, then we have:

```
class Circle : Shape
{
/* data */
     public const double PI = 3.14159;          // rounded
     double radius;

/* methods */
     double CalculateArea()
     {//implementation
     }
}
```

An instance (object) of class Circle would be represented as follows:

```
     Circle    objCircle = new Circle();
```

We would then expect to be able to execute statements such as:

```
     double    x,y, area;

     x = objCircle.PI;
     area = objCircle.CalculateArea();
```

We would be able to execute similar statements for every object of class Circle that we created.

Class Fields

We could argue that it is redundant for each object of class Circle to have its own copy of PI, as it doesn't change. So, ideally, we would like to have one copy of PI that could be shared by all objects, instead of one copy for each object, as implied above. Fields for which there is one copy, shared between all instances of a class are called *class*

fields[36]. There is one copy of a class field, in a special area of memory that is shared by all instances of the class.

Here is another example of the need for class fields. Let's imagine a mechanism for counting the number of objects of a class created by a program. We've said objects are unique, have a state, have their own copy of data, etc. Nothing we've said thus far has implied that the number of these objects is available to us, for example. One way to keep track of the number of individual objects of a particular class is to utilize a class field. You would increment the class field every time an object was created, and decrement the class field when the object was destroyed.

Class fields are also referred to as *static fields*, based on popular object-oriented languages such as C++ and Java. In our pseudocode, we will also use the keyword static to describe class fields.

We may then rewrite class Circle as follows:

```
class Circle : Shape
{
/* data */
     public static double PI = 3.14159;          // rounded
     double radius;

/* methods */
     public double CalculateArea()
     {
     }
}
```

Methods

As discussed above, method definition is a design-time activity. By the time you've written your program, you've also defined your class' methods. Unlike fields (instance or class), methods do not change during execution. The statements in a method may be passed different parameters, but the statements themselves (and the methods) do not change. As such, object-oriented languages as C++ and Java allow objects to share methods. So each method accesses (and effectively gets a copy of) the methods (and their local variables) of a class at execution time.

[36] Class fields are also termed static fields in languages such as C++ and Java.

Object Initialization

We've stated that an object has state, identity and behavior. This implies that each time we create an instance of a class, we expect that object to behave as certain way, based on the blueprint (class definition). In fact, we expect an object to be ready for use. This also implies that we expect any initializations to be done prior to use.

A *constructor* is a special method that is used to initialize objects prior to their use.

Let us revisit our earlier (non-static) example of the class Circle:

```
class Circle : Shape
{
/* data */
     public const double PI = 3.14159;          // rounded
     double radius;

/* methods */
     public double CalculateArea()
     {
     }
}
```

Let us create a particular circle, as follows:

```
Circle         objCircle         // based on our pseudocode
```

If we create an instance of this class, what value do we expect radius to have? Depending on the environment used for implementation, radius may be initialized by default to be 0 or 0.0 (integer or real). Just as likely, radius may not be initialized at all. Regardless, if we create a particular instance of Circle, we do not have control over what initial value the field radius has initially. This is potentially dangerous and could lead to system errors. For example, if the field radius is initialized with a negative value, say, what does that mean for the class Circle? That may not be appropriate. Instead, what we need to ensure is that each circle is created with an initial value that is appropriate. We will need to explicitly create a constructor to accomplish this.

Object-oriented environments typically provide a default constructor for cases where a constructor isn't explicitly defined. This may or may not address issues such as the initial value of a number, as outlined

above. Default constructors have other limitations. In cases where the design of a particular class includes dynamic memory allocation, default constructors would not be able to handle these scenarios, as the constructor would not be able to make appropriate decisions.

Constructors are methods that are executed prior to the object being ready for use. This is important. It is important that the constructor complete its tasks prior to the object's use, as this will properly initialize the object's fields. While it is possible to set initial values of the objects after creation, i.e. not using a constructor, there is no control over what values are used to initialize the object's fields if initialization is done this way. Neither is there any control over the completeness of the initialization. Either or both of these potential issues can lead to hard-to-find problems later on.

Constructor Usage

One of the most important aspects of constructor usage is to remember why you're creating the constructor in the first place. Earlier, we talked about classes having "semantics", i.e. rules. A class is a programmatic representation of something concrete in the real world. Whatever we're representing will have a set of rules that we use to make sure it is valid. Many of these rules may apply to what the real-world item is like when it is created or made available, i.e. initially. In our programmatic world, these rules are enforced using constructors. Here is an example. Let's say we are creating a program that represents a group of students in a classroom, where each student is represented by an individual object. Each student object would have a name, address, student ID, to name a few, as a real student (after which our object is modeled) would. In the real world, would a student without a name be allowed in the class? How about without a student ID? If the answer to any such question is "no", then in our program, we cannot allow student objects to exist without having the required information within. How do we enforce this? We create constructor(s) that require the necessary information to be passed as arguments. In this example, it will never be ok to have a student object without this information. So, in addition to creating constructors that require this information, we must not create a default constructor for this class. Remember, the system only provides a default constructor when you do not explicitly define one. So in this case, having defined one or more constructors to enforce your class' rules, the system would try to utilize those constructors only.

Suppose you have a class that you don't want instantiated? What can you do to enforce this? This is just another "rule" that a constructor can enforce. We mentioned above that constructors are public methods that are executed after the memory for an object is allocated, but before the object is ready for use. We also said that if no constructors are defined, the system will provide one for you. Now, what would happen if there was a constructor, but it was not accessible, i.e. not public. Well, an object of that class would not be able to be instantiated, because a critical part of the instantiation process would be inaccessible. Thus, when you define a private constructor in a class, you will not be able to instantiate that class. This has a very different meaning from defining abstract methods in a class. In this case, your class definition is complete, i.e. your class' methods are fully implemented – you just don't want anyone instantiating your class. In the case of abstract classes, the class definition is incomplete, as you do not have full implementation for all the methods of your class (which is why it is an abstract class).

Object De-initialization

An object has a finite lifetime. This lifetime starts when it is create and ends when it is destroyed. Different systems manage object destruction differently. However, there are similarities. Memory that was allocated for the object needs to be reclaimed. In some cases, dynamic memory was allocated for a particular object, i.e. "inside" an object. This dynamic memory has to be de-allocated.

As with constructors, destructors are also provided by default, for cases where a destructor has not been explicitly defined. As with default constructors, though, default destructors do not handle de-allocating dynamically allocated object memory well. In fact, using default destructors in these instances will typically leave this memory behind, thus causing a memory leak.

More recent languages provide "garbage collection" to help with object destruction. Languages such as Java and the Microsoft .Net family, have mechanisms in their platforms that keep track of objects using reference counting, i.e. the keep track of the number of references to an object. Once the number of references to an object reaches zero, the next time the garbage collection process executes, this memory allocated for this object can be returned to the heap. Languages such as these do not provide explicit destructors the same way as languages such as C++ do. They provide a language and/or platform features that allow you to call or implement methods that help you

deallocate dynamic memory used by the object, but such methods do not deallocate the memory used by the object itself at that time. Such mechanisms are quite helpful in preventing memory leaks. However, they are not foolproof, so care must still be taken with dynamic memory allocation in objects.

Objects and Access Levels

As we've mentioned before, classes provide the blueprints for objects. Classes also define levels of access. Objects, being instances of classes, are constructed based on these blueprints and also adhere to the access levels. Thus, the examples outlined last chapter (for public, private and protected access), apply to the objects of those classes as well. In fact, the access levels defined in the class, i.e. at design-time, are meant to control the interaction between the objects at run-time. Each of the diagrams presented above would also be able to reflect these constraints.

Class-Object Relationships

Earlier, we introduced class hierarchies, i.e. inheritance, aggregation, composition and association. Let us revisit aggregation and composition. Both aggregation and composition are examples of the "has-a" relationship. In both cases, the class that is the aggregation or composition has objects of other classes as its member variables (in addition to other member variables as necessary). This means, at design time, your class (i.e. the aggregate or composite) will already be demonstrating object interactions, even though we would expect object interactions to occur only at run-time. As a result, classes that are aggregations and compositions exhibit class-object interactions, because the class (the aggregate or composite) is interacting with the objects it is an aggregate (or composite) of.

Objects and Inheritance

Last chapter, we discussed inheritance and polymorphism. Let's look at an object's structure to determine how inheritance and polymorphism work.

Earlier, we said a new class introduces a new *data type* to the compiler. All objects are instances of a class. This means, each object is of a specific type also. The type of an object is the class of which it is an instance. What about objects of a subclass? What type are they?

In inheritance hierarchies, subclasses are related to superclasses by the "is-a" relationship. If we take "is-a" a step further, we see that this will have an impact on what the type of a subclass' objects are. Each object of a subclass will have that subclass as its type (as above), but, in addition, will also have that subclass' superclass as its type also. In general, the object of a superclass will also have that subclass' superclassses as types, as many as there are.

The structure of the object of a subclass will then also include all of the elements of that subclass' superclass, even those that are not public. This is true, because, the "is-a" statement would not be true if we left some parts (that were defined in the superclass(es)) out of the object. Everything is included, because that is the definition as described by those superclasses. Let's say the diagram below represents an object of the superclass in this example:

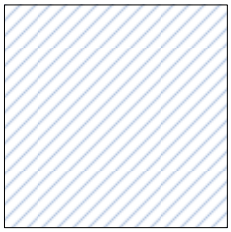


Fig 5.1: Superclass object representation

Then, an object of the subclass could be represented by this diagram:

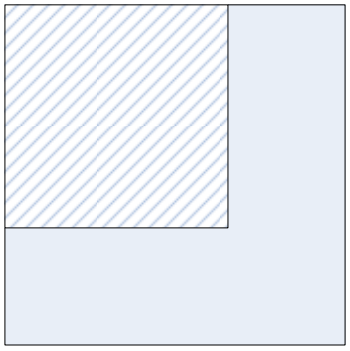


Fig 5.2: Subclass object representation

As the diagram indicates, the definition of the superclass is "included" in the definition of the subclass (via inheritance). As such (also indicated by the second diagram), the object of the subclass is actually of two types, as outlined earlier – the subclass itself, and the type of its superclass.

In the previous chapter, we looked at polymorphism. The mechanism we've outlined above is actually what allows us to utilize polymorphism to our advantage in our programs. It is the fact that each subclass object is its own type, as well as the type(s) of its superclass(es) that allow us to use references and/or pointers of a superclass (base class) type to manipulate objects of a subclass type. This is possible because each subclass object "is-a" superclass object as well – so there is a "type equivalency" between the superclass and subclass types. As we mentioned earlier, if any part of the superclass' definition was omitted in the construction of the subclass object, the "type equivalency" between superclass and subclass types would not be correct.

Subclass Initialization

From the diagrams and discussion, we know we need to have a "complete" superclass object in order to have a valid subclass object. Earlier in this chapter, we saw how we can use constructors to enforce a class' semantics. Might these superclass constructors affect the creating of subclass objects? Absolutely! We need to adhere to these rules to create superclass objects, and without valid superclass objects, we will not have valid subclass objects.

Many languages provide elements that allow us to call an immediate superclass' constructor from inside a subclass' constructor. In many cases, such a call must be the first executable line in the subclass constructor. This allows the superclass "part" of the subclass object to be created properly, i.e. adhere to the rules, during the subclass object's construction process. Here is an example:

```
//C# syntax
// Superclass 2DPoint
public class 2DPoint{
     int x,y;  //represents 2-dimensional point in space

     public 2DPoint(int x, int y){
          this.x = x;
          this.y = y;
     }
}
```

```
//Subclass 3DPoint
public class 3DPoint : 2DPoint{
     int z;          //represents 3rd dimension in space

     public 3DPoint(int x, int y, int z){
          base (x,y);          //in Java, this would be:
                               //super(x,y);
          this.z = z;

     }
}
```

In this example, 2DPoint objects must have x- and y-coordinates at time of instantiation. Since the 3DPoint class inherits from the 2DPoint class, in order to have a 3DPoint object, we must have created a valid 2DPoint object. So, the call to `base(x,y)` in this example, ensures that we have a properly constructed superclass object (2DPoint) before we continue initializing the 3DPoint object.

Object Interactions and Relationships

At run-time, a system's functionality is provided by all of the objects and their associated interactions. It is important to be able to represent this aspect of the system. The diagrams we have reviewed thus far are not capable of capturing a system's run-time behavior. The diagrams we have reviewed thus far capture static views of the system. In fact, we need entirely new diagrams to model this behavior.

Modeling Object Behavior at Run-Time

What behavior are we trying to model? From the discussions earlier in this section, we saw that objects have distinct lifetimes, i.e. they are created (instantiated), provide some functionality while they exist and are then terminated. During their lifetimes, objects communicate with each other. As before, in an object-oriented system, the operation of the system is based on the cooperative interaction of objects. Objects communicate via messages. A message is a method invocation. So, if I have two objects, A and B, then A passes a message to B if A invokes a method of B. Likewise, B passes a message to A if B invokes a method of A. This demonstrates possible links between objects A and B. In addition, at any given point in time, objects have states, i.e. the value of the object's attributes.

We use class diagrams to represent the static elements of a system. These static elements are the classes that form the blueprints of all of the objects in the system. We need to model the dynamic elements of the system. This will be from a more "real" perspective.

The dynamic elements of the system include the following scenarios: We need to model the objects of the system at a particular point in time (instances in time).

We need to model how groups of objects collaborate in some behavior, i.e. the behavior of a single use-case. We need to model the objects and the messages that are passed between these objects within the use-case (interactions).

To model the objects of the system at a particular point in time, we use object diagrams (also called instance diagrams). These diagrams represent each object and the messages that are passed between them at some discrete point in time.

To model how groups of objects collaborate, we use interaction diagrams. In UML, there are two types of interaction diagrams – Sequence and Collaboration.

Sequence Diagrams

Sequence diagrams are useful for modeling how groups of objects proceed, over time, to provide the functionality required by (or to satisfy) a particular use case. As mentioned earlier, a use case is a description of a particular usage scenario. Sequence diagrams focus on the sequence of method calls, rather than the relationship(s) between the objects involved (see Collaboration Diagrams below). The Sequence diagram allows you to see the functionality provided by methods in each object participating in the use case.

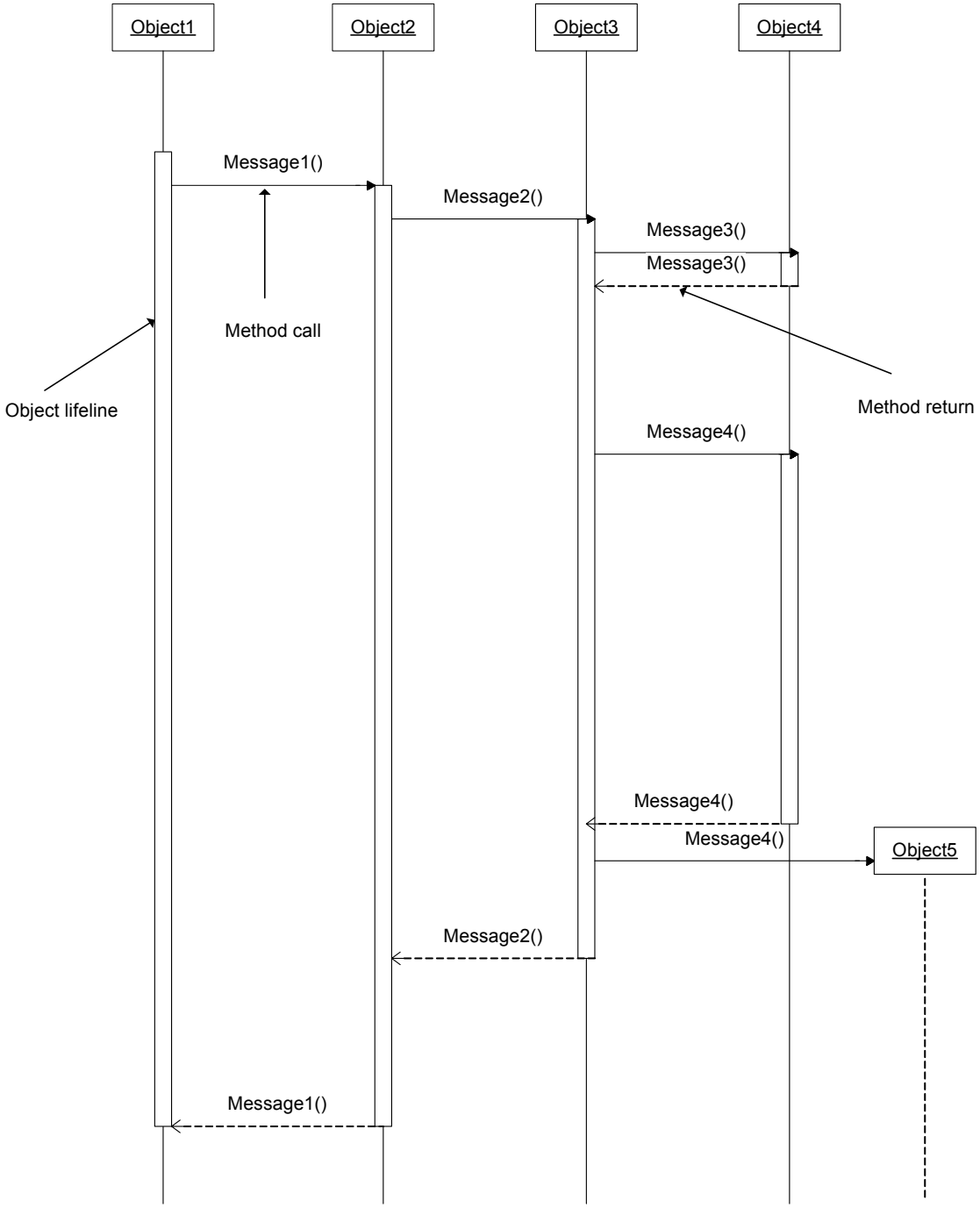


Fig 5.1 UML Sequence Diagram

Let us review the elements of the diagram above. The boxes at the top of the diagram represent objects. We use this diagram to understand sequence in which methods of these objects are called as they collaborate in order to satisfy some functionality (described by a

use case, for example). From each object downward is a dashed line. This line represents the "lifeline" of the object, i.e. the time during which the objects exist (lifetime), in the context of the functionality we're reviewing. Between lifelines, we see arrows. Each arrow represents a message sent between objects (method call). On each lifeline, there are narrow rectangles that. Each arrow (message) extends between two narrow rectangles. Each rectangle shows the "activation time" of an object. They show the time it takes for each method to complete.

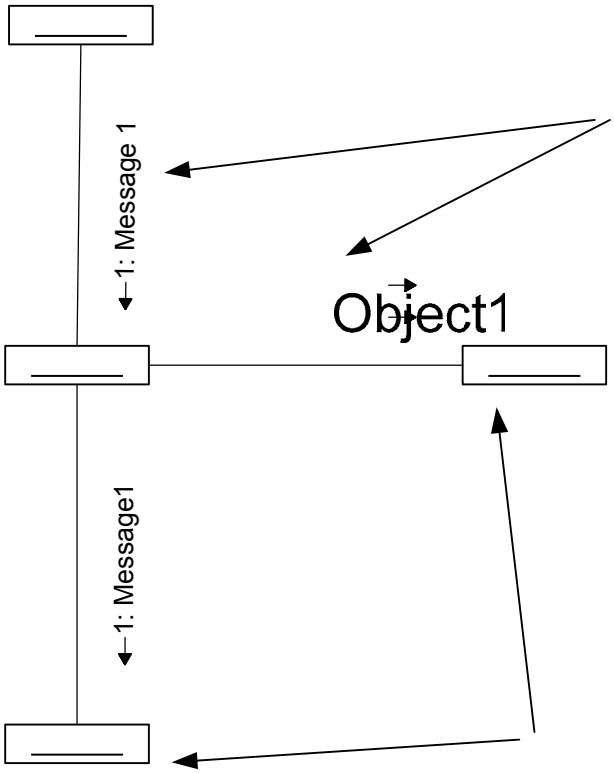
Implicitly, at the end of the method, there is a return to the calling object. However, the sequence diagram can be drawn to include explicit return calls to the calling object. These explicit returns are depicted by dashed lines, in the opposite direction to the method call.

Sequence diagrams can also depict asynchronous method calls (synchronous by default). In the case of asynchronous calls, instead of a "full" arrowhead, a "half" arrowhead is used for each "half" of an asynchronous message.

Collaboration Diagrams

In addition to Sequence diagrams, we may also use Collaboration diagrams to describe how a use case is satisfied. Collaboration diagrams focus on the way several objects collaborate (i.e. work together) to accomplish some unit of work. They focus on the relationships between objects to a greater degree than the sequence in which the methods on those objects are called. These diagrams also make it easier to reconcile the activity depicted with the class diagrams (static model) representing the system.

A Collaboration diagram shows the interaction between object via numbered messages. With this diagram, the sequence of interactions is not as easily evident as in the Sequence diagram, but we are able to see, via the spatial layout, how the objects are linked together. UML uses a decimal numbering scheme to make it clear which operation is calling which other operation, though it makes it harder to absorb the overall sequence.



1: Message1
2: Message2

Fig 5.2 Annotated UML Collaboration Diagram

Object2

Objec

Let us decipher what the diagram above depicts. Each rectangle is an object. The lines between the objects in the diagram are links that represent relationships (association, aggregation or composition) between objects. The arrows indicate method calls and their respective directions. The numbers associated with each arrow indicates the sequence in which the methods are called. They can be grouped, using the "." (dot) notation to show which methods are called from within method calls.

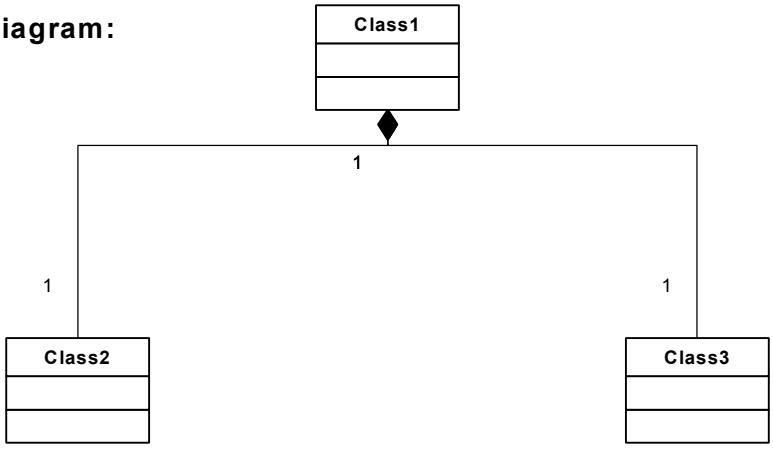
Object Diagrams

In general, to view the objects of our system at a particular point in time, we can use generic "object" diagrams, also referred to as instance diagrams. Object diagrams show the interactions between objects, without describing the sequence in which the messages occur or describing the messages. So, an object diagram describes an example configuration of objects. An example of an object diagram follows.

Object4

C

Given this class diagram:



An example object diagram is as follows:

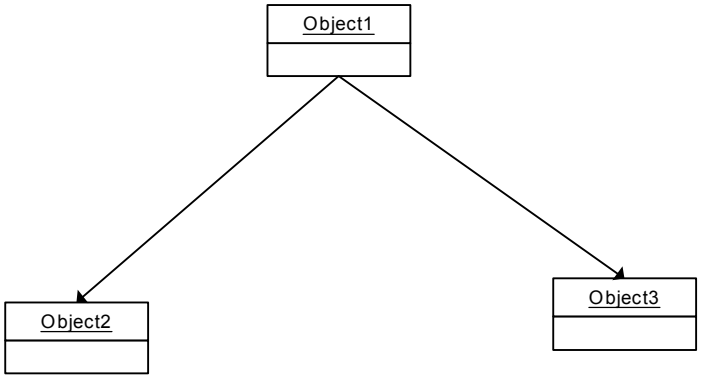


Fig 5.3 UML Diagrams

State Diagrams

The state of an object is the value of its data attributes (fields, etc.) at a given point in time. To capture the objects' states visually, we use state diagrams. State diagrams allow us to see how the object reacts to various messages over time.

Unlike the previous diagrams, we use state diagrams to really put the focus on how one object's data values change over time. If the data in an object is private, the object's data will only change in response to its public methods being called. Thus, each individual state of the object would be caused by one or more method calls.

Each state of the object is something that is determined by you, the designer. Each state (i.e. value of the data in the object), may be given a name to be easily identified. A simple example of a state diagram follows.

Each state (represented by the oval) is a particular state of an object as a particular point in time. As a result, a state diagram is used to depict various states of an object (one per diagram) and the messages, conditions, etc. that cause states to change (transitions), as the object moves from the initial state to the final state.

Simple state diagram

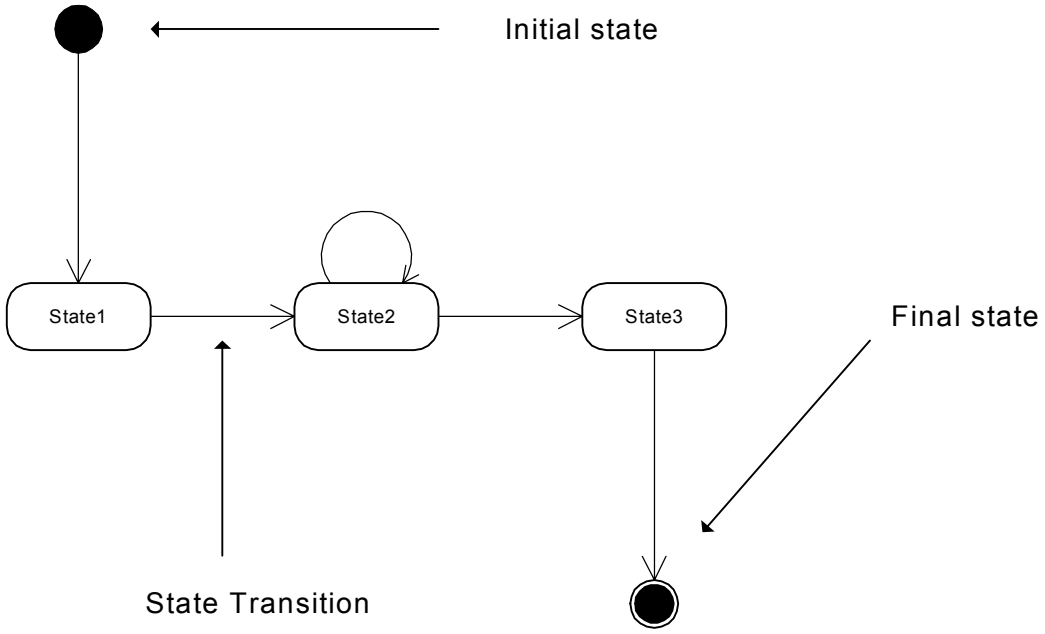


Fig 5.4 UML State Diagram

Static vs. Dynamic Modeling

Static models, i.e. class diagrams, do a great job of describing the overall structure of a system's classes. They are termed "static" because this structure cannot be changed during execution. It is fixed during the analysis and design phases of development. So, while the number of objects in existence may vary over time, the structure of each object, regardless of how many there are, will not change while the program is running.

It is clear, that static models do not provide a clear understanding of how a system will behave, while it is running. We have to find another perspective for views of the system while it is running. This is what dynamic modeling allows us to do.

Sample Project

Our list of abstractions is now as follows:

Typical Student
Faculty Student
Transfer Student
Home College
Name
Address
Subject
Major
System
Student Majors
Student Subjects
All Majors
All Subjects
All Students

We know have to describe all of the attributes (and methods) for each of our abstractions. In doing this, we may realize that we have to make changes to our list of abstractions. Let's also begin by making all classes' attributes private. This implies that we have to define public accessor methods for each class. I will include these accessor methods below as well.

Class Typical Student

Attributes:
Student Name (Object of class Name)
Student Address (Object of class Address)
ID
Majors (Object of class Student Majors)
Subjects (Object of class Student Subjects)
Grade
Discount

Methods:
GetStudentName()
SetStudentName()
GetStudentAddress()
SetStudentAddress()
GetStudentID()

SetStudentID()
GetStudentMajors()
SetStudentMajors()
GetStudentSubjects()
SetStudentSubjects()
GetStudentGrade()
SetStudentGrade()
GetStudentDiscount()
SetStudentDiscount()

Class Faculty Student

Attributes:
All attributes of Typical Student
Subject Taught
Date Employed (Start of employment - used to calculate length of service)

Methods:
All methods of Typical Student
GetSubjectTaught()
SetSubjectTaught()
GetDateEmployed()
SetDateEmployed()

Class Transfer Student

Attributes:
All attributes of Typical Student
Home college (Object of class College)

Methods
All methods of Typical Student
GetHomeCollege()
SetHomeCollege()

Class Home College

Attributes:
College Name
College Address (Object of class Address)

Methods:
GetCollegeName()
SetCollegeName()

Class Name

Attributes:
First Name
Middle Initial
Last Name

Methods:
GetFirstName()
SetFirstName()
GetMiddleInitial()
SetMiddleInitial()
GetLastName()
SetLastName()

Class Address

Attributes:
Street Address
City
State
Zip

Methods:
GetStreetAddress()
Set StreetAddress()
GetCity()
SetCity()
GetState()
SetState()
GetZip()
SetZip()

Class Major

Attributes:
Name_of_Major

Methods:
GetMajor()
SetMajor()

Class Subject

Attributes:
Subject Name
Credits

Methods:
GetSubjectName()
SetSubjectName()
GetCredits()
SetCredits()

Class System

Attributes:
Majors (Object of class All Majors)
Subjects (Object of class All Subjects)
All Students (List of objects representing all types of students)

Methods:
GetMajors()
SetMajors()
GetSubjects()
SetSubjects()
GetStudents()
SetStudents()

Class Student Majors

Attributes:
Majors

Methods:
GetMajors()
SetMajors()

Class Student Subjects

Attributes:

Subjects

Methods;
GetSubjects()
SetSubjects()

Class All Majors

Attributes:
Majors

Methods:
GetMajors()
SetMajors()

Class All Subjects

Attributes:
Subjects

Methods:
GetSubjects()
SetSubjects()

Class All Students

Attributes:
Students

Methods:
GetStudent()
SetStudent()

Let's assess where we are so far. We've discussed attributes and accessor methods. Let's discuss the other methods we need, per the requirements.

From the requirements, we see that we have to provide the following functionality:
1. Adding a new student's information
2. Searching and displaying a student's information
3. Deleting a student
4. Changing/assigning classes and credits to students
5. Changing/assigning a student's major

6. Changing/assigning a student's type
7. Changing/assigning a student's status, i.e. full-time or part-time according to the rules above
8. Producing reports as follows:
   Sorted list of full-time students (all information)
   Sorted list of part-time students (all information)
   Number of students of each type (typical, faculty and transfer)
   For each type of student, a sorted list of student names and addresses
   For each type of student, a reversed list of student names and addresses
   List of all students, their majors and number of credits
   A sorted list of all students based on their cost for the semester

We will refine our design in the next chapter.

Chapter Summary

- An object is a structure containing methods and data.
- Each object gets its own copy of an instance field.
- All objects in a class share one copy of a class field.
- Objects are initialized using a specialized method called a Constructor.
- Objects are destroyed using a specialized method called a Destructor (depending on language - not always available).
- Object behavior may be captured at runtime using UML Sequence diagrams, Collaboration diagrams and State diagrams.

Exercises

1. Give an example of situation where class fields (as opposed to instance fields) would be required.

2. What changes the state of an object?

3. How do objects collaborate at run-time?

4. What aspects of a system do class diagrams capture?

5. What aspects of a system do object diagrams capture?

Chapter 6

Designing with Classes and Objects

Design is primarily a refinement of the analysis model. It incorporates non-functional requirements of the system and the constraints of the environment to transform the analysis model into something that can be coded. Design must anticipate and factor in issues such as memory constraints, exception handling consistency, scalability, etc.

Design is where the requirements, as abstracted by the activities in Analysis, head toward being made real and concrete (i.e. implemented). We refine the models from Analysis, as required. In Design, models are produced as well to augment those produced in Analysis. The models depict aspects of the system that are real, not abstract. In addition, these models can be directly implemented in code.

In this chapter, we will continue exploring the object oriented design process using the tools and concepts we've discussed in the previous five sessions. In addition, we will look at popular abstractions that we can use in developing our design.

We will also discuss the elements of good system design. We will concentrate on the elements of good class design, as this activity is the cornerstone of good object-oriented system design. This will also allow us to benchmark where we are in our course example, to make sure we're on the right track. We will look at various aspects of design, patterns, elements, guidelines and metrics.

Overview

To effectively design object-oriented systems, we must grasp the relationships between our design strategies, language constructs and

the software engineering goals the combination of these is meant to achieve. After all, there is a reason why we choose to do object-oriented development in the first place.

In order to design anything, we must first have a solid understanding of what problem we are trying to solve (what) and the methodology we will use to solve it (how). These sound almost trivial, but they bear repeating as a solid understanding of these two elements early will allow you to avoid costly pitfalls later.

Design Guidelines

In object-oriented design, our fundamental building blocks are classes and objects. Classes are the static structures created at "design-time". At run-time, its the objects that are collaborating to provide the functionality of the system. Objects are dynamic. They exist at runtime and are created dynamically.

In order to have a good object-oriented design, we must:

- Determine what the classes and objects will be. This is accomplished by performing the two steps below:
    - Identify the data to be contained in each class. This may be accomplished using the techniques outlined in Chapter two.
    - Determine the operations to be defined on each class. As before, it is an iterative process.
- Identify any hierarchical relationships between the classes and objects.
- Define which operations will comprise the class' interface, i.e. public methods.

These steps are part of an iterative process. How do you know you're done? You're done when you cannot make any more meaningful refinements to your list of classes. You will find that they will need to be repeated as necessary. It is unreasonable to expect perfection the first time through. In addition, there may be additional classes that are not immediately evident from the requirements but which are discovered as a result of the analysis and design.

Some of the changes that may arise as part of these iterations may have greater effects than others depending on whether the action is a change to a public area of the object or to a private area of the object.

The architecture of an object-oriented system encompasses its class and object structure. This structure is essential to constructing a system that is understandable, extensible, maintainable and testable. According to Grady Booch[37], good software architectures tend to have several attributes in common:

- They are constructed in well defined layers of abstraction, each layer representing a coherent abstraction, provided through a well-defined and controlled interface and built upon equally well-defined and controlled facilities at lower levels of abstractions.

- There is a clear separation of concerns between the interface and implementation of each layer, making it possible to change the implementation of a layer without violating the assumptions made by its clients.

- The architecture is simple: a common behavior is achieved through common abstractions and common mechanisms.

The idea behind object-oriented design is to create a set of models that identify and use real-world elements and represent them by programming language elements. This is made possible by the programming language's ability to support the fundamental elements of object-oriented development
.
From what we've seen, the process involves identifying these classes and objects and then building the system based on the collaboration of these elements. Thus, object-oriented design is a way of designing a system based on classes and objects.

Object oriented design is based on the notion that a program is better if it more closely "resembles" the elements in the problem domain. By "resembles" we mean uses language from the problem domain and uses abstractions based on (i.e. representing) elements in the problem domain (real world).

Let's restate what we've seen so far: In object-oriented design, we have various tools and methods at our disposal that were not available to us (at least not in the same way) with structured programming. We have object-oriented elements (abstraction, encapsulation, hierarchical relationships, modularity and persistence) that we can identify and

[37] Object Oriented Analysis and Design with Applications, 2nd Edition, Benjamin/Cummings Publishing Company, 1994

leverage. Let us look at how leveraging these elements will help us create better programs.

Abstraction

Abstractions (via abstract data types, classes, etc.) are valuable tools for reducing the overall complexity of a problem. They allow you to write your program in layers and to write the program in terms of the problem domain, rather than in computer science terms. We can use abstraction to create programmatic elements that represent real-world elements.

With abstraction, we can focus on the important details and ignore information that is not relevant to our solution. If you had to focus on every last detail of information all the time, you would accomplish nothing. Abstractions are used as representations of real-world objects, as discussed in Chapter 2.

The objects that you will identify and design will typically fall into a few categories. You may design objects that represent the elements of the problem domain that directly represent some aspect of the problem and are most likely to be named after the element in the problem domain.

Other objects will represent user-interface elements. This includes windows, dialog boxes, buttons, scroll bars, etc. Nowadays, most GUI (Graphical User Interface) elements are presented as objects, part of an object-oriented framework. Microsoft Foundation Classes (MFC) and AWT/Swing are examples of GUI frameworks that present GUI elements in this way.

Some objects will possibly be dedicated to management of specific tasks. This may include objects representing lower-level system elements, possibly hardware interfaces, etc. However, these may also include objects that provide other relatively low-level services.

Some objects may be dedicated to data management. These objects would include abstractions used to represent legacy data of whatever format. This group would include the objects that user to integrate these data sources into our object-oriented system.

Refining Class Selections

In designing a system, the correct abstractions must be selected. This is obvious. But, how do we know we have selected the correct

abstractions? Once we have candidate abstractions selected (before they become key abstractions), we should focus on the following questions, for each abstraction:

- Will objects of this class be created?
- How are objects of such a class created?
- What are the semantics of the class?
- Can objects of such a class be copied and/or destroyed?
- What is the behavior of objects of such a class?
- How will objects of the class be persisted?

The review of answers to questions such as these will sometimes instantly disqualify candidate classes from consideration. Indeed, if these questions cannot be answered satisfactorily, then the selection needs to be revised[38].

In answering these questions we also have to consider the following:

- Decide the "level of abstraction"
  - Have objects interact at the same or similar levels of abstraction
  - Try to be uniform
  - Employ abstraction to present a consistent object interface

- Static methods
  - Decide what should be static, what shouldn't

Viewing Your Classes from Another Perspective

Once we think we have a good set of candidate classes that exist after we've weeded out the rest, we need to evaluate our abstractions from another perspective. In performing the iterations outlined above, we're primarily looking at classes from the standpoint of the way objects of one class will interact with objects of other classes. This perspective gives us an understanding of how each class' objects will "fit" into delivering the functionality of our system.

There are however, other perspectives that we can take at this point in our design, to look at the quality of the structure of our classes and also how their objects will interact. Of necessity, this step must occur after you've gone through enough loops to make you feel somewhat comfortable with the classes in your design. If done too soon, you risk

[38] Remember, the process is iterative, so all of these questions do not have to be answered immediately.

throwing away the effort if you decide to exclude a class from your system.

This perspective will give us valuable information regarding the "quality" of our abstractions. There are things we can look for that are related to the structure and content of our classes, in addition to the interaction between objects of the classes. Lets start with structure and content.

Structure and Content Quality

In looking at the structure and content of our classes, there are a few questions we want to ask, as follows:

1. How strong is any connection from one class or object to another?
2. Are the methods defined in each class relevant to the stated behavior and semantics of the class?
3. Given the expected behavior and the semantics of the class, are there enough methods to allow this behavior to be realized?
4. Given the expected behavior and the semantics of the class, are there enough methods to give appropriate access to the data in the class and/or to make the class "complete"?
5. Are the methods named appropriately, according to some convention and/or indicative of their purpose?
6. Are the methods and data defined with the appropriate access levels given the semantics of the class?

1. How Strong is any Connection From one Class or Object to Another?

In answering this question, we're trying to identify the degree to which a class is coupled to another class. We're trying to measure the strength of association established by a connection from one class or object to another. Classes may be exhibit strong or weak coupling.

Strong Coupling

Strong coupling complicates a system, since a module will be harder to understand or modify by itself, thus adding to the overall complexity of the system. An inheritance structure is a tightly coupled structure, whereas a hierarchy based in aggregation is loosely coupled. The inheritance structure is tightly coupled as any change to a superclass is automatically propagated to all subclasses.

Weak coupling

Aggregation is weakly coupled as you are able to make changes to an individual element's class (object) that is part of the aggregation,

without affecting other elements, such as the containing abstraction itself.

2. Are the Methods Defined in Each Class Relevant to the Stated Behavior and Semantics of the Class?

This question illustrates an important query. In designing a class, we should strive to include only methods and data that are relevant to the behavior of objects of the class, and the semantics of the class. The naming of the class itself will be critical to making sure you have relevant members in the class. The name of the class implies what the class is all about, i.e. what objects of the class will be able to do. The name of the class reflects its semantics. So, to determine whether you have relevant methods and data elements in a class, you have to start with what the class represents and what its name is. You want to identify the measure of the degree of connectivity among the elements of a single class or object. Entirely unrelated elements of abstractions should not be placed together in one class. Unrelated behaviours should not be captured in the same class. A class' responsibility should be easy to determine. Any ambiguity in the interpretation of a class' responsibility should be taken as a signal that the class is implying something other than the designer intended. This will ultimately be confusing to the class' users. The responsibilities of the class should make sense overall. They should not go beyond the responsibilities implied by the abstraction. Otherwise, the true responsibilities will be obscured by the additional "noise". Thus, in a class named Cat, we shouldn't have a method called Bark().

3. Given the Expected Behavior and the Semantics of the Class, are There Enough Methods to Allow this Behavior to be Realized?

It won't make much sense for you to pick the "correct" abstractions, then "under-implement" the classes. You want to make sure that you include all methods necessary to make the object's functionality available to other objects in your solution. So, you need to review whether or not a class or module captures enough of the characteristics of the abstraction to allow meaningful and efficient interactions. The responsibility of the class should capture completely those responsibilities implied by the abstraction. Again, this also comes from the responsibility of the class. If you decided a class was to be responsible for some piece of your system's overall functionality, make sure you give it the tools (methods, data) to do so.

4. Given the Expected Behavior and the Semantics of the Class, are there Enough Methods to give Appropriate Access to the data in the Class and/or to make the Class "Complete"?

Your class definition should include enough methods to provide a "complete" interface. This includes defining public accessor methods for private data where that data needs to be obtained by other objects, as well as defining methods for all of the meaningful functionality as implied in the name of the class. You should identify whether the interface of your classes capture all of the meaningful characteristics of their abstractions. You should ensure that the responsibility of each class completely captures that implied by the abstraction.

5. Are the Methods and data Elements Named Appropriately, According to some Convention and/or Indicative of their Purpose?

As mentioned earlier, the interface of a class (object) is the only thing that will be visible from outside the class itself. So, a class should have a well-designed interface. The interface is critical as well-defined interfaces can also contribute to reuse. The methods should be clearly named, with the name describing the operation effectively and unambiguously. Different methods should have different names, except for overloaded methods). Care should be given to parameter names also.

The class definition should also include offsetting methods. This means if there is a method Get(), there should be a corresponding method Set(), unless there is a clear design reason why either should not exist.

We should also strive to have "primitive" operations. This means we should try to eliminate operations that only group other operations, unless there is a particularly good reason for doing so. There will be occasions where you need to alias functions to maintain backward compatibility, for example. However, these should be explicit decisions, not accidental ones. You should review the degree to which the operations of your classes can be efficiently implemented if only given access to the underlying representation of the abstraction.
As designers, we can utilize overloading to give clarity to our design. With overloading, we can reuse of operations' names with different signatures. This can provide easier use of the design and overloading provides flexibility to the interface.

The use of naming standards for methods and data elements can go a long way to clarifying the purpose and scope of class members, as long as there is consistency in the use of the convention.

6. Are the Methods and data Defined with the Appropriate Access Levels Given the Semantics of the Class?

We know, from earlier chapters, that we have the ability to decide what members are public, private or protected. But how do we know which member should have what access level? We know by basing our decisions on the semantics of the class and the responsibility of the class. As you look at what the functionality of classes is, you also have to look at which objects of other classes will interact with each other. This interaction will indicate which methods need to be public, as the public methods of a class (object) provide external access to the functionality of the class.

Identifying Class Relationships

Thus far, we've looked at abstraction with an eye to seeing if they are similar. If they are similar, we can define an inheritance hierarchy that leverages this similarity. We can do this because of the similarities between what they are. What do we do if there is no such similarity forthcoming? One idea is that in addition to what thing are in common, we can look at what they do in common. So, we can create relationships based on actions, not just based on structure. By looking at a set of abstractions from this perspective, we are able to identify potentially valuable relationships.

Encapsulation

With encapsulation, we can take the individual objects based on our abstractions and hide how those objects are implemented. This means we can separate the interface of an object from the implementation of that object. So, if an abstraction represents a higher level of detail, encapsulation prevents inspection of the "lower" level details of the object that is based on that abstraction, i.e. how it is implemented. This also gives the designer freedom to change or improve the implementation of an object without affecting the users of the object.

In reality, encapsulation hides how an object's operations (methods) are implemented (information hiding). All the user of the object is aware of, if the operation is part of the object's interface, is the name of the operation, what parameters it expects and what it returns. Indeed, this is all they need to be aware of at this level of detail.

We exploit encapsulation for many reasons. This is a double-edged sword. Along with the benefit of encapsulation comes the

responsibility of designing a good interface for the class. Great care should be taken when deciding which methods should be public (see above).

The issue does not stop with the public operations, however. We have to decide which elements should be made protected and private as well. We should make any operation that is never called directly from the "outside" private. If no client directly invokes the operation, it may not need to be included in the interface. This requires further clarification, as we do not want to be shortsighted. If this scenario arises, two things may be going on. If an operation is only invoked internally, then we have to decide if it should be private, or if we have a case where our interface is not primitive enough. If we are able to remove the operation that invokes the one we are looking at, without a loss of functionality that is an option we might take. This is an issue of encapsulation that has an impact on our discussion of hierarchy as well. Once an operation is private, it is not visible to any operations outside of that class. If you decide your class may become a superclass and you'd like the operations in the superclass to invoke a particular operation, make it protected instead.

Deciding which operations should be protected is also a challenge. By giving an operation this level of visibility, we are laying the foundation for the operation to be used by subclasses of the class we are designing. Given that we may be planning for a hierarchy that doesn't necessarily exist, we need to look objectively at the semantics of our abstraction. Depending on our abstraction, we need to identify those operations that may need to be modified if this abstraction was to eventually become a superclass. Keep in mind that this level of visibility makes it possible for subclasses to invoke these methods, but for all other classes, it is the same as private. There should be no conflict between operations included in the interface (public) and those you're deciding should be protected.

Hierarchical Relationships

When designing software systems, you'll notice that many objects may have much in common. They may be essentially the same, except for differences of varying degrees. Understanding the similarities and differences will allow the creation of a hierarchy of one sort or another. Creating a hierarchy simplifies programming, in as much as you are able to leverage previously designed elements and inheriting from them and enhancing their functionality. We can leverage other hierarchies such as association and aggregation.

Hierarchical relationships work well with Abstraction and Encapsulation. We can visualize and create designs using elements from the problem domain. As a by-product of Abstraction, we can also visualize and conceptualize the relationships between elements. We can leverage hierarchy in this way because of its inclusion in the object-oriented paradigm. We can use Hierarchy to represent in our abstractions, the relationships that exist in the real world.

Given a new abstraction, we must place it in the context of existing class and object hierarchies we have designed. This involves that incremental and iterative process mentioned earlier. As we progress iteratively, we might change our architecture as follows:

1. Class hierarchies become reorganized. As we examine our abstractions and their relationships, we may decide that the basis of the hierarchy should change. This may mean changing the superclass entirely, which has a cascading effect on all subclasses. We may decide to add functionality to the existing superclass, yielding a different hierarchy. Often, we will find that we have to define a different superclass, one with less functionality, i.e. a more granular abstraction. This may yield benefits as more narrowly scoped classes may improve the overall level of reuse and reliability.

2. Some classes may move up in the hierarchy. We may decide to promote a class to take advantage of the semantics of that abstraction in relation to the other classes in the tree. We may need to leverage the attributes and operations defined in that class in other classes in the tree, which would warrant promoting it. Sometimes, a class may be too general, making inheritance difficult. As we've said before, it is much better to have granular, narrowly scoped abstractions that are closely related to the real-world object they represent. If the scope is too large (the semantic gap), we may have difficulties reusing the objects and reliability may be affected.

3. We may decide to create a hierarchy where there wasn't one to facilitate leveraging polymorphism in our program. We may create a superclass purely to allow this to happen.

Persistence

Depending on the functionality of our system, the ability to save and restore the states of the objects in the system may vary from important to absolutely critical. When we talk about saving and restoring, what are we really saying. What really transpires is that to

"save" an object, we must transform the data of the object from the "in-memory" form to some format that can be persisted (i.e. stored) or transmitted somehow. To "restore" an object, we have to reconstitute the "in-memory" view of the object, so it can continue its role as part of the functionality of our system. This may take various forms. Some languages have built-in mechanisms that support object persistence[39]. The challenge exists when we are using languages like C++ that do not provide a direct, built-in way of implementing persistence.

In a nutshell, we just need to save the state of an object, in order to persist it. We can create a string-based version of the object, creating a delimited string with all of an object's values. This is one area in which XML has become popular, because we can create an XML string version of our object. This is the idea behind SOAP, the Simple Object Access Protocol, which is a defined XML format that is used to persist objects.

Another possibility is to save the values of the attributes of the objects in relational databases. I'm sure if we think further, we can find many different strategies for implementing persistence.

The other side, of course, is being able to convert from our "saved" version of the object back to our "run-time" version of the object. We have to implement methods to restore the value of the object. At the end of the restoration, we have an object that is equivalent to the object we saved.

The ability to transform an object from the run-time, in-memory version to another version, (i.e. string, etc.) is important, not just to "save" the state. Once we transform the state of an object in this way, we can communicate that new state to other systems. This means, for example, that if we can convert an object to a string, we can communicate that string to other systems or components on other platforms, in different locations. These other systems or components would then have to have the ability to restore objects from the strings, and vice-versa. Indeed, as we will see in Chapter 9, this is an important facet of distributed computing.

[39] Java is an example of such a language.

Design Goals

We utilize object-oriented software development techniques in an attempt to improve the overall quality of our developed software. This improved quality includes the following: reusability, reliability and extensibility. Let us look at how some of the concepts we've learned thus far apply to these goals.

Reusability

Object oriented development promotes software reuse. This reuse typically occurs at the component, class and object level. Many of the elements of reuse are due to the presence of hierarchy in the object-oriented paradigm.

How can we improve reusability? The discussions in this and previous chapters lead us to identify the following as some mechanisms we can use:

1. **Modularity**

When we make something modular, we are creating a sub-program, i.e. one or more classes. This means, if designed properly, we could reuse this sub-program over and over again. An example of modularity employed this way is a .dll in Windows. Many different programs can utilize the same .dll, which can simplify the development process.

Modularity on a grander scale comes into play when we include distributed systems in our discussion. Distributed Systems are discussed in Chapter 9.

2. Classes and Objects

The very idea of encapsulating a certain amount on functionality in a class facilitates reuse, if (and only if) the design of the class is correct. This is one of the benefits you can realize when you take care in the choice and quality of your abstractions (see above). The better the quality of the abstraction, with regard to the semantics of the class and the selection of methods, etc., the more likely it will be suitable for reuse. Big, bloated classes that do too much, i.e. have too much functionality, will not be as easy to reuse. The tighter scoped, better defined classes will be easier to reuse by far.

3. Inheritance, Aggregation and Composition

With inheritance, we can directly leverage previously developed code. This goes way beyond the "cut and paste" technique. Say a function is created which copied code in a second function. We now have two independent functions, even though they share code. Each of these functions needs to be separately maintained. If we fix a bug in the original function, this fix is not propagated to the second function, automatically or otherwise. This is not a good example of code reuse. If we look at inheritance however, we see that we can directly leverage previously developed code in such a way that any changes to the originally developed code (class) is automatically available to new code (subclasses).

We can also make any class a superclass. This means we do not have to redefine functionality completely. We can merely add to it, by creating a subclass that has the new elements defined. It is critical then, that our classes, in general, are well defined, so that if we need to make them the root in an inheritance chain, their design makes them suitable to do so.

If we look at aggregation and composition, we see other possibilities for reuse. With these two elements of hierarchy, we can assemble new objects from combinations of previously existing ones. This allows us to create new loosely coupled objects, whereas inheritance allows us to create new tightly coupled objects.

Reliability and Robustness

To describe software as "reliable", it must be robust in how it handles exceptional situations. An exception is a potentially severe error condition. Reliable systems must be able to run exception handlers to "handle" exceptional situations and recover from them. Many object-oriented programming languages provide exception detection and handling, greatly aiding the development of reliable software. Even though many languages now support exception handling, using these

techniques is not mandatory. It is not enforced directly by the programming language. Of course, we could let exceptions go un-handled, thereby causing our systems to crash.

What else can we do to improve the reliability and robustness of a system by defensively trying to avoid exceptional situations? Here are some things we can do:

- Reduce potential for problems by using "privates" for data, and controlling access to data via methods
- Control object creation via constructors, i.e. based on semantics and using copy constructors

Extensibility

The key to extensibility is to make sure the design is done in a "granular" way. By granular, we mean a design wherein each class' scope is narrowly defined, closely matching the scope of the element it abstracts. This means selecting the correct abstractions and making sure each is cohesive. It also means creating individual classes, the result of the object-oriented decomposition, that are at the correct level of abstraction. We should guard against creating classes based on abstractions with scopes that are too large in the context of the problem. The drawback of having abstractions that are too large in scope is that relatively minor changes could potentially become major system changes. If a class' scope is too large, we potentially lose the ability to create elegant hierarchies. It is far better to create small, tightly scoped classes that are building blocks, than large monolithic objects which are not only less extensible, but have less reusability as well.

Additional Design Factors

In order for our design to be successful, there are other factors we have to consider. These include accounting for non-functional requirements and employing design patterns.

Non-Functional Requirements

We have to look at more than "just" the abstractions, etc. based on the functional requirements of the system. We also have to include "non-functional" requirements. "Non-functional" requirements include items such as environmental requirements. For instance, these may be items such as data center requirements that must be met before an application is allowed to be in production in that environment. Maybe there are special classes and/or modules that need to be included to allow remote monitoring of the application, logging, etc. These are not items that would be communicated by the user as part of the

functional requirements, but they obviously need to be included in the design of the solution.

Modeling

As we mentioned before, modeling allows us to present a picture, i.e. a visual representation of our system. However, there are added benefits of creating and using models during the design process, not just in creating artifacts at the end. We can use modeling to improve the design process in the following ways:

Present and validate architecture

With class diagrams, it is easier to see how elements of our system interact with each other, i.e. the relationships classes have with each other. We can quickly identify from the diagrams what relationships are in place and more importantly, whether or not these relationships are correct. This allows us to verify the overall architecture, i.e. static structure of our system.

Model object interaction and behavior at runtime

With object interaction diagrams such as Sequence and Collaboration diagrams, we can model how objects will behave at runtime, as they collaborate to provide functionality. It is easy to see and verify that the objects are providing the functionality, as expected.

To shine even more light on an individual object, we can use State diagrams to examine individual objects greater detail.

Design Patterns

We utilize design patterns all the time. A design pattern is a generalized sequence of steps to be used to solve a commonly occurring problem.

Typically, patterns do not extend to specific code for a particular solution. Instead, it presents a solution in a more general form. When we utilize patterns, we must modify the pattern to fit our particular problem. This may require us to supply details that may have been missing from the original pattern.

As we gain experience and knowledge, the patterns that come from the solutions we've developed are at our disposal. We can use these in solving new problems. The presence of patterns and the ability to use them make the solution development easier.

Patterns exist at all levels. There are large, architectural patterns such as component-based development models, n-tiered models

(generalization of the client-server model), layered models, etc. On a smaller scale, patterns may represent particular programming constructs.

For object-oriented development, patterns involve classes and objects. The classes in the pattern represent the elements of the problem the pattern is a solution for. The pattern captured the relationships among the classes. These relationships are based on aggregation, inheritance and association and are all in the context of the problem to be solved.

There are many design patterns available for us to apply to problems. Some of the more popular patterns are listed below:

Containers

A container is a design pattern that describes a class that manages the collection of other objects. The interface of the class provides the expected behavior with operations to provide a count of objects, append, delete (all or a specific object). Containers also support the notion of GetFirst() and GetNext(), to deliver the objects of the container in some predetermined order.

Containers are very useful abstractions to include in a design. They may be used anywhere a collection of objects (not necessarily all from related classes) is required. The container's operations allow the easy inspection of the contents.

Wrappers

In Chapter 1, we presented the notion of using an abstraction to represent legacy systems[40] with which object-oriented systems interact[41]. This technique may be expressed as the Wrapper pattern. In the Wrapper pattern, we define a set of public operations that represent the operations of the legacy system. These operations are the interface of the abstraction, the implementation being the legacy system itself. Of course, with Encapsulation, this implementation detail is hidden. The operations of the abstractions would be implemented by making function calls to the legacy system. This pattern is flexible enough to support object-relational interactions.

Object Factories

The Container pattern above describes a class that manages a set of objects. If we extend this functionality to include the responsibility of

[40] Non-object-oriented systems in general.
[41] Discussed in detail later in this chapter.

creating objects, we have the Object Factory pattern. The Object Factory would create instance of specific classes, and in some cases, may be extended to include management and tracking of the created objects.

Model-View-Controller

The Model-View-Controller (MVC) pattern was introduced in the late 1980's and has become popular as a pattern for Graphical User Interfaces (GUI's). The pattern separates user input, presentation and data into separate parts. Controllers are responsible for the collection of user inputs. The Model contains the core functionality and data of the application. The View represents the various views or presentation of the data (in the Model). Effectively, MVC separates data from presentation, which fits well with client-server architectures.

Design Elements

There are various "design elements" that we'll mention in this section. The term "design element" is being used to describe various structures, options and techniques that we have at our disposal.

Strategic vs. Tactical Decisions

A strategic decision is one with sweeping architectural implementations –i.e. one with large consequences across the design. Strategic decisions affect the long-term view of your system. Strategic decisions are based on a full understanding of the requirements, an understanding of the environment in which the system will operate and expectations about the evolution of the system[42]. At its most basic, strategic decisions will affect which abstractions become key abstractions. By thinking strategically, we will try to find a "best fit", among all of the attributes such as reuse, reliability, etc. Our strategic view will be helped by having properly scoped abstractions. Depending on your overall strategy[43], changes to a class' behavior (i.e. functionality), error-handling mechanisms (exception handling), etc. are examples of strategic decisions.

A tactical decision is one that is more localized, i.e. has fewer consequences. "One-off" decisions or decisions about a particular algorithm to implement a method are tactical decisions. Earlier, we said that encapsulation hid the implementation and we could safely implement a specific algorithm for a sorting operation, away from the

[42] Many other factors affect strategic decisions. The platform(s) for deployment, databases, planned upgrades/new features. There are usually a host of environmental issues that must be considered also.
[43] It is important that there is one that governs the development effort.

view from outside. The selection of a particular sorting operation could be a tactical decision, as its effects may be localized.

Additional Considerations

When designing a system, there are many additional considerations that we must take into account.

Designing for Interoperability

Interoperability is the ability of systems to communicate, i.e. to interoperate. As requirements become more complicated, we have the need to leverage data owned by other systems. These other systems may be databases, mainframe systems, client-server systems, object-oriented systems, web systems, etc. If we are designing an object-oriented system, how do we account for these systems, especially the non-object-oriented? Strictly speaking, there is no "simple" solution. There are however, certain "patterns" that we can follow[44]. The approach selected depends on which environmental issues are to be addressed.

Relational Databases

One scenario might be to communicate with relational databases[45]. This may be because there is important data already stored in existing databases, on various platforms. There are a few approaches we can take to integrate this data into our system. We may create an abstraction to represent the entire data store. This means, there would be one abstraction representing a database. An alternate approach is to include operations in exiting abstractions that represent the operations on the database[46].

If we choose the approach of one abstraction representing the database, we have to decide how to implement the operations of this class, given the scope of the abstraction. We have to be careful about which operations we include in our interface. Obviously, if we interact directly with the database, we have the full capability of SQL at our disposal. This would not be the most elegant solution. In this case, the operations would be highly dependent on SQL. Though it seems easier to implement such a "pass through" interface, it may limit flexibility, as we'll see.

[44] See previous discussion of Design Patterns in this chapter.
[45] Non object-oriented.
[46] We are ignoring details such as libraries and infrastructure used to communicate with the database management system. For the purpose of our discussion, we are assuming these exist. These issues are at a lower level of abstraction than our discussion.

We could, instead, determine which operations on the data are required. For example, we can determine the necessary queries, updates, deletions, insertions. If the system is of even marginal complexity, there may be many of these database operations that are required. We would have to determine how to present these operations in the interface of our class. A particular business operation might include a combination of these operations, in addition to supplying parameters to the operations. The business operations, which may or may not correspond directly to each of the database operations, may define the public operations in our abstraction. Indeed, some combination of these would comprise the interface of our class.

If we construct the abstraction as immediately above, we have a few advantages. For example, all of the details of the database are ultimately hidden as the implementation details of the class. This means, the database could be replaced or changed radically, with no impact on the other classes, unless the interface(s) are changed as a result.

What about attributes and private or public operations (for either case)? The need for these would be determined based on what the complete functionality of the class is.

Legacy Systems

We can employ a similar strategy to integrate relational data into object-oriented systems. In general, we may create abstractions that represent the other systems. The operations that we define on the abstraction would correspond to the operations that we need to execute in the other system.

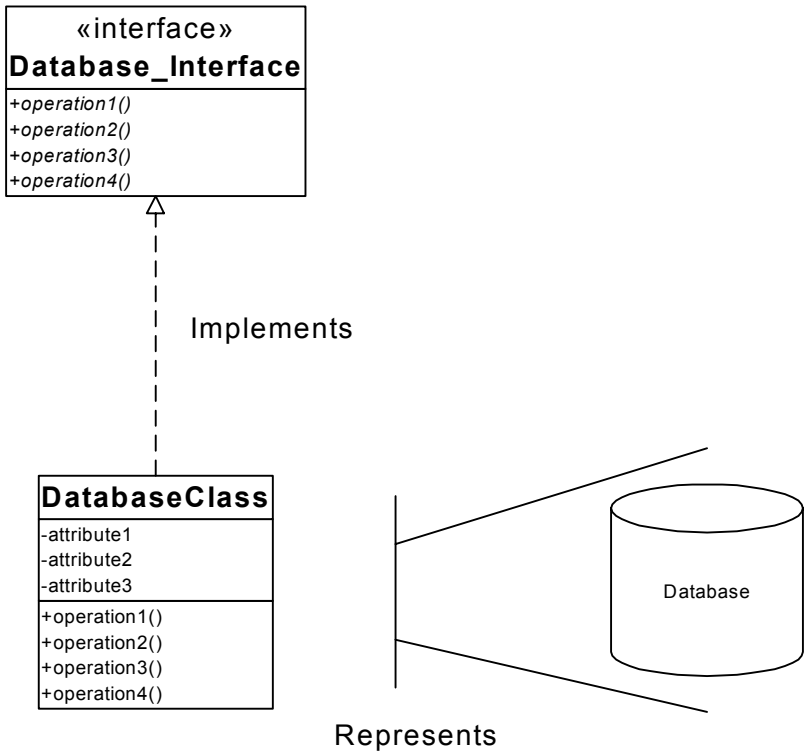


Fig 6.1 Database Abstraction

In practical terms, there are many issues to be addressed when we talk of integrating disparate systems. We have to consider factors such as what platforms the systems are on, what infrastructure exists between systems, i.e. communication, protocols, etc. What we need is a set of services that would allow us to communicate effectively between systems, across processes and platforms. Effectively, these services allow us to distribute the overall processing across many platforms, even though these platforms may be geographically distant. As a result, these services must be robust and able to handle the communication that needs to happen across platforms as different as mainframes and PC's. In Chapter 8, we will investigate this in our discussion of Components.

Language Features

Many languages provide "pre-made" constructs (classes) that we can use in our design phase. Some of these are direct construct of the language. Others are provided as library routines, separately. Some of these classes are listed below.

String Classes

Some object-oriented languages provide string classes. These classes allow string manipulation functions including tokenization[47], searching, replacing, etc. Having these utility classes "pre-made" frees us from having to create these "from scratch".

Collection (container) Classes

Many languages provide "off-the-shelf" collection classes that we can use in our design. Collection classes typically manage their own memory. These collection classes include the following.

Generic Collection Classes

These classes support un-ordered grouping of other objects. The collection classes (and their objects) allow us to aggregate groups of objects, not necessarily all from the same class or inheritance structure. Collection classes such as these typically provide methods to add to the collection, delete from the collection, provide a current count, etc. These operations would be part of the interface of the collection object.

Sets

A set is a specific type of collection where the contents of the class are ordered. Typically, no duplicates are allowed and in some cases, sets are implemented such that there can only be one null value.

Lists

List objects hold an ordered collection of objects as well. However, a list is typically sequentially accessed. This means you can only go from one object in the list to another, the next in the order. Some lists also support moving backward, the previous in the order.

Maps

Maps are collection objects which maintain a <key,value> pair in the object. This means each object in the collection is associated with a key. This key is used to retrieve the value from the map. No objects are added to the map without a corresponding key.

Operator overloading

Earlier, we discussed function (method) overloading. Operator overloading is similar in concept. Think of an operator as a function with an implicit operand (parameter) and zero or more explicit operands. Then overloading an operator becomes the same as overloading a function. We are changing the parameter list. Why

[47] A string is tokenized when it is broken up sequentially into tokens (groups of characters), separated by delimiters.

would we overload operators? To more closely adhere with the semantics of our abstractions. Say we have a class that needs to support the logical addition and subtraction operations, based on the real-world objects it represents. We could define operations Add() and Subtract() in the class to provide this functionality. Our other alternative, if operator overloading is supported, is to define overloaded operators that are the functional equivalents to the Add() and Subtract() functions. Overloading the operators, if available, does not add new functionality. Instead, it contributes, sometimes greatly, to the overall clarity and readability of the code.

Parameterized Classes

Depending on the language or environment being used, some of the collection classes may be implemented as parameterized classes. Parameterized classes, if supported by a particular programming language, can be important tools for us as designers. In a parameterized class, the fundamental types used in implementing the class are passed in as parameters when creating an instance of the class. So, if we wanted to create two lists, each of which would hold completely unrelated objects, we could create one parameterized class. We would then create instances, each with the appropriate type.

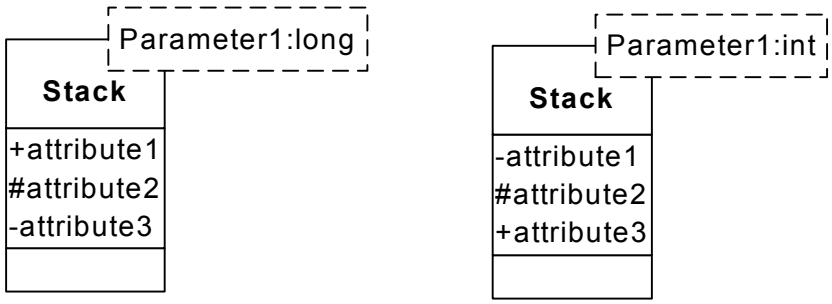


Fig 6.2 Parameterized class

Sample Project

Let us apply what we've learned so far to our example. As we evolve our design, there are some questions that now need to be answered. The way we answer these questions will obviously influence our design. We may also discover that our requirements do not completely specify certain areas. In this case, we may have to make some assumptions. Our journey continues below:

1. Adding a new student's information: Where should the method or methods corresponding to this belong? Are the "student" classes responsible for this, or is the System class responsible for this? What does it mean to have the "student" classes responsible for this activity? That means an object of the correct type of student would have to be instantiated and initialized properly. Let us think of semantics also. Is it reasonable to expect a student object to know how to add itself to the system? Also, consider the sequence of events. Something (i.e. the All Students object) would have the instance of the student object created then the student object would know how to add itself to the list of other student objects maintained by the system. Doesn't seem reasonable. Instead, it appears that the method to add a new student to the system should have the correct student object instantiated and initialized (probably via the correct constructor), and then add the student to the list managed by the All Students object, all within the System object. So let's call the method "AddNewStudent()" and add it to the All Students class. Let us replace our "SetStudents()" accessor method with this method.

2. Searching and displaying a student's information: This is an operation of the entire list of students, held within the System object. As a result, this method, which we will call "SearchStudents()", will be inside the All Students class also.

3. Deleting a student: As before, this is an operation on the entire list of students. Let's call this method "DeleteStudent()" and include it in the All Students class.

4. Changing/assigning classes and credits to students: Let us examine what we have to do to change student information. We have to locate the particular student object by some means then access the information in that object to change it. As the list is maintained in the System object the method or methods to change student information should be there also. Let's assume we have one method: ChangeStudentClassesAndCredits(), which we will add to the All Students class. This method will access the access methods of the student object to indirectly manipulate the private data.

5. Changing/assigning a student's major: This is similar to the methods we've defined immediately prior to this. Using the

same logic, let us call this method ChangeStudentMajor() and add it to the All Students class.

6. Changing/assigning a student's type: This method will cause a student's type to change from Typical to Faculty or Transfer, etc. This involves creating and deleting objects, i.e. deleting an existing object and creating a replacement. This should also be in the All Students class. Let us call this method ChangeStudentType().

7. Changing/assigning a student's status, i.e. full-time or part-time according to the rules above: Same reasoning as above. Let's call the method ChangeStudentStatus() and add it to the All Students class.

8. Producing reports: We need to provide a total of seven (7) reports. Let's make each report be generated by a method, named as follows:
   SortedFullTimeStudents()
   SortedPartTimeStudents()
   NumberofStudentsOfEachType()
   SortedNamesAndAddresses()
   ReversedNamesAndAddresses()
   StudentsMajorsAndCredits()
   StudentsCostForTheSemester()

So, let's list our classes again, including all of our methods:

Class TypicalStudent
Attributes:
Student Name (Object of class Name)
Student Address (Object of class Address)
ID
Majors (Object of class Student Majors)
Subjects (Object of class Student Subjects)
Grade
Discount

Methods: (All public)
GetStudentName()
SetStudentName()
GetStudentAddress()
SetStudentAddress()
GetStudentID()

SetStudentID()
GetStudentMajors()
SetStudentMajors()
GetStudentSubjects()
SetStudentSubjects()
GetStudentGrade()
SetStudentGrade()
GetStudentDiscount()
SetStudentDiscount()

Class FacultyStudent
Attributes:
All attributes of TypicalStudent
Subject Taught
Date Employed (Start of employment - used to calculate length of service)

Methods:
All methods of TypicalStudent
GetSubjectTaught()
SetSubjectTaught()
GetDateEmployed()
SetDateEmployed()

Class TransferStudent
Attributes:
All attributes of TypicalStudent
Home college (Object of class College)

Methods
All methods of TypicalStudent
GetHomeCollege()
SetHomeCollege()

Class HomeCollege
Attributes:
College Name
College Address (Object of class Address)

Method:
GetCollegeName()
SetCollegeName()

Class Name

Attributes:
First Name
Middle Initial
Last Name

Methods:
GetFirstName()
SetFirstName()
GetMiddleInitial()
SetMiddleInitial()
GetLastName()
SetLastName()

Class Address
Attributes:
Street Address
City
State
Zip

Methods:
GetStreetAddress()
Set StreetAddress()
GetCity()
SetCity()
GetState()
SetState()
GetZip()
SetZip()

Class Subject
Attributes:
Subject Name
Credits

Methods:
GetSubjectName()
SetSubjectName()
GetCredits()
SetCredits()

Class Major
Attributes:
Name_of_Major

Methods:
GetMajor()
SetMajor()

Class System
Attributes:
Majors (Object of class All Majors)
Subjects (Object of class All Subjects)
All Students (List of objects representing all types of students)

Methods:
GetMajors()
SetMajors()
GetSubjects()
SetSubjects()
GetStudents()
SortedFullTimeStudents()
SortedPartTimeStudents()
NumberofStudentsOfEachType()
SortedNamesAndAddresses()
ReversedNamesAndAddresses()
StudentsMajorsAndCredits()
StudentsCostForTheSemester()

Class StudentMajors
Attributes:
Majors

Methods:
GetMajors()
SetMajors()

Class StudentSubjects
Attributes:
Subjects

Methods;
GetSubjects()
SetSubjects()

Class AllMajors
Attributes:

Majors

Methods:
GetMajors()
SetMajors()

Class AllSubjects
Attributes:
Subjects

Methods:
GetSubjects()
SetSubjects()

Class AllStudents
Attributes:
Students

Methods:
AddNewStudent()
SearchStudent()
DeleteStudent()
ChangeStudentClassesAndCredits()
ChangeStudentMajor()
ChangeStudentType()
ChangeStudentStatus()

There are other methods to consider. Let us explicitly discuss the constructors and destructors for each of our classes.

Constructors

There are three "student-related" classes, Typical Student, Faculty Student and Transfer Student.

The TypicalStudent class represents all typical students. It is also, by design, the base class of our student hierarchy. So, our constructor will have to initialize the attributes of TypicalStudent. How do we initialize objects of the TypicalStudent class? If an object of this class represents an individual student then which attributes are "must haves"? In this context, it would not make sense to have a TypicalStudent object with no name and address. Neither would it make sense to have a TypicalStudent object without an ID. However, we could conceivably have a student object without a list of majors or

subjects, a grade or a discount. These could be added later, and in fact, the ability to modify these is part of the overall requirements. We can use constructors to assist here. Also, the requirements tell us we need to provide for changes in type, i.e. changing from TypicalStudents to Faculty/Transfer and vice versa. We can use a constructor to assist here as well. We can define a constructor that will allow us to create an object of TypicalStudent from an object of Faculty or Transfer. So we could have three constructors, as follows:
TypicalStudent(name, address, ID)
TypicalStudent(FacultyStudent)
TypicalStudent(TransferStudent)

Why are we using constructors to convert between base class and subclass objects? The decision is based on the need to have an object that represents each type of student, in keeping with our design thus far. We could access FacultyStudent and TransferStudent objects from a reference (or pointer) to a TypicalStudent object, but the underlying object would not be a TypicalStudent object. This is one case where we're not trying to exploit inheritance and polymorphism.

For FacultyStudent and TransferStudent objects, the ideas above hold also. As such, for each, we would need a constructor that only allows objects of that class to be created with a name, address and ID. As well, we will include constructors for FacultyStudent and TransferStudent that would allow for conversions as well.

We then have the following:
FacultyStudent(name, address, ID)
FacultyStudent(TypicalStudent)
FacultyStudent(TransferStudent)

TransferStudent(name, address, ID)
TransferStudent(FacultyStudent)
TransferStudent(TypicalStudent)

We may also define other constructors as necessary. For example, we may have a need to define copy constructors, i.e. constructors that take an object of the same class and make a copy of the attributes. If we need any of these later on, we'll add them.

HomeCollege
What should our constructor for the College class be defined as? The College class has an attribute for name and an attribute for address. If we decide that an object of the college class should not be created

without the name and address of the home college, then we need to have a constructor that does this. So we would have the following:
Home College(name, address)

We do not have a requirement for conversions or copies, etc. So we'll keep this as our only constructor.

Name
For the Name class, we should not have an object created without the First and Last names. Not all people have a middle initial. So, it seems we need two constructors, as follows:
Name(first, last)
Name(first, middleinitial, last)

Address
For the Address class we should not have an object created without the Street, City, State and Zip fields. This obviously is a US-centric model. In any case, our constructor can ensure this, as follows:
Address(Street, City, State, Zip)

Subject
We should create objects of the Subject class without a Name and Credits. We're assuming each subject available has an explicit number of credits associated, greater than or equal to zero. So, we use our constructor as follows:
Subject(Name, Credits)

Major
We should create objects of the Major class without a name. So, we use our constructor as follows:
Major(Name_of_Major)

System
The System class is a very critical class in our design. As a result, the constructor for our system class is important as well. Let's examine this constructor. So far, we've decided that the system class will manage the lists of subjects, majors and students. In addition, the "reporting" functionality will be in the System class. How then, do we employ a constructor to properly create an object of the system class? Our constructor needs to properly initialize all of the lists managed by that System object. The constructor only has to initialize the list to be empty. What about parameters? There aren't any that we would identify at this point. So, it seems we need to implement the following constructor:
System()

StudentMajors
This class represents the list of majors for a particular student. The constructor only has to initialize the list to be empty. There are no parameters. So, we need the following constructor:
StudentMajors()

StudentSubjects
This class represents the list of subjects for a particular student. The constructor only has to initialize the list to be empty. There are no parameters. So, we need the following constructor:
StudentSubjects()

AllMajors
This class represents the list of all majors available to any student. The constructor only has to initialize the list to be empty. There are no parameters. So, we need the following constructor:
AllMajors()

AllSubjects
This class represents the list of all subjects available to any student. The constructor only has to initialize the list to be empty. There are no parameters. So, we need the following constructor:
AllSubjects()

AllStudents
This class represents the list of all students. The constructor only has to initialize the list to be empty. There are no parameters. So, we need the following constructor:
AllStudents()

Destructors

We need to consider how we will "clean up after ourselves". What does this mean? This is more related to implementation. Depending on our choice of implementation for our lists, i.e. static or dynamic, we may need to explicitly define methods (i.e. destructors) that deallocate the memory allocated for each object on our list. Again, this may be a very good idea for each class that manages a list: System, All Subjects, All Majors, Student Subjects and Student Majors.

Checkpoint

So, our list of classes, attributes and methods are now:

Class TypicalStudent
Attributes:
Student Name (Object of class Name)
Student Address (Object of class Address)
ID
Majors (Object of class StudentMajors)
Subjects (Object of class StudentSubjects)
Grade
Discount
Methods: (All public)
TypicalStudent(name, address, ID)
TypicalStudent(FacultyStudent)
TypicalStudent(TransferStudent)
GetStudentName()
SetStudentName()
GetStudentAddress()
SetStudentAddress()
GetStudentID()
SetStudentID()
GetStudentMajors()
SetStudentMajors()
GetStudentSubjects()
SetStudentSubjects()
GetStudentGrade()
SetStudentGrade()

GetStudentDiscount()
SetStudentDiscount()

Class FacultyStudent
Attributes:
All attributes of TypicalStudent
Subject Taught
Date Employed (Start of employment - used to calculate length of service)

Methods:
All methods of TypicalStudent
FacultyStudent(name, address, ID)
FacultyStudent(TypicalStudent)
FacultyStudent(TransferStudent)
GetSubjectTaught()
SetSubjectTaught()
GetDateEmployed()
SetDateEmployed()

Class TransferStudent
Attributes:
All attributes of TypicalStudent
Home college (Object of class HomeCollege)

Methods
All methods of TypicalStudent
TransferStudent(name, address, ID)
TransferStudent(FacultyStudent)
TransferStudent(TypicalStudent)
GetHomeCollege()
SetHomeCollege()

Class HomeCollege
Attributes:
College Name
College Address (Object of class Address)

Method:
HomeCollege(name, address)
GetCollegeName()
SetCollegeName()

Class Name

Attributes:
First Name
Middle Initial
Last Name

Methods:
Name(first, last)
Name(first, middleinitial, last)
GetFirstName()
SetFirstName()
GetMiddleInitial()
SetMiddleInitial()
GetLastName()
SetLastName()

Class Address
Attributes:
Street Address
City
State
Zip

Methods:
Address(Street, City, State, Zip)
GetStreetAddress()
Set StreetAddress()
GetCity()
SetCity()
GetState()
SetState()
GetZip()
SetZip()

Class Subject
Attributes:
Subject Name
Credits
Subject Grade

Methods:
Subject(Name, Credits)
GetSubjectName()
SetSubjectName()
GetCredits()

SetCredits()

Class Major
Attributes:
Name_of_Major

Methods:
GetMajor()
SetMajor()

Class System
Attributes:
Majors (Object of class All Majors)
Subjects (Object of class All Subjects)
All Students (List of objects representing all types of students)

Methods:
System()
GetMajors()
SetMajors()
GetSubjects()
SetSubjects()
GetStudents()
SortedFullTimeStudents()
SortedPartTimeStudents()
NumberofStudentsOfEachType()
SortedNamesAndAddresses()
ReversedNamesAndAddresses()
StudentsMajorsAndCredits()
StudentsCostForTheSemester()

Class Student Majors
Attributes:
Majors

Methods:
StudentMajors()
GetMajors()
SetMajors()

Class Student Subjects
Attributes:
Subjects

Methods:
StudentSubjects()
GetSubjects()
SetSubjects()

Class AllMajors
Attributes:
Majors

Methods:
AllMajors()
GetMajors()
SetMajors()

Class AllSubjects
Attributes:
Subjects

Methods:
AllSubjects()
GetSubjects()
SetSubjects()

Class AllStudents
Attributes:
Students

Methods:
AllStudents()
AddNewStudent()
SearchStudent()
DeleteStudent()
ChangeStudentClassesAndCredits()
ChangeStudentMajor()
ChangeStudentType()
ChangeStudentStatus()

Let's look at our class diagrams. We will only depict the attributes of each class, as well as the hierarchical (i.e. inheritance and composition) relationships between classes.

Part A:

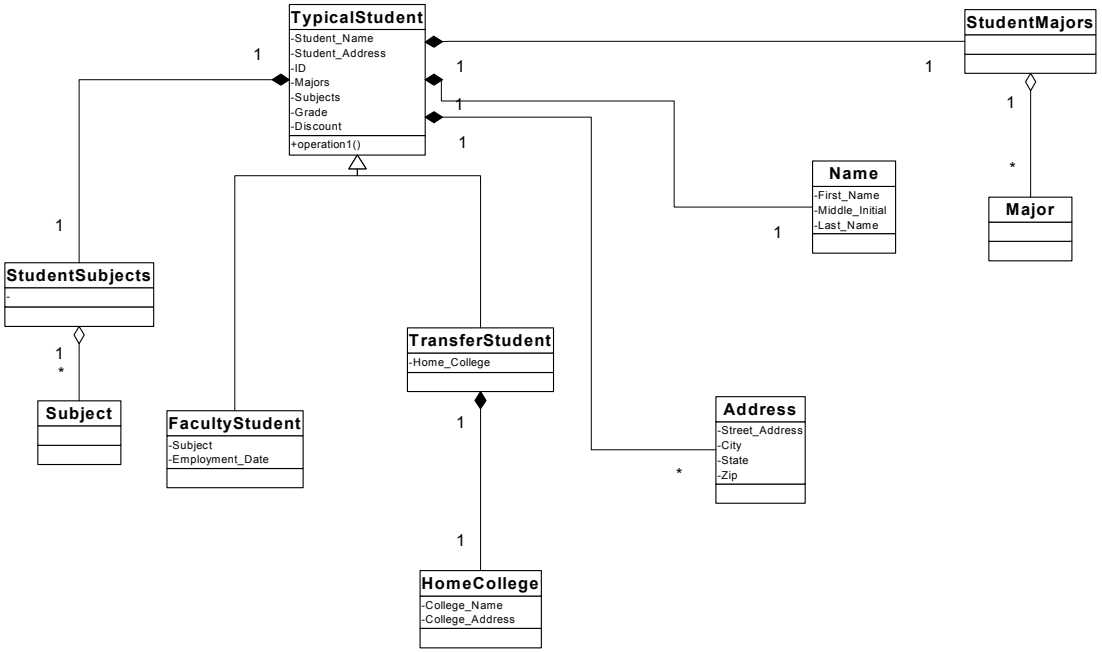


Fig 6.1 Student class relationships

Part A depicts the inheritance relationship between the "student" classes. In addition, we can see the associative relationships. Due to the inheritance relationship between the sub-classes and base class, the associative relationships that involve the base class are inherited by the sub-classes as well. Some of the relationships are 1-1, i.e. name, address, etc. Others are 1-many, i.e. between the class managing a list of subjects and individual subjects.

Part B:

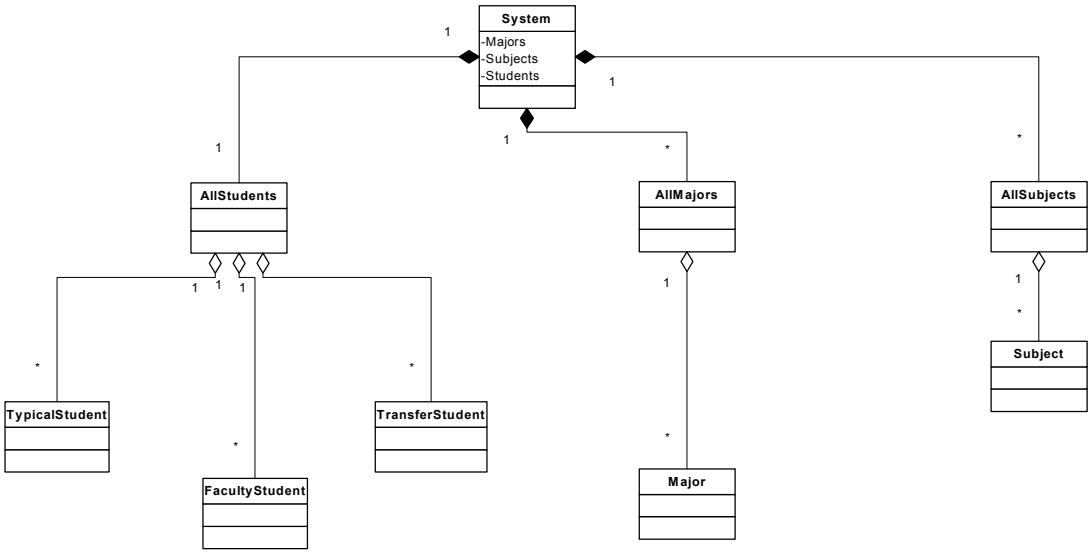


Fig 6.2 System class relationships

Part B depicts the relationships involving the system class. Here also, we can see the associative relationships. Again, some of the relationships are 1-1, while others are 1-many.

Persistence and Data Management

So far, we have not spent any time with regard to persistence or data management. Let's look at these separately.

We know from the initial requirements that the system is expected to maintain student data in a relational database. From our analysis, we understand what student data needs to be maintained. If we take a linear view of the data, we have the following

For class TypicalStudent:
Student Name (object of class Name)
Student Address (object of class Address)
ID
Majors (object of class Majors - aggregate)
Subjects (object of class Subjects - aggregate)
Overall Grade
Discount

For a FacultyStudent:
All attributes of TypicalStudent
Subjects Taught (object of class Subjects)

Date Employed

For TransferStudent:
All attributes of TypicalStudent
Home college (Object of class College)

For HomeCollege:
College Name
College Address (Object of class Address)

For class Name:
First Name
Middle Initial
Last Name

For class Address:
Street Address
City
State
Zip

For class Major:
Name of Major

For class Subject:
Subject Name
Credits
Subject Grade

Given that our available database is relational, we have to map the data defined in our classes (multi-dimensional) into a 2-dimensional model. This will give us some insight into the tradeoffs and compromises we have to make in the real world.

Our unique identifier is the student ID. No two students will have the same student ID. For simplicity, let's define the ID as a number, though it could just as equally have been a character string. We can then organize the data relevant for a student with the key being the student ID.

A database schema[48] may be designed that identifies a table for each of the abstractions listed above, with a few notable exceptions. Our list is as follows:

Table TypicalStudent (corresponds to class TypicalStudent):
ID
First Name
Middle Initial
Overall Grade
Discount

Table FacultyStudent(corresponds to class FacultyStudent):
ID
Date employed

Table TransferStudent(corresponds to class TransferStudent):
ID
Home College Name
Home College Street Address
Home College City
Home College State
Home College Zip

Table StudentAddress(corresponds to class Address):
ID
Street Address
City
State
Zip

Table StudentMajor(corresponds to class StudentMajor):
ID
Name of Major

Table StudentSubject(corresponds to class StudentSubject):
ID
Subject Name
Credits
Subject Grade

You'll notice that we do not have a separate table for Name. We've included the attributes of the Name class in the TypicalStudent table.

[48] A database layout.

While this does simplify our database schema, it limits association between name and ID to 1:1. So, students are limited to using one name only[49]. If this was deemed insufficient, we could add another table. In practical terms, this would be unlikely.

We do not have a table for HomeCollege either. How is this possible? As was done for Name, the data from the HomeCollege class has been included in the TransferStudent table. The overriding assumption is that we keep track of the last college the student transferred from only.

Each of our tables will have primary keys defined. The list of keys is below:

Table TypicalStudent:
ID

Table FacultyStudent:
ID

Table TransferStudent:
ID

Table StudentAddress:
ID
Street Address

Table StudentMajor:
ID
Name of Major

Table StudentSubject:
ID
Subject Name

Earlier, we defined the student ID as the unique identifier to be used for students. Since it is unique, it is the only value required as a key for TypicalStudent, FacultyStudent and TransferStudent. For StudentAddress, we add the value of the street address to the key. The assumption here is that a student may have different addresses, i.e. home, dorm, etc. For StudentMajor, the key includes the ID and the name of the major. Likewise, for StudentSubject, the key includes

[49] It is a good assumption that students would not give multiple aliases at time of registration.

the ID and the name of the subject. We use a multi-valued key in each table where we might have multiple rows existing with the same ID, as is possible in StudentMajors, StudentSubjects and StudentAddresses.

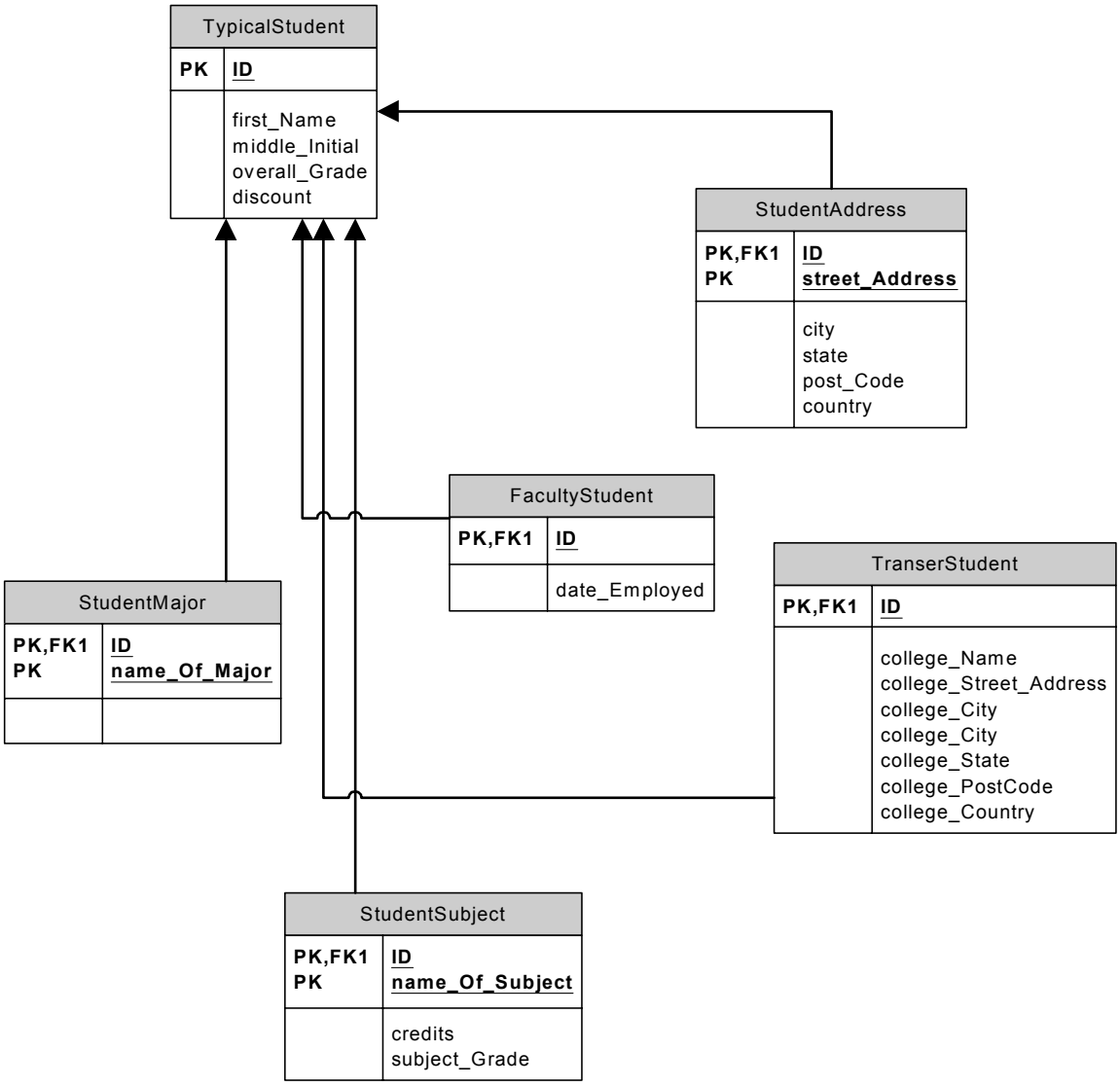


Fig 6.3 Entity-Relationship diagram for student data

The diagram above expresses the relationships that exist between tables.

There are other data values that we need to store in our database. We need to store the complete lists of classes and majors, in addition to keeping information to be used in calculating discounts and student costs. These additional tables are outlined in the diagram below.

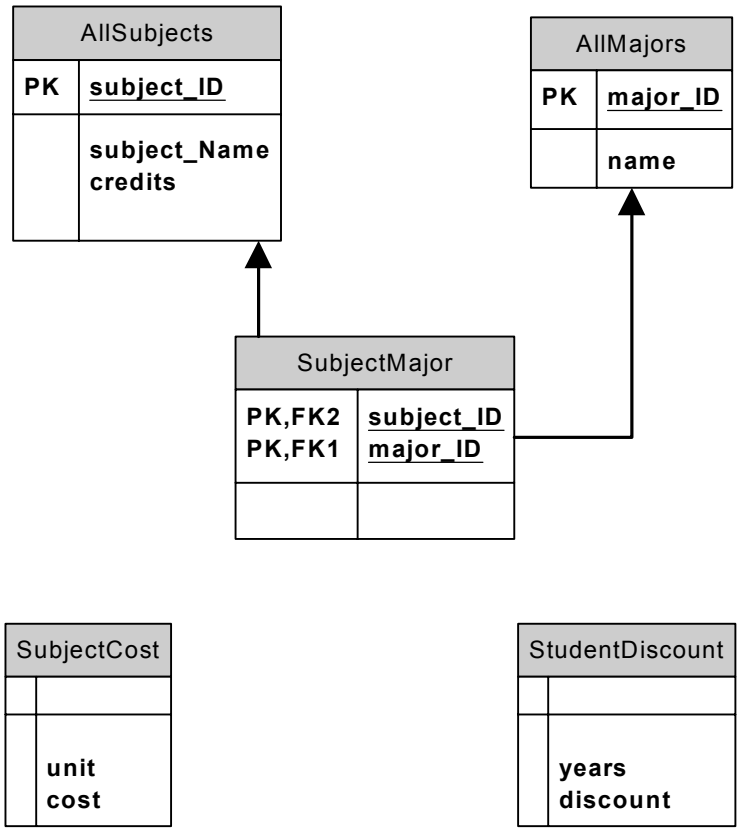


Fig 6.4 Additional entities

Given this schema, how do we implement functionality that we can use to save to and restore from our relational database? We made some trade-offs in an attempt to "flatten" the data so it would fit into a relational model. We now need to build a bridge between our relational model and our object model. In practical terms, this means we need to implement methods to transform object-based data into relational form and vice-versa.

Student-Related Data

If we want to store the state of a student, we need to save the states of the various objects that are cooperating to give us our overall view of a student and their related information. This means each class will have to have methods to save to and restore from our relational database. Let's look at this in more detail, reviewing each class' needs individually. Our goal is to understand where to implement our methods. We will call the method to save data Save() and the method to restore data, Restore()[50].

[50] See Chapter 2.

As complicated as this might sound, in this case it is actually quite straightforward as we have effectively constructed a 1:1 correspondence between entities and classes[51]. However, we have to figure out how to create Name and HomeCollege objects. In addition, we have to decide what kind of abstraction should represent the database in our design.

TypicalStudent

Saving Data

This class needs to store data in the TypicalStudent table. There is a direct correlation between this data and the data in the TypicalStudent class. However, as we mentioned before, we included information that we would expect to find in the name and address attributes as well. So, when we save an object of this class, we are indeed saving the objects of classes Name, Address, Student Majors, and StudentSubjects() that are embedded in any object of TypicalStudent, in addition to saving the other attributes in the TypicalStudent class. Saving a TypicalStudent object involves obtaining and saving all of the values from the attributes.

As part of the implementation, we must also determine whether we are doing an insertion or an update. This means we must be able to determine if the object being persisted represents a new student or an existing student. We must devise a mechanism that allows us to do that. In either case, we must be able to construct the necessary SQL statements that will allow us to insert or update multiple tables. The logical view of a student includes the TypicalStudent, StudentAddress, StudentMajor and StudentSubject tables. We know from the requirements that the minimum information that we can enter for a student is their name and address. As a result, the SQL statements to manipulate the corresponding tables should be included in a transaction, as we want to ensure successful inserts or updates for all tables involved.

Restoring Data

How do we create an object of the TypicalStudent class using data held in our database? We know that the primary key in our TypicalStudent table is the student ID. So, in order to retrieve an existing student, we must have the ID. We also mentioned earlier that

[51] There are indeed similarities between entities and classes. An entity is itself an abstraction. However, while classes are abstractions also, they include data and the operations defined on that data and can harness the power of Encapsulation and the other attributes of object-orientation. Strictly speaking, this, in addition to the semantics as defined for a class, differentiates tables from classes.

we would support searching based on a student's last name. In this case, we would also accept a last name, doing whatever we needed to do to match last name to ID.

Once we have the ID of the student, we need to execute SQL statements to retrieve data from all student-related tables. The data we retrieve is a "flat" representation of data that needs to be kept in multiple objects. Practically speaking, this means we need to be able to construct Name and Address objects, in addition to StudentMajor and StudentMajor objects.

FacultyStudent

Saving Data

TypicalStudent is also the superclass in our inheritance hierarchy. We have the opportunity to leverage this implementation or override it in the FacultyStudent class. Based on our implementation of Save() in the superclass, we will need to override this in the FacultyStudent class as that implementation executes SQL statements which do not work for the FacultyStudent class. All the steps we did for the TypicalStudent class are still valid. However, our SQL statements now have to include the data elements of the FacultyStudent class.

Restoring Data

Unfortunately, we will not be able to reuse the Restore() method of the superclass, for the same reasons outlined above. We therefore have to implement Restore() in the FacultyStudent class. The modification involves obtaining data from the FacultyStudent table, in addition to the other student-related tables.

TransferStudent

Saving Data

As with the FacultyStudent class, we will have to override the Save() method. In this case, we need to obtain data from the TransferStudent table in the database. As a result, we will need different SQL statements to accomplish this.

Restoring Data

Here also, we will not be able to reuse the Restore() method of the superclass, for the same reasons outlined above and for the FacultyStudent class. We therefore have to implement Restore() in the TransferStudent class.

HomeCollege

Saving Data

In our database, the data corresponding to a HomeCollege object is held inside the TransferStudent table. This data must be obtained by the Save() method of the TransferStudent class. As a consequence, it does not need its own Save() method. However, the accessor functions must be public.

Restoring Data

The HomeCollege class does not need its own Restore() method either. The logic here is the opposite of the logic for saving data.

Name

Objects of class Name would be not need their own Save() or Restore() methods, as their data is maintained in our database as part of the TypicalStudent table.

Address

Objects of class Address would be not need their own Save() or Restore() methods, as their data is maintained in our database as part of the TypicalStudent table.

StudentMajors and Major

Saving Data

An object of this class represents all of the majors of an individual student, zero, one or two (per requirements). An object of StudentMajors corresponds to the rows in the StudentMajor table for a given student ID. Since an object of StudentMajors is an aggregate of objects of class Major, each object of class Major represents one row returned from the StudentMajor table for a given student ID. SQL statements would be constructed to insert or update the data in the StudentMajor table.

Restoring Data

When restoring data from the database, we have to construct an object of StudentMajors will contain the rows returned from the StudentMajor table for a given student ID. As above, we will construct objects of class Major for each row returned from the StudentMajor table for a given student ID.

StudentSubjects and Subject

Saving Data

An object of this class represents all of the subjects being taken by an individual student, zero, or more (per requirements). An object of StudentSubjects corresponds to the rows in the StudentSubjects table for a given student ID. Since an object of StudentSubjects is an aggregate of objects of class Subject, each object of class Subject represents one row returned from the StudentSubjects table for a given student ID. SQL statements would be constructed to insert or update the data in the StudentSubjects table.

Restoring Data

When restoring data from the database, we have to construct an object of StudentSubjects will contain the rows returned from the StudentSubjects table for a given student ID. As above, we will construct objects of class Subject for each row returned from the StudentSubjects table for a given student ID. SQL statements have to be constructed to retrieve data from the StudentSubjects table.

AllMajors

Saving Data

An object of this class represents all of the majors available to students. An object of StudentMajors corresponds to the rows in the AllMajors table. Since an object of StudentMajors is also an aggregate of objects of class Major, each object of class Major represents one row returned from the AllMajors. SQL statements would be constructed to insert or update the data in the AllMajors table.

Restoring Data

When restoring data from the database, we have to construct an object of AllMajors will contain the rows returned from the AllMajors. As above, we will construct objects of class Major for each row returned from the AllMajors.

AllSubjects

Saving Data

An object of this class represents all of the majors available to students. An object of AllSubjects corresponds to the rows in the AllSubjects table. Since an object of AllSubjects is also an aggregate of objects of class Subject, each object of class Subject represents one row returned from the AllSubjects table. SQL statements would be constructed to insert or update the data in the AllSubjects table.

Restoring Data

When restoring data from the database, we have to construct an object of AllSubjects will contain the rows returned from the

AllSubjects. As above, we will construct objects of class Major for each row returned from the AllSubjects table.

AllStudents

An object of this class represents all of the students. An object of AllStudents corresponds to the rows in the TypicalStudents table, with additional knowledge of whether they are typical, transfer or faculty. An object of AllStudents is obviously an aggregate of objects of class TypicalStudent, FacultyStudent or TransferStudent. So, to save data, we would invoke the Save() method as defined for each object. Likewise, to restore data, we would invoke the Restore() method as defined for each object.

System Functionality and Report Requirements

From the report requirements, we know that there are various business functions that we must provide. These will be essentially insertion, deletion, update or query operations. These are outlined below:

Adding a new student's information

This entails inserting new student information into the database. This is an insertion operation. As stated in the requirements, we will not have to deal with storing incomplete student data. The assumption is that we will always be entering a complete student record, with the exception of majors and subjects. We should assume we will have complete name, address, etc.

Searching and displaying a student's information

Retrieving a student's information is based on executing a series of queries based on the student's ID (simplest option). However, we should also support queries by student last name, which may return multiple records. In any case, we will be executing a series of queries as we need to create the appropriate objects in our object model from the data in the database, which does not map exactly. For example, how will we determine whether a student is typical, faculty or transfer? We first have to obtain a student ID that is either entered by the user or derived from the student's last name. We have to query the FacultyStudent query with an ID. If there are records with this key, the ID belongs to a student that is also a member of faculty. If not, we still have to execute a query to determine if there is a record corresponding to this ID in the TransferStudent table. Of course, we

are assuming that a transfer student can never be a faculty student and vice-versa. This is implicit in the requirements.

Deleting a student
This function involves deleting all records related to a particular ID from all tables in the database.

Changing/assigning classes and credits to students
This is effectively an update operation that requires us to update class and credit data based on an individual ID.

Changing/assigning a student's major
This is effectively an update operation that requires us to update class and credit data based on an individual ID.

Changing/assigning a student's type
This operation is a bit more complicated, as it involves multiple database operations, depending on the predecessor and successor types. If a student changes from faculty to typical, that involves a deletion of data from FacultyStudent. If the direction was reversed, we would insert into FacultyStudent. Similar operations would occur if a transfer student became a typical student, say at the beginning of the following school year. We would have to remove the transfer college information from TransferStudent. To go from a transfer student to a faculty student, or vice-versa, we have to do two operations, a delete from one and an insert into the other. These two operations need to execute successfully, as the database would be in an inconsistent state otherwise. These operations would need to be included in a transaction[52].

Changing/assigning a student's status, i.e. from full-time to part-time or vice-versa
This is effectively an update operation that requires us to update class and credit data based on an individual ID.

The same analysis can be done for the reports that the system must provide. The list of reports is below.

Sorted list of full-time students (all information)
This is implemented as a query that returns all of the students with more than ten credits

[52] A transaction represents a group of operations that execute atomically, i.e. as one. This means if one operation in the transaction fails, the transaction as a whole, fails.

Sorted list of part-time students (all information)
This is implemented as a query that returns all of the students with less than ten credits

Number of students of each type (typical, faculty and transfer)
This is implemented as a series of queries as we have to determine how many records are in the TypicalStudent table that are not in either of the other two tables, in addition to determining how many records are in each of the FacultyStudent and TransferStudent tables.

For each type of student, a sorted list of student names and addresses
This is similar to the solution above, except we need to produce a sorted list, not a count.

For each type of student, a reversed list of student names and addresses
As above, but sorted in reverse order.

List of all students, their majors and number of credits
This is similar to the very first report and would be implemented in a like fashion.

A sorted list of all students based on their cost for the semester
This operation will require a series of queries as well, in order to calculate the appropriate cost for the semester.

Implementing Data Management Methods

From our discussion earlier, we know we have two choices. One is to define a class that represents the entire database, with operations defined that match the business functionality requirements, or to include similar operations in other classes. How should we create an abstraction for our data management? Another way of looking at this is to say, what level of abstraction should we select for our data management operations? So far, we know we will have Save() and Restore() methods defined for many classes. In addition, we will have methods that correspond to the data required for our reports. We also have classes AllStudents and System.

Creating an abstraction representing the database gives us some advantages. For example, we can centralize the mechanism to connect to the database, i.e. supplying the database name, user id, password, etc. In addition, we could also centralize handling cursors,

executing dynamic SQL statements and interacting with stored procedures. The interface for such a class would reflect this. To avoid confusion, let's name this class DBClass[53]. Since we are working with one database, there would be one instance of this class in our system. We will need to define one or more constructors and a destructor for DBClass.

Now we need to determine which classes need to interact directly with DBClass. Some object has to create an instance of DBClass. This will be an object of class System. We also need to interact with the database to manipulate student data. The logical place to have this interaction is in class AllStudents, as this represents all of the students in the system.

Class Details Revisited

In order to get our system to "hang together" correctly, we have to modify the definitions of constructors and other methods, add new attributes, etc. Let's review the list of classes to be modified.

System

The major modification is that we have to add a new attribute: an instance of DBClass. This will represent the database we will be using throughout. Based on our simple example, it is logical that we would attempt to instantiate this attribute during the construction of the system object itself. In addition to other functionality, the destructor for System would have to "clean up" the object of class DBClass.

DBClass

This is a new class we "discovered". Since it represents a database, it is appropriate for its constructor to take the parameters necessary to make the connection. Let's assume we need a database name, user id and password, all character strings. The constructor is then as follows:

```
DBClass(database_name, user_id, password)
```

We could also define a destructor for DBClass that would terminate the connection to the database.

In addition to the methods above, we could also define read-only accessor methods in the class to return the name of the current

[53] Creating the new class DBClass is an example of identifying classes by discovery, as mentioned in Chapter 2.

database name and the name of the current user (Get() methods only, no Set()).

AllStudents

Since this class will be interacting directly with the database, we need to have a way to refer to the object that represents our current database. As in everything, we have choices here as well. We could make an accessor function that returns the database object a public method of the System class, in which case we need to be able to refer to the system object in order to use the database. The other choice is to make a public accessor function as well, but also supply a reference (or a pointer) to the database object to the object of class AllStudents for its use. This solution is more elegant, as the only thing we need to use is the database object, not the entire system object. We should not need to pass references to the system object around. In order to implement our choice, the constructor of AllStudents() would need to be modified to accept the reference to the database object. In addition, we would need to add an attribute in AllStudents to store that reference. The other methods currently defined in AllStudents remain the same.

AllStudents is a class that is an aggregate of TypicalStudent, TransferStudent and FacultyStudent objects. We will add methods GetFirst() and GetNext(), Count(), Append() and Delete() to the interface of all aggregate classes. This will make the interaction with them easier. GetFirst() and GetNext() would allow callers to navigate through the object that are in the aggregate. This facilitates any browsing of the objects in the class. Count() would return the number of objects currently in the aggregate. Append() would add objects to the collection (aggregate) and Delete() would remove objects from the collection.

Address

As with AllStudents, we will add GetFirst() and GetNext(),Count(), Append() and Delete() methods. This is because a student could have more than one address.

Student Majors

As with AllStudents, we will add GetFirst() and GetNext(),Count(), Append() and Delete() methods.

Student Subjects

As with AllStudents, we will add GetFirst() and GetNext(),Count(), Append() and Delete() methods.

All Majors

As with AllStudents, we will add GetFirst() and GetNext(),Count(), Append() and Delete() methods.

All Subjects

As with AllStudents, we will add GetFirst() and GetNext(),Count(), Append() and Delete() methods.

Our updated class models are below.

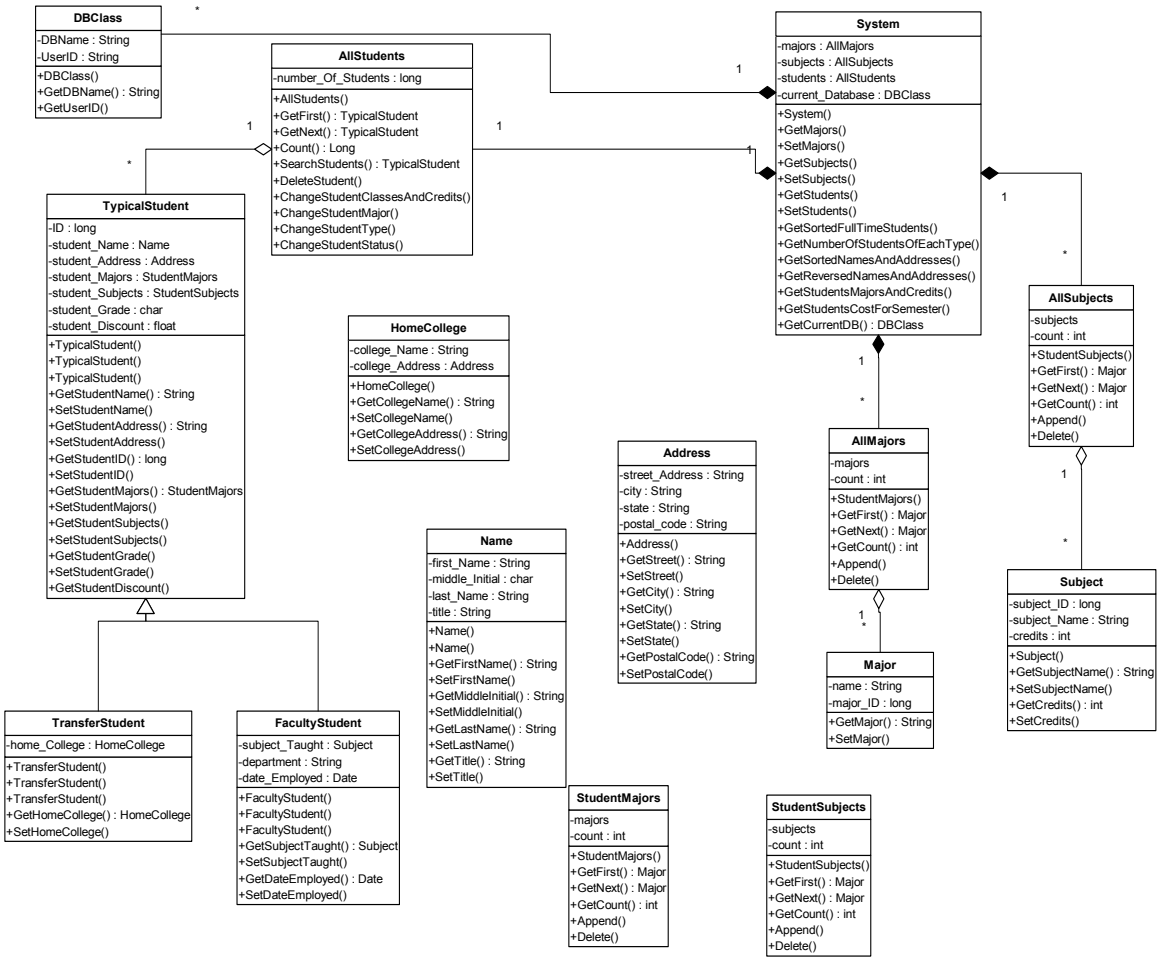


Fig 6.5 Student-related focus

In this diagram, we are focusing on the "student" classes, highlighting the relationships therein.

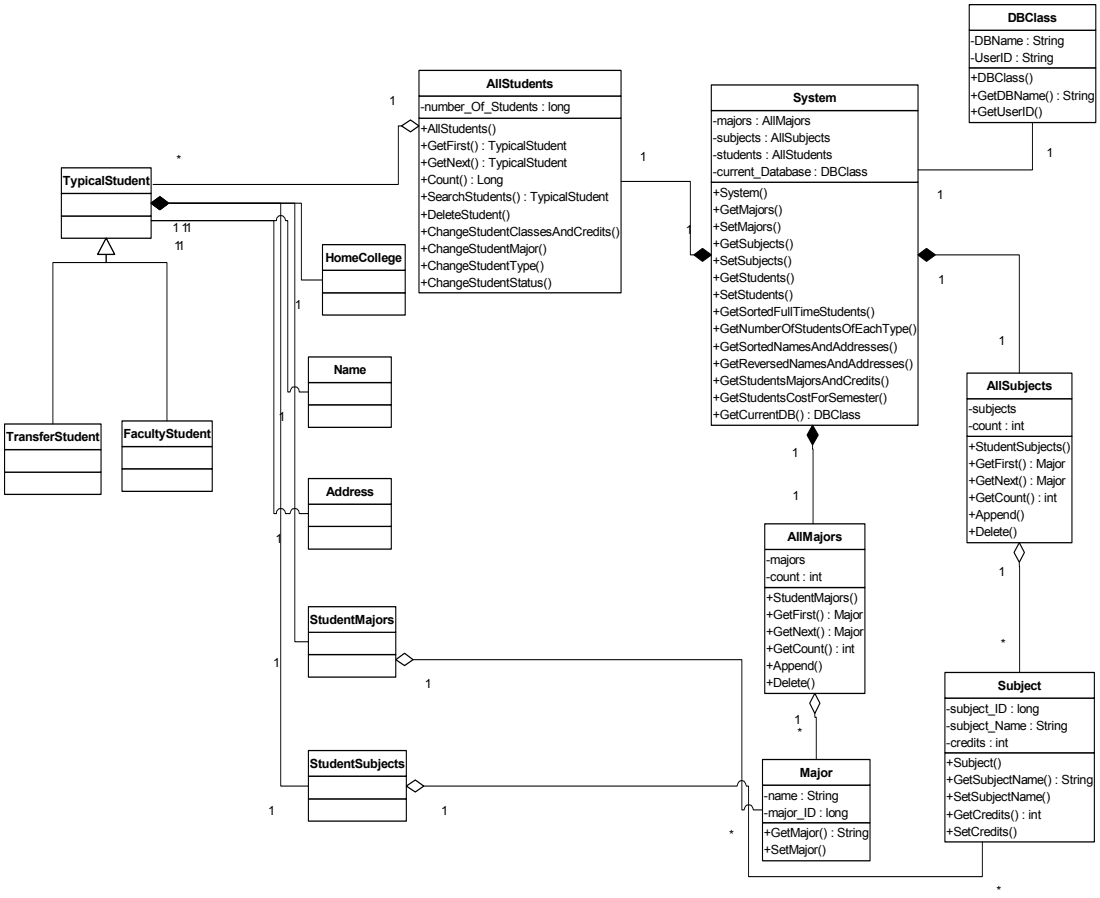


Fig 6.6 System-related focus

User Interface

Creating usable human-computer interfaces is a topic that could fill a volume by itself. That is not our intent here. As such, we will not attempt to review all the elements of a "good" user interface. Instead, we will discuss how we design the presentation, given the requirements. In our example, it is obvious that we will have to provide a user interface. It is just as obvious that

In order to accomplish this, we must quickly review the form of the various data elements we have to work with. For example, we need to present a student's ID, name, grade, discount, etc. These are single values, meaning there is only one for each student. Some of the data we have may multiple values per student. For instance, a student may have multiple majors, subjects and addresses. In addition, we also keep track of all available majors and subjects. We will use a different metaphor to present this data.

The objective of a user interface is to provide a usable interface for an application. Consequently, a user interface should be designed from the perspective of the user and to benefit the user, while factoring in technical constraints that exist and which will be accommodated in the design. A user-interface should not reflect the underlying data or object-structure, unless indeed that is easiest for the users to use. The key word in "user interface" is "user".

Each user interface screen is comprised of various GUI elements[54]. The choice of elements, layout, usage, usability, "look and feel" are what separates good user interface designs from great user interface designs. How do we know when we have a good design? That is not an easy question, as it is somewhat subjective. However, we do have tools at our disposal to help make sure our user interface satisfies all requirements[55]. One of those tools is Prototyping.

Prototyping

A picture is worth a thousand words. A direct effect of this is the importance of Prototyping. Prototyping is very useful for presenting enough details of a system to increase understanding and generate feedback. Prototyping is an activity that starts in the Analysis phase, once there is some understanding of the requirements.

[54] Graphical User Interface. See Appendix 3 for a discussion.
[55] There may be additional requirements that are supplied that govern the user interface. These would be categorized as non-functional requirements also.

At the simplest, a prototype may consist of paper-based designs. A simple prototype could be created using simple shapes (rectangles, squares, etc.) to represent GUI elements such as windows, menus and buttons. This serves to give the user a "feel" of the application's interface. In other cases, more complicated prototypes may be developed. Obviously, the purpose of the prototype is to facilitate the understanding of the requirements and to set expectations.

As in other aspects of object-oriented development, Prototyping is iterative also. It may take several iterations before the developers, analysts and users are in agreement. While this may seem tedious, this actually underscores the importance of prototyping, as potentially troublesome issues can be addressed easily (and cheaply) in the prototype. Some of these problems could become major issues if they had to be rectified later in the system development cycle. In general, the earlier problems are identified and solved, the better of everyone is.

System Prototype

For our example, we will review a very basic prototype of the system. The prototype covers some areas of the system. It is not complete. It represents what would be the beginning of our iterative process. We would expect the prototype to be modified, possibly significantly, before it was considered ready and approved.

This figure outlines a proposed menu structure and the main screen of the application. The menu choices represent activities culled from the requirements. Let us review the menu options.

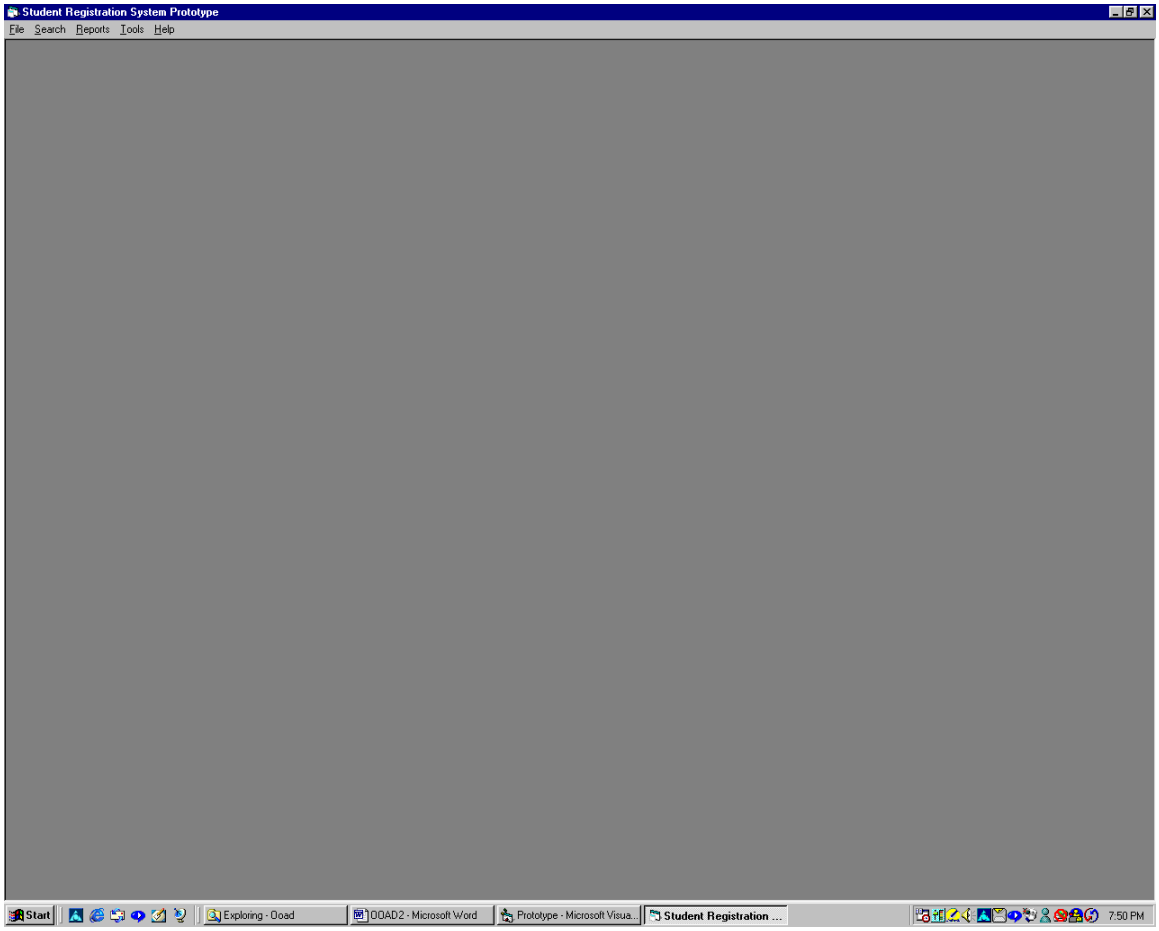


Fig 6.7 Main menu

Our main menu items would also have sub-menus. Under the File menu, we could have menu options to add and delete students and exit the system.

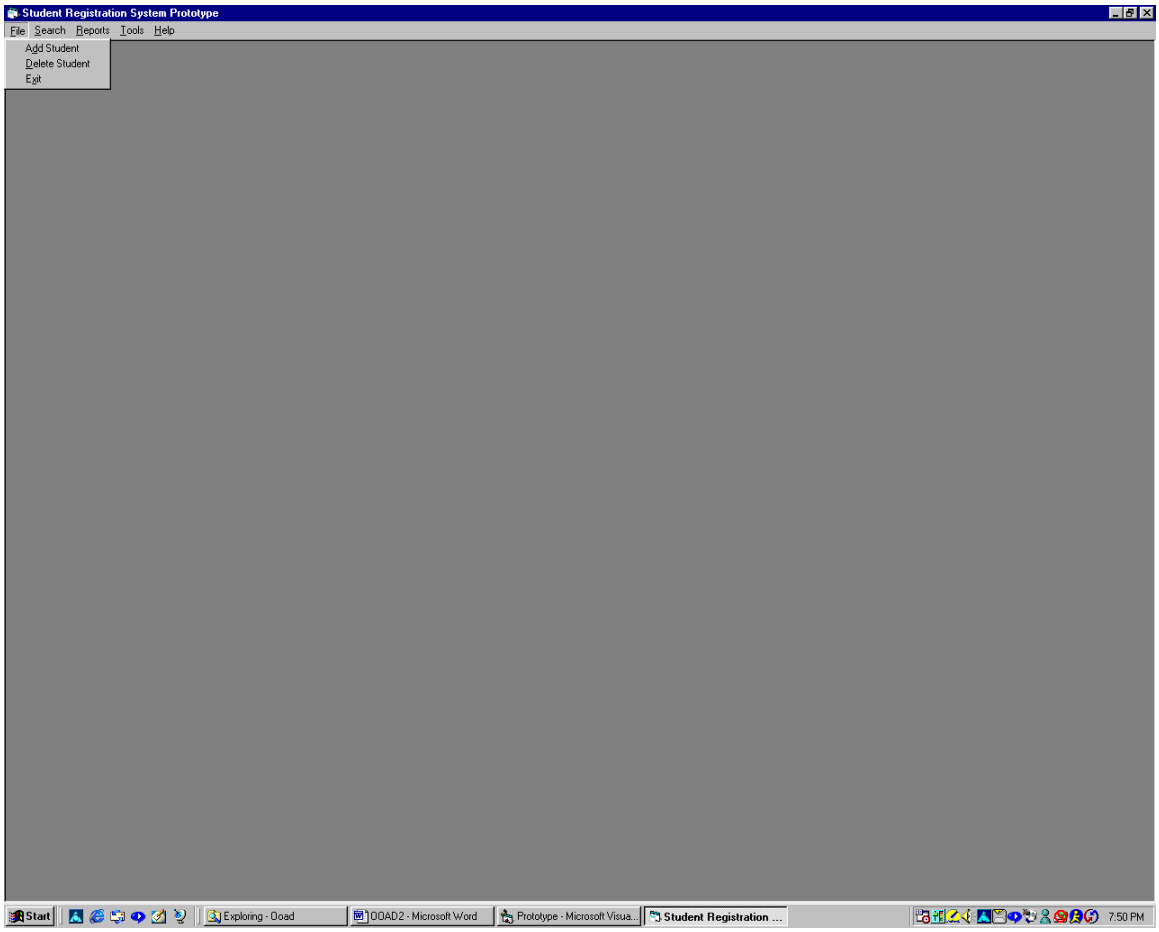


Fig 6.8 File sub-menu

These may not be the only options that we include under the File menu. As we develop the prototype, we may find that it is appropriate to add more options to this menu.

Adding a Student

If we select "Add Student" from the File menu, a window is launched that displays the form to be used to enter student information. In our prototype, this is the only screen used to capture student information. As such, it has all the fields needed for each type of student, all in one place.

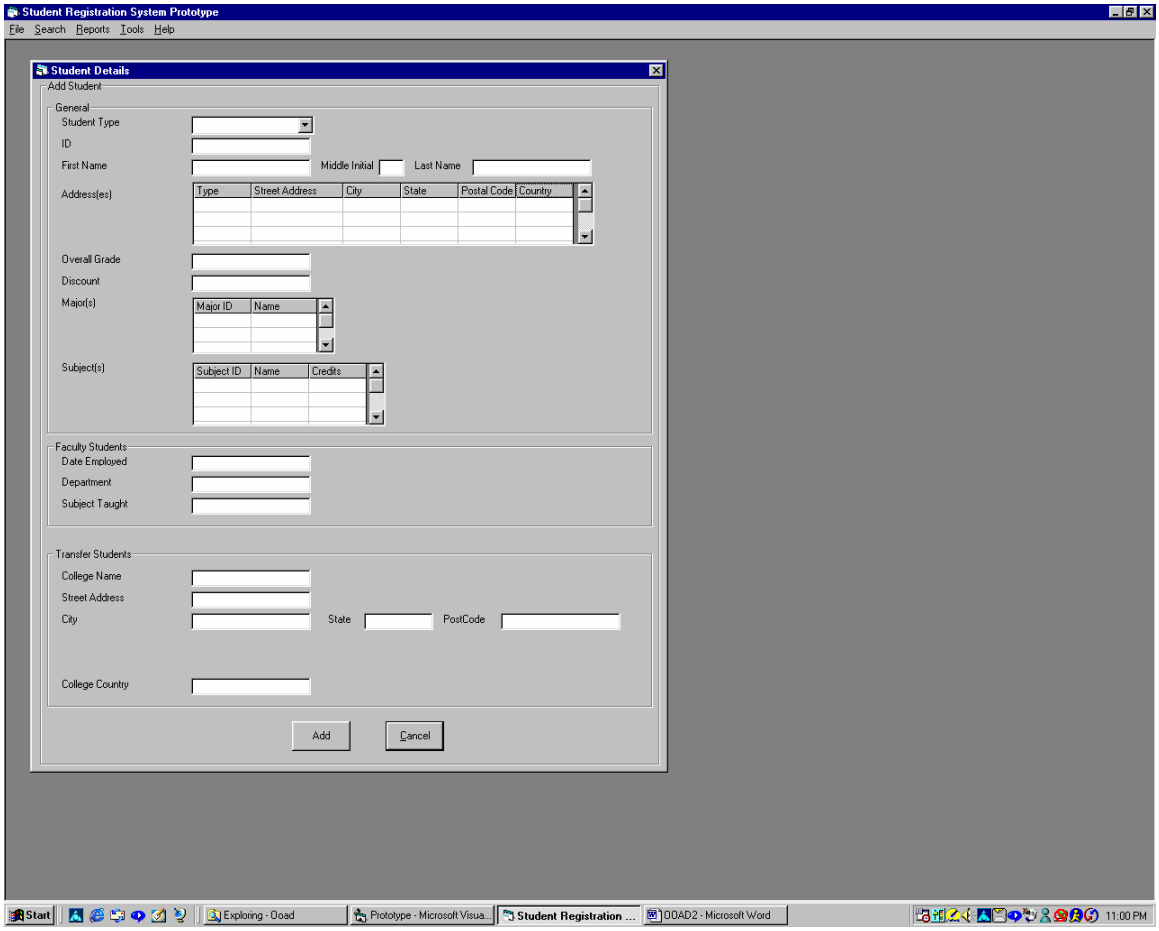


Fig 6.9 Adding a new student

When we add a student, we need to know what type of student we are adding. We use a combo box to allow the users to select the type. The user would select the type corresponding to the type of the student that is being added. This would cause certain fields to be available and others to be unavailable, as warranted by the type of student. Examples follow.

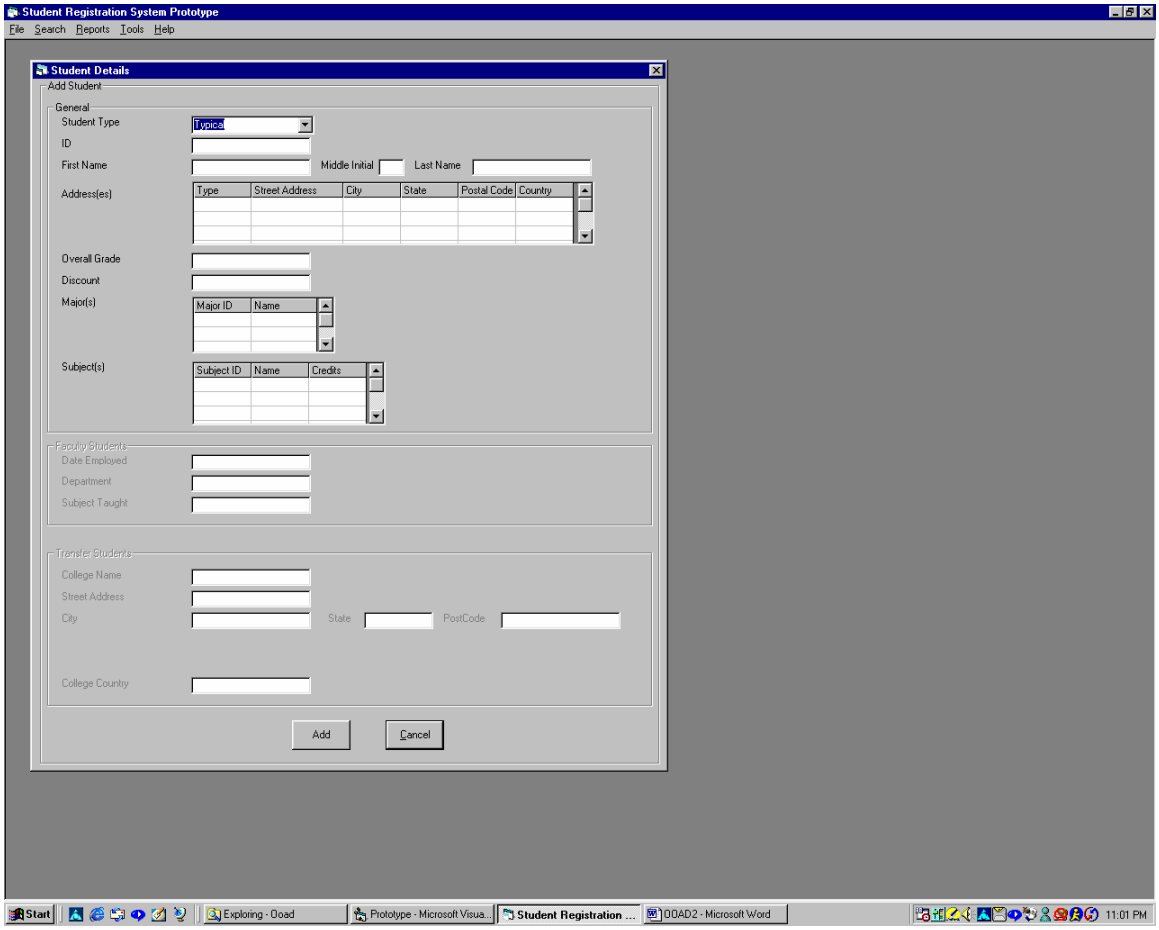


Fig 6.10 Adding a Typical student

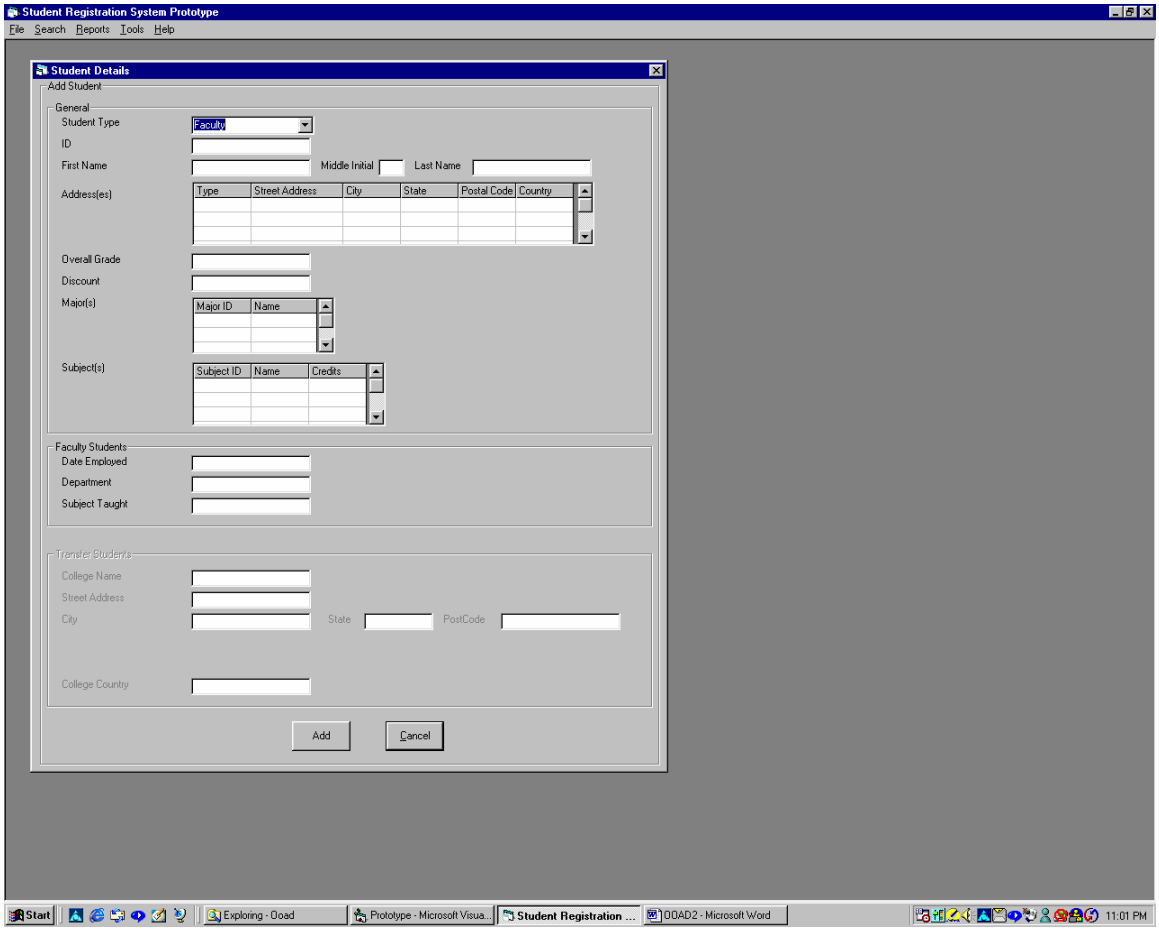


Fig 6.11 Adding a Faculty student

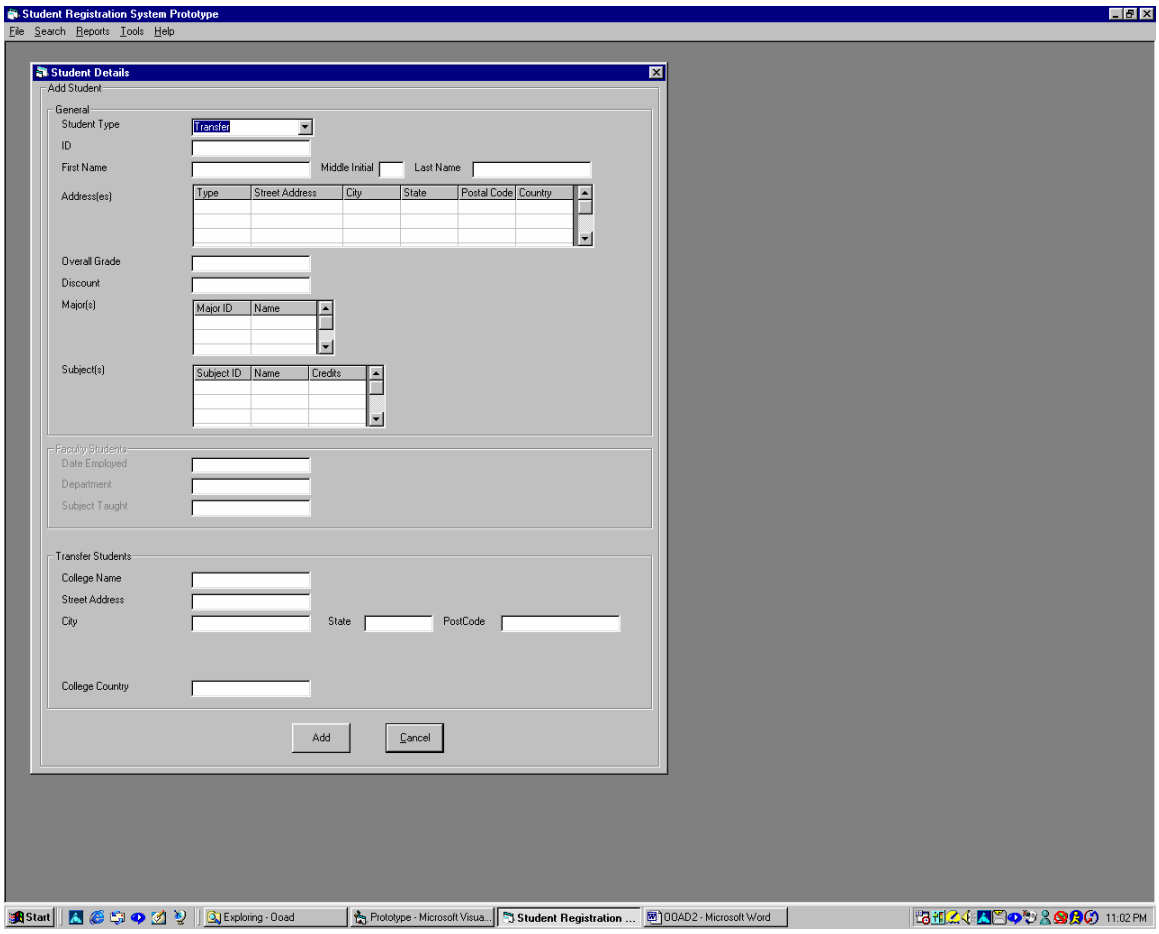


Fig 6.10 Adding a Transfer student

Searching for Students

Our requirements dictate that the system must allow searches of students. The user interface for searching is described below. It has multiple steps, broken down into obtaining search criteria, presenting search results and displaying the detail for the selected student. If the search criteria is a student's last name, we may have multiple results as there may be many students with the same last name. Thus, we need to display multiple rows of information in our search results and allow the user to select the appropriate one.

To allow us to search, we have a menu option, "Find Student" that is a sub-menu option of the "Search" menu.

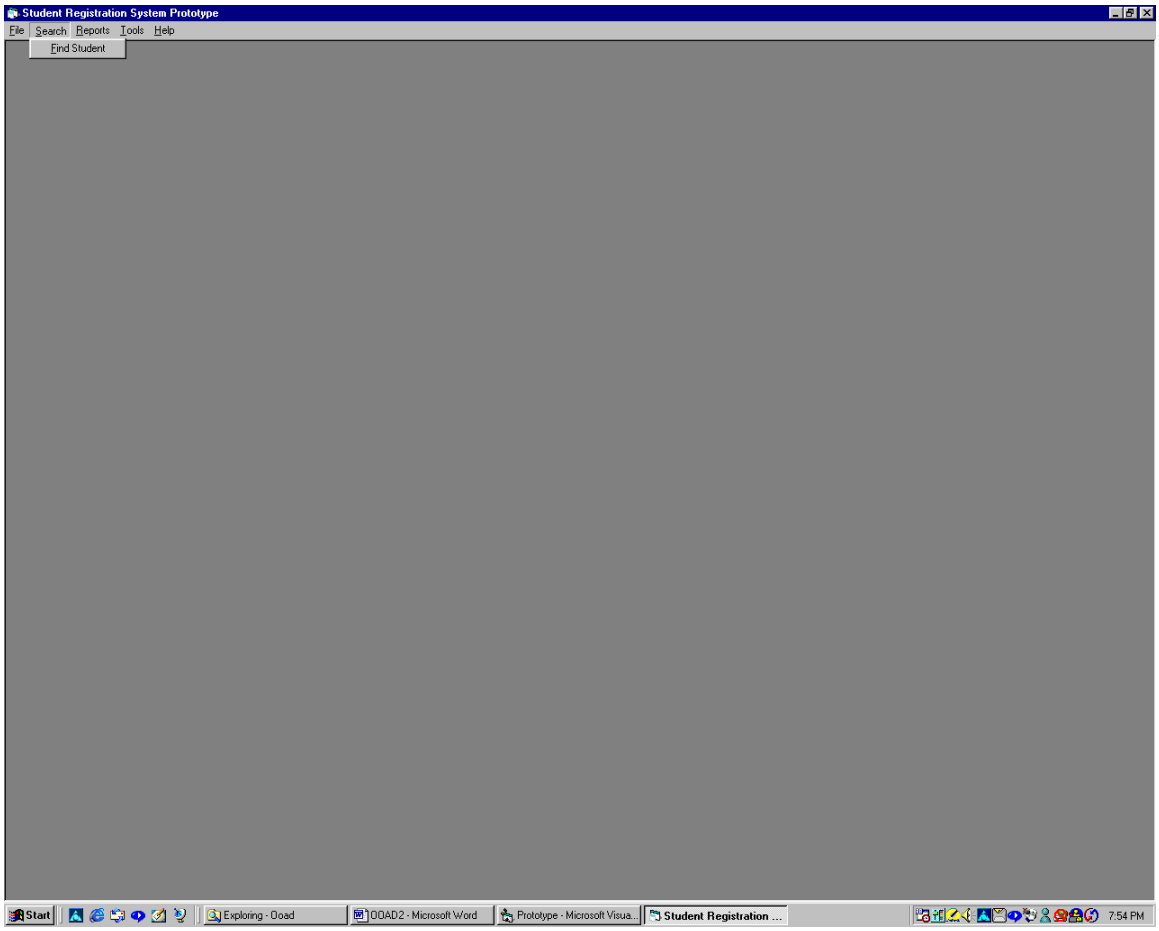


Fig 6.11 Find Student menu item

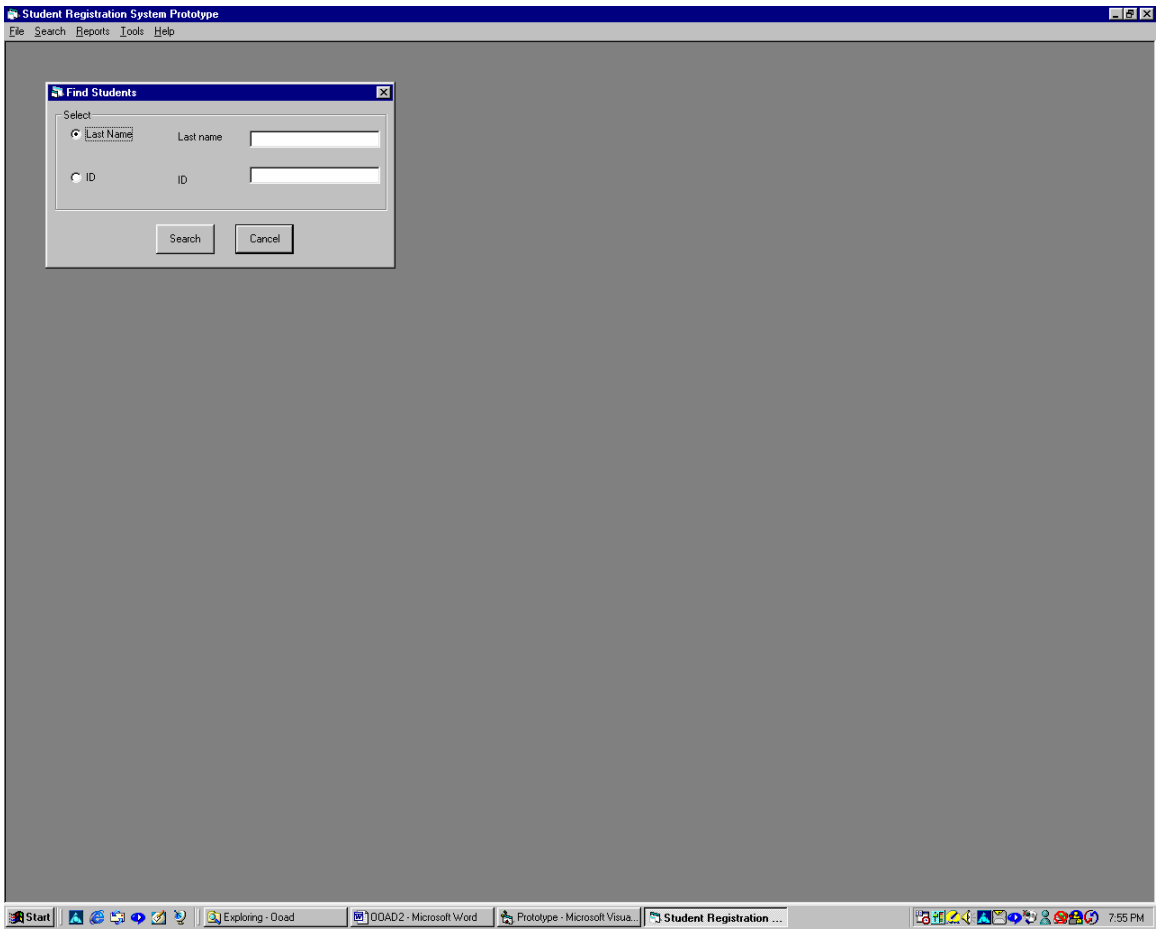


Fig 6.12 Search criteria

On this form, we see the two search options that we incorporated earlier. We are able to select whether we want to search by a student's last name or by their ID by selecting the appropriate radio button.

Once we click on the "Search" button, we would execute the search based on the criteria and launch another window ("Search Results") to display the search results.

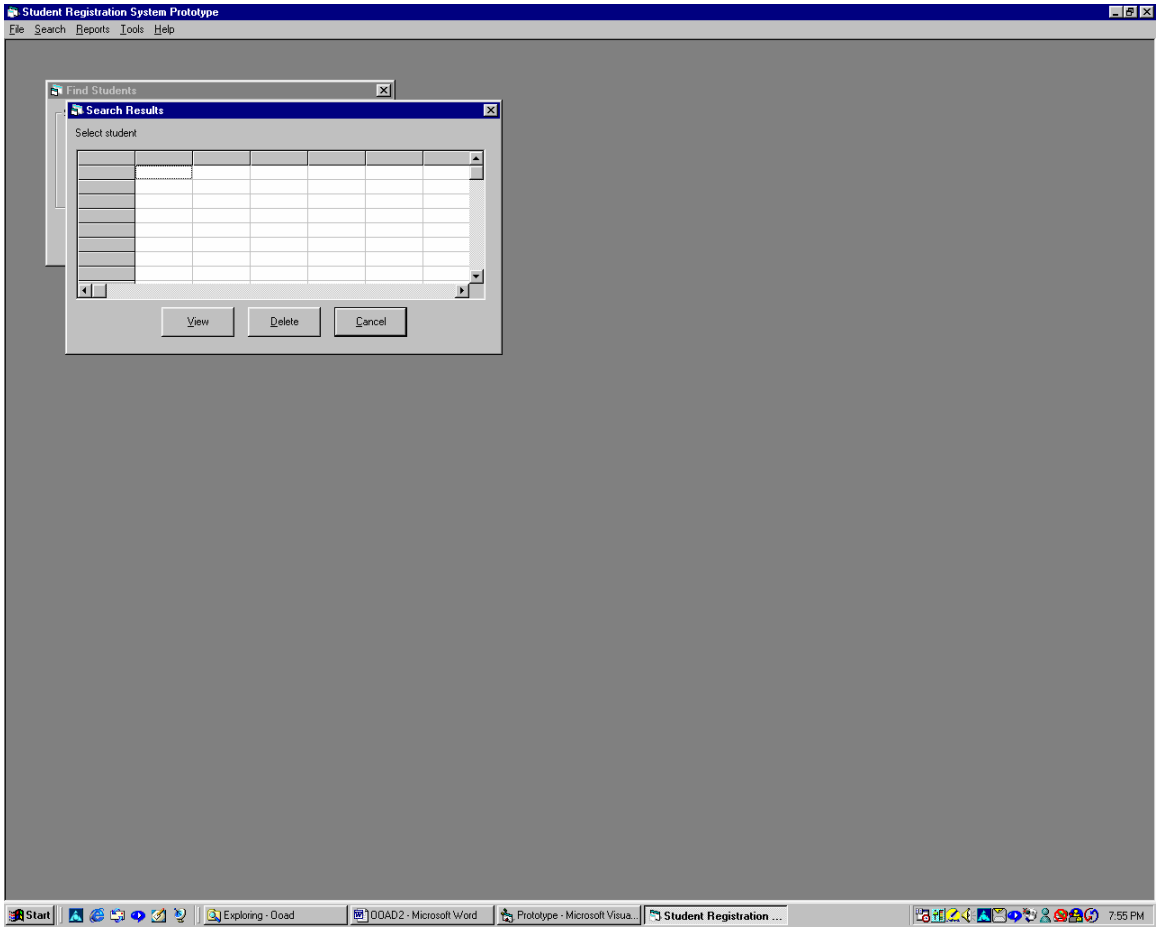


Fig 6.13 Search Results

As we said, we may obtain multiple results from the executing our search. We will use a tabular grid, to display one or more rows with multiple columns.

The user would select the correct student from the display. Once that selection was made, the third form in the sequence would be displayed. This is the same form as was used for adding a new student, with some contextual changes. For example, we no longer need a combo box for the student type. The combo box will be replaced with a text box, as this form is now being used to display existing data, not create new data. Indeed, we will have to make the text box un-editable.

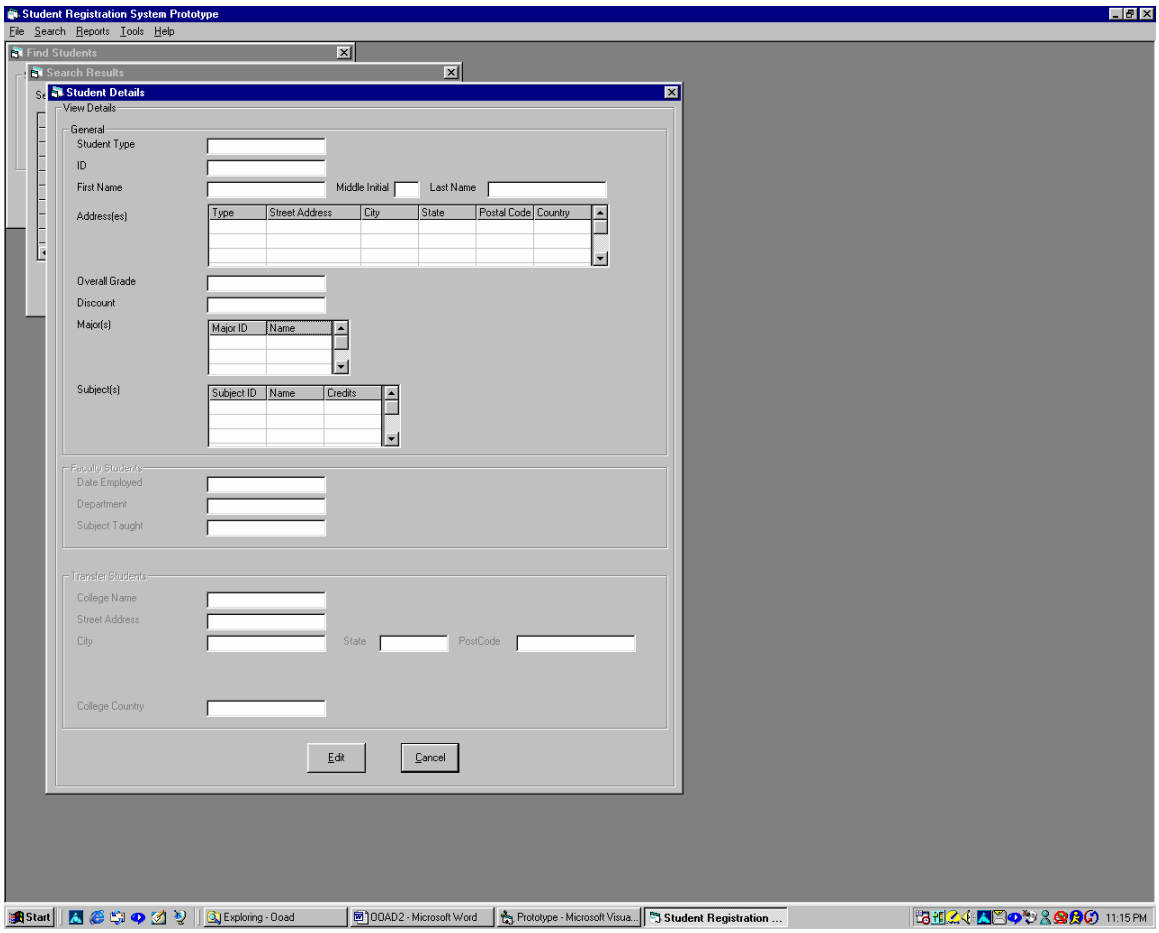


Fig 6.14 View student details

This example shows how the screen would look if we retrieved information for a typical student.

From before, we know a student might have multiple addresses, majors and subjects. Each of these corresponds to an object that is an aggregate. The methods GetFirst(), GetNext() and Count(), as defined on each of the "aggregation" objects, will be very useful in populating these tabular grids. As before, each column in the grid would correspond to an attribute from the object.

We also have to give the users the ability to modify a student's data. In order to accomplish this, there is an "Edit" button on the screen. The idea is that the users would click on this button to modify data. This would allow the elements on the screen to be editable, not "read-only" as in "view" mode.

Once we return to "edit" mode, the screen elements would revert to those apparent when we chose "Add Student" from the "File" menu, i.e. the combo box would return, etc.

Reports

All of the reporting options of the system could be grouped under the "Reports" option on the main menu. We could place all report options on a single separate form as follows.

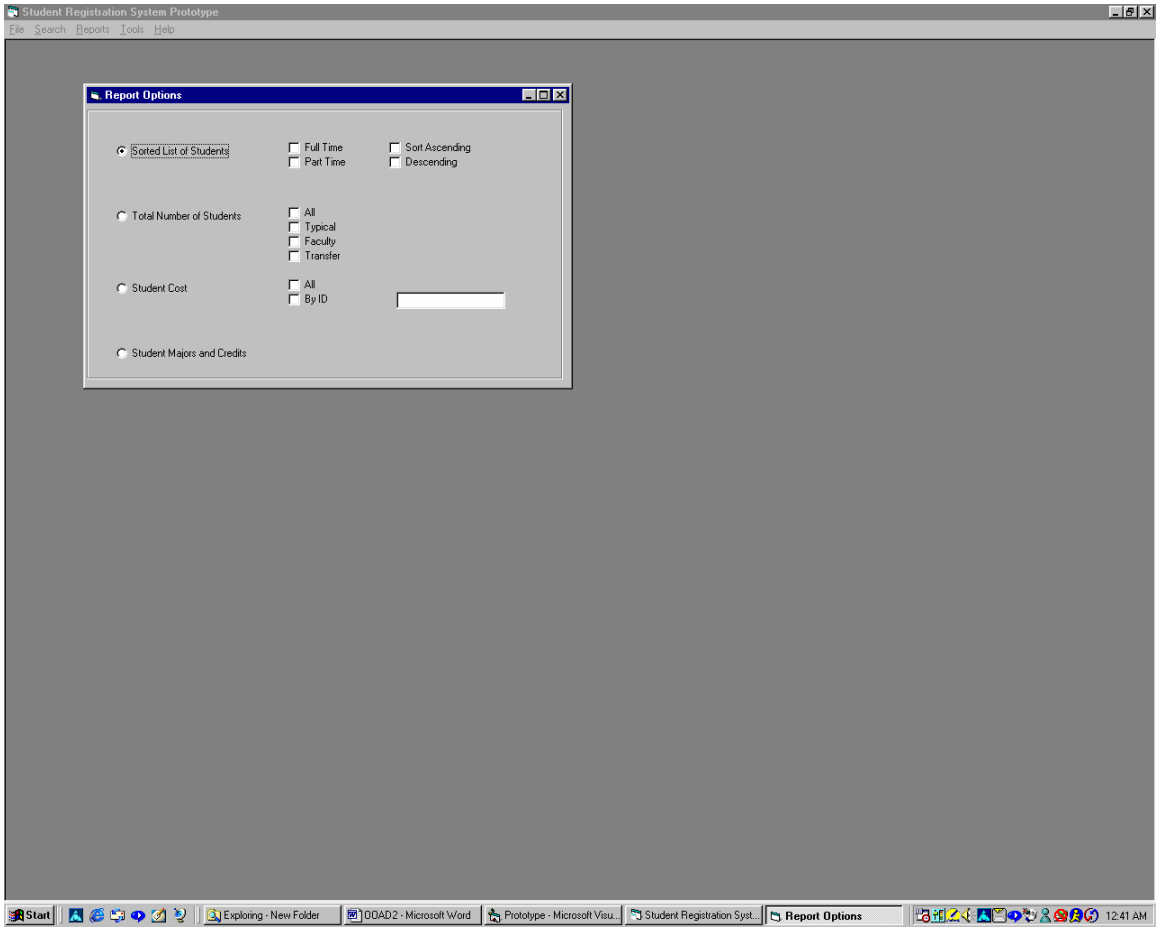


Fig 6.15 Report options

Maintaining System Data

As we saw with our data model, there are other data elements that we need to allow our users to maintain. They need to maintain subject and major data. We've grouped these elements under the Tools menu, as you can see from the picture below.

Each of these options would probably need to have its own form which would give the users the ability to add new majors, delete majors, add new subjects and delete subjects.

As we go along, we may also discover additional tools and utilities that need to be included. For example, maintaining the discount rate that is offered by the institution, for which we do not yet have an interface.

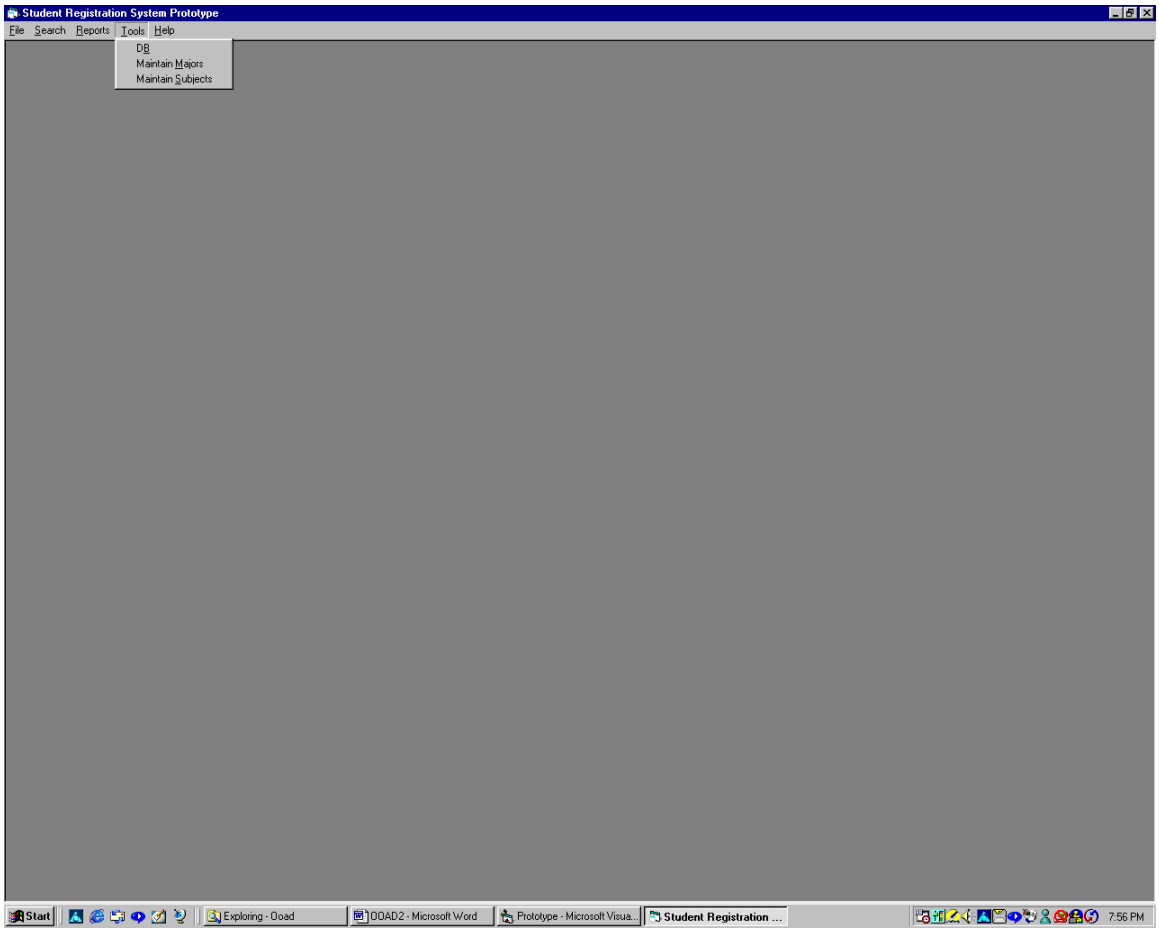


Fig 6.16 Tools menu

System Help

As a matter of course applications should provide help for users. Typically, there is a main menu item called "Help". Among the items in the "Help" menu, we may also find the "About" item which gives information about the system such as version, serial number, operating system information, etc.

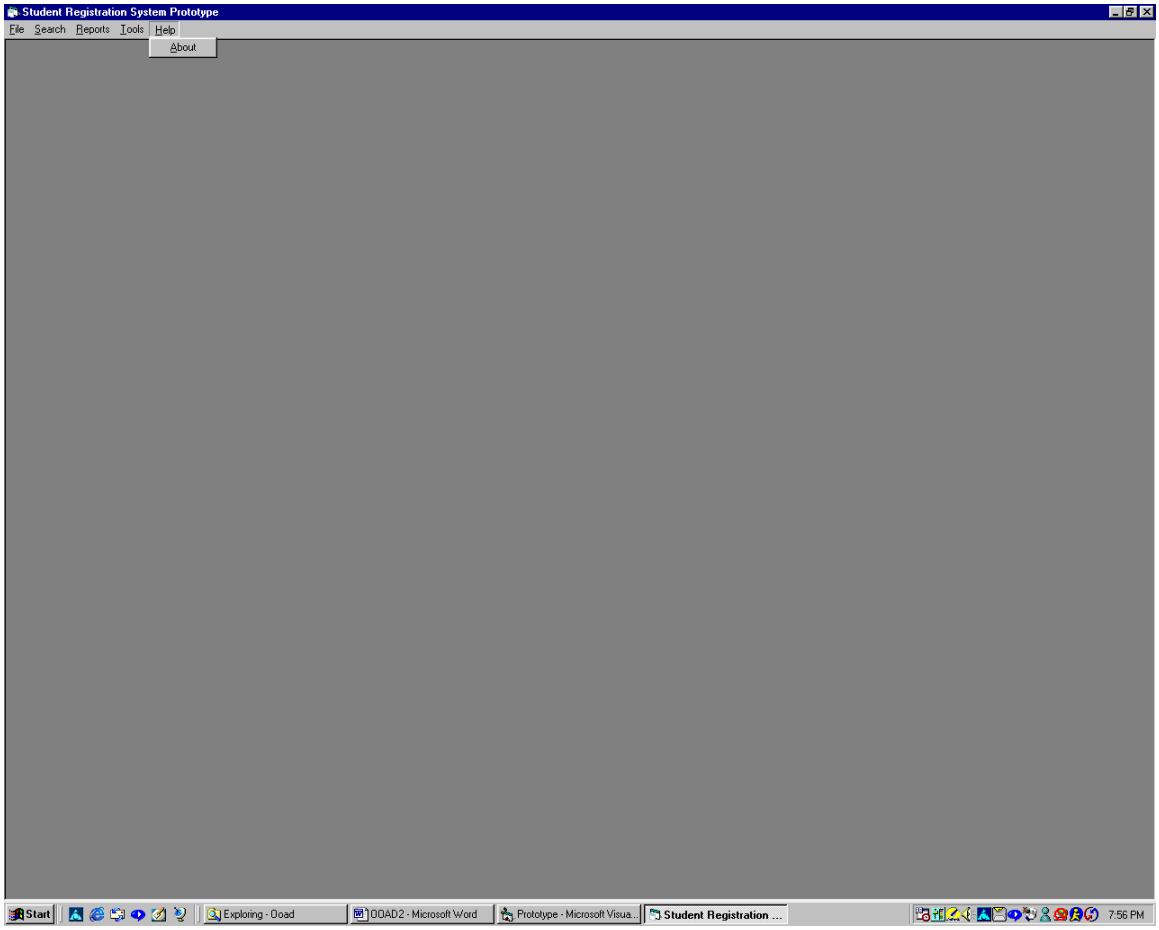


Fig 6.16 Help menu

Summary

We've taken an object-oriented approach to designing this system. This is definitely not the only possible design we could have discovered. However, it is important, whatever the design ultimately is, that you are able to justify the decisions and tradeoffs that you make. Each decision will have ramifications, some minor, some major.

While we have gone through a few iterations, we are not finished. This is a design in the "middle" stages and we definitely have more to do. For example, we need to do more work with our methods. We are assuming the methods are public and the attributes are private or protected. However, this is not necessarily true. Also, we have objects that do list management. However, because these objects are manipulated from "within" another object, i.e. are part of a composition relationship, we may decide that we need accessor methods in the "outer" objects that call methods of the "inner" object. Why? If we declare the "inner" object, i.e. AllStudents, StudentSubjects, etc., as private in the "outer" class, then this object would not be accessible from the outside of the "outer" object. This means if AllStudents is a private member of System, then no object outside of System could cause a student to be added. This may or may not be okay, depending on the language to be used for implementation. So, we may need to create or utilize an accessor function to allow objects from the "outside" of this class to invoke methods which in turn invoke methods of the "inner" class. Of course, the methods of the "inner" class would have to be public, or none of this works!

We took a first try at a user interface. However, there are many other details to work out. Firstly, our prototype is incomplete. We only prototyped adding and viewing student data. We still need to prototype editing data, maintaining system data, deleting students, reports, etc. If we look further, there are some details of our abstractions that have yet to be finalized. As part of the iterative nature of object-oriented development, we would expect to review our classes, their attributes and operations until we were comfortable with the outcome.

In addition, we would most likely look to implement our aggregate classes, i.e. AllStudents, StudentMajors, etc. with pre-defined collection classes or creating instances of parameterized classes. This is because, though the classes may hold different objects, their

behavior is the same. We would have to ensure that such classes provided the necessary methods such as GetFirst(), Count(), etc.

Evaluation

Let us evaluate our design using the metrics we discussed earlier in the chapter. These metrics allow us to test the quality of the classes we've chosen, the overall semantics, operations, etc.

Coupling

Coupling is the measure of the strength of association established by a connection from one class or object to another. Strong coupling complicates a system, since a module will be harder to understand or modify by itself, thus adding to the overall complexity of the system.

Our design has some elements that are tightly coupled and some which are loosely coupled. The inheritance relationship represents relatively tight coupling in terms of any significant changes to the base class structure. We benefit from this tight coupling, as we can leverage attributes and methods of the base class in the subclasses. Here is an example of a trade-off, stemming from one of our decisions. In fact, this appears to be the tightest coupling in our design.

We also have elements that are loosely coupled. The associative relationships involving the TypicalStudent and System classes, and those involving the "list management" classes are more loosely coupled. We can make changes to the classes on either side of the association, and not have any (or minimal) effect on other classes.

Cohesion

Cohesion is the measure of the degree of connectivity among the elements of a single class or object. Entirely unrelated abstractions should not be placed together in one class. Unrelated behaviours should not be captured in the same class.

In our design, we've separated our abstractions greatly, creating new abstractions for functions such as managing lists, etc. In reviewing our classes' responsibilities, we have limited our attributes and methods to only those that are applicable to a particular abstraction. The greater the extent to which this is done, the more abstractions, i.e. classes, your system will have. This is another explicit decision to be made, which yields a trade-off.

Sufficiency

Sufficiency is the measure of whether or not a class or module captures enough of the characteristics of the abstraction to allow meaningful and efficient interactions.

We have to look at sufficiency in the context of the requirements. We've had to supply information where it was lacking in the requirements (in practice, when this occurs, we would have consulted the business experts). In this context, how do our classes measure up? We are somewhat sufficient thus far, but as we mentioned earlier, we are not finished, i.e. we need to review methods, etc. So we would potentially expect an incrementally greater degree of sufficiency at the true end of the exercise.

Completeness

Completeness is the measure of whether or not the interface of a class captures all of the meaningful characteristics of the abstraction.

Completeness and Sufficiency would seem to be at odds. However, we have to find a balance. In the context of the requirements, we seem to be quite complete. However, as we refine further, we would expect an incrementally greater degree of completeness as well.

Primitiveness

Primitiveness is the measure of the ability of operations to be efficiently implemented, only if given access to the underlying representation of the abstraction.

Based on the methods defined thus far, we may infer that they are somewhat primitive. However, we have not defined how our methods are implemented, i.e. what their steps will be. We need to examine our methods for redundancy, within each class. This would be a by-product of the refinement we have alluded to earlier.

This exercise has demonstrated the incremental and iterative nature of creating an object-oriented design. As we refine our design, we may yet discover that we have to change our design to refine behaviours, and to be better aligned with the spirit of the metrics that we use to evaluate our abstractions.

We have only depicted class relationships. One aspect of our design is how we expect objects of our classes to interact at run-time. We need to consider this as the functionality of our system is based on the collaboration and interactions between the objects of the classes in our

design. We can depict these collaborations and interactions between objects by means of diagrams such as object diagrams, interaction diagrams, sequence diagrams, state diagrams, etc. We will explain all of these and their relationship to the overall design and performance of systems later on.

Chapter Summary

- Design is primarily a refinement of the analysis model.
- Reusability, reliability and extensibility are among the object-oriented design goals.
- Refine class selections by examining coupling, cohesion, sufficiency, completeness, quality of interface and primitiveness.
- Design Patterns are generalized steps used to solve commonly occurring problems.
- There are various language features that may be available for use such as paramterized classes, string classes and collections.

Exercises

1. Describe how you would implement a method Count() that returns the number of students in the system.

Chapter 7

System Development Processes

How do we formalize the activities depicted in Chapter 6 into a process?

What we want, indeed need to do, is to create a process for solving problems. This process needs to be well defined, repeatable, well defined and well managed and well optimized. It is very important that we distill how we develop object-oriented software into a process that we can reuse repeatedly. A repeatable process thus becomes a "pattern" for us to apply when faced with solving new problems. As we've seen with design patterns in Chapter 6, this does not mean we can't modify the process to fit our problem. As we've seen, these items are in the context of an iterative and incremental life cycle. We need to take everything we've learned about how we go about developing object-oriented software and "package" it into a Software Development Process.

What is a Software Development Process?

A software development process describes a series of activities, their pre-requisites and their resulting artifacts (products) that describe how to develop (hopefully) quality software. An overall process with these attributes is composed of multiple levels of detail.

The Software Development Process

Object-oriented design should be viewed as is an inherently interactive and incremental process. It is iterative in the sense that it involves the successive refinement of an object-oriented architecture. It is incremental in the sense that each pass through the analysis and

design steps leads us to gradually refine our strategic and tactical decisions, which ultimately yields an appropriate solution, based on the requirements.

In order to describe our process, let us examine the activities that we have done so far, using the example as our guide. We've gone through a number of steps so far. We were given requirements. That implies that at some point, the requirements for our example system were collected. At this point, we are not done. We haven't finalized our design and we have not had any discussions about what comes after. Here are the steps that have occurred so far.

- Conceptualization and requirements gathering
- Analysis
- Design

Let's examine each of these, in an attempt to formalize the activities corresponding to each.

Conceptualization and Requirements Gathering

While requirements are the cornerstone of our development efforts, the first step in our process is to get the concept of the system. This is effectively brainstorming about what the system should do, i.e. what functionality it should provide, etc. In many cases, this includes "selling" the idea to management, etc. to get funding to continue. The concept must be deemed "sound" before anything else is done. Thus, the notion of conceptualization includes having the idea for the system and doing those activities that will prove the concept sound, such as providing sufficient "expected" detail to do a cost-benefit analysis, for example.

Once we have passed the concept stage, we must gather details about the expected functionality of the system. These are the requirements. As stated before, good requirements are critical, as they provide the boundaries and the guidelines within which our software solutions must operate. This is true for both functional and non-functional requirements. From Chapter 2, we know that there are various types of requirements. This does not tell us how to actually go about capturing the requirements. A full discussion of how we gather requirements is out of the scope of this book. However, we need to account for this step as part of our overall process.

Analysis

Let's review what we did during our Analysis step. We reviewed our system requirements. We applied various techniques (nouns and verbs, behavior analysis, etc.) to select various candidate abstractions and applied our evaluation criteria to determine which abstractions would be our key abstractions. We refined these abstractions as necessary, factoring and re-factoring. We identified any relationships between abstractions, seeking to leverage inheritance, aggregation and association. Let's look at these steps in more detail.

Fundamentally, we are interested in the activities of choosing quality abstractions, etc. In practical terms, these would be activities performed by developers and/or analysts, either individually or in teams. These activities include the following:

1. Identify classes and objects at a given level of abstraction.
2. Identify semantics of these classes and objects.
3. Identify the relationships between classes and objects

Let us examine each of these activities.

Identify Classes and Objects at a Given Level of Abstraction

Purpose

As we've seen, we use this to establish the boundaries of the problem. This is the first step in devising an object-oriented decomposition of the system. In this step, we identify which real-world objects we will model in our system. Via abstraction, we represent these objects and we can focus only on the characteristics of the real-world object that ate relevant to the system. These abstractions will probably be named for the objects they represent, thus utilizing names and nomenclature from the problem domain. As a result of doing this, we are deciding what is and isn't of interest to our system, giving us, in effect, a boundary. We may not know all of our abstractions at the end of Analysis. In fact, as part of design, we may discover new abstractions .

It is also important to make sure that our classes are at the same level of abstraction. This means, it is important to be consistent across classes when deciding which details (of the real-world objects being modeled) are relevant to the system and which are not. Without this consistency, we may be unable to leverage class relationships. If we have abstractions at different levels of detail, we will have a difficult time understanding the class and ultimately object interactions.

Products

How do we track and store the abstractions and details as we go along? As you've seen, it gets unwieldy very quickly. Indeed, as problems go, our example was a relatively simple problem to solve. Ideally, we need some way of managing these abstractions that would make it easier to deal with them. As you may imagine, the need for something like this increases exponentially as the number of team members, i.e. individual developers and analysts increases. A central repository, i.e. a data dictionary consisting of all classes and objects using meaningful names is necessary for large-scale efforts. This repository would need to be continuously updated as development proceeds and would form part of the project's overall documentation.

This step is complete when we have a reasonable set of candidate abstractions. If the set is large, it may be appropriate to employ a repository even at this stage, i.e. a data dictionary.

Activities

The activities at this stage of development include discovery of abstractions, either from the requirements directly or not.

Identify Semantics of These Classes and Objects

Purpose

In this step, we establish the behavior, attributes and rules of each abstraction identified in the previous phase. We need to refine our candidate abstractions. At this point, we have candidate classes, many of which will not ultimately be included in our system. The process of establishing the semantics of each of our candidate classes will help us weed out those classes that have no place in our model. This means we have to keep looking at what the responsibilities of each class are. We have to intelligently distribute the responsibilities, based on what each class' semantics are.

These responsibilities will directly translate into the operations that we define for each class. We need to specify concrete operations (i.e. protocols) for each abstraction. The result is a precise signature for each operation. The signature of a method is the combination of the name of the method and its parameters. We also need to define the set of attributes to complement our methods, for each abstraction.

It is worth noting that our methods may or may not change as we continue to refine our abstractions (part of our incremental, iterative process). The effective distribution of responsibilities is based on repeatedly evaluating our abstraction using the metrics.

Products

- Develop each abstraction's protocol:
  - Create specifications for each abstraction
  - Write the interface for each class

Activities

Some of the main activities are below.

Scenario Walk-Through

The idea of storyboarding is to do a walk-through of scenarios involving the abstraction(s), like they do when developing television shows and movies.

We may summarize the ideas as follows:

- Select one or a set of scenarios related to an area of functionality
- Walk through the activity of the scenario assigning reponsibilities to each abstraction that is participating. Assign responsibilities that are enough to accomplish the desired behavior, based on the semantics of the class.
- As the storyboard continues, reassign responsibilities as required, so that there is a reasonably balanced distribution of behavior (as mentioned above).

Focus on one class at a time

Sometimes, focusing on one abstraction at a time can give us great insight into our overall model. Here is a summary of how we proceed:

- Select an abstraction
- Identify and list its roles and responsibilities
- Devise a sufficient set of operations that satisfy these responsibilities
- Review each operation individually. Ensure it is a primitive operation. If not, try to expose its more primitive operations and redefine into more than one operation.
- Consider specific scenarios for construction (constructor), copying (copy constructor) and destruction (destructor) (later in cycle)

- Review operations and add any other primitive operations as required.

Pattern Scavenging

The notion of pattern scavenging is akin to our earlier discussion of the importance of design patterns. Here is how we could apply this technique

- Recognizes patterns of behavior, which represent opportunities for reuse.
- Given the complete set of scenarios at this level of abstraction, look for patterns of interaction among abstractions. This might point to similarities that may be exploited in the form of inheritance or aggregation, etc.
- Given a set of responsibilities also at this level of abstraction, look for patterns of behavior. Common roles and responsibilities should be unified in the form of common base, abstract classes, etc.
- Look for patterns within operation signatures. Identifying operations in different classes with the same functionality may give opportunities for leveraging similarities between the classes.

At the end of this activity, we should have a reasonably complete, sufficient, primitive set of responsibilities (methods) for each abstraction.

Identify the Relationships Among the Classes and Objects

Purpose

In order to identify the relationships between classes and objects, we need to do the following:

- Review and strengthen the boundaries of each abstraction (based on semantics).
- Identify the collaborators with each abstraction (class) identified earlier in the detailed process.
- Formalize the physical and conceptual separations of concern among abstractions begun in previous step.
- Identify important inheritance/aggregation relationships and associations between classes.

This activity will refine the semantics and relationships of the abstractions and will serve as a blueprint for implementation.

Products

- Class diagrams
- Object diagrams
- Module diagrams
- Refinement of data dictionary

Activities

Identification and Specification of Hierarchical Relationships

- For a given set of classes, at the same level of abstraction, populate a class diagram with each abstraction's important operations and attributes
- Try to identify semantic dependencies between any two classes, i.e. if for class A to behave correctly, it must be associated with class B. If this semantic dependency exists, establish an association relationship. Establish cardinality and attributes (i.e. mandatory, optional) of the associative relationship. For each association, specify role of each participant.
- Validate decisions by walking through scenarios

Identification of Collaborations

- Identify the classes that have objects that will collaborate.
- Produce object diagrams to model these mechanisms and interactions.
- If common (i.e. similar) classes are found, leverage this similarity by implementing inheritance hierarchies.
- On a larger scale, group and organize classes into modules and subsystems

Identify Patterns that may Exist Among Classes and Objects

- Look for opportunities of inheritance relationships
- If there are patterns of structure, consider creating new classes that capture this, or refining existing abstractions. Consider classes of similar behavior as candidates for parameterized classes

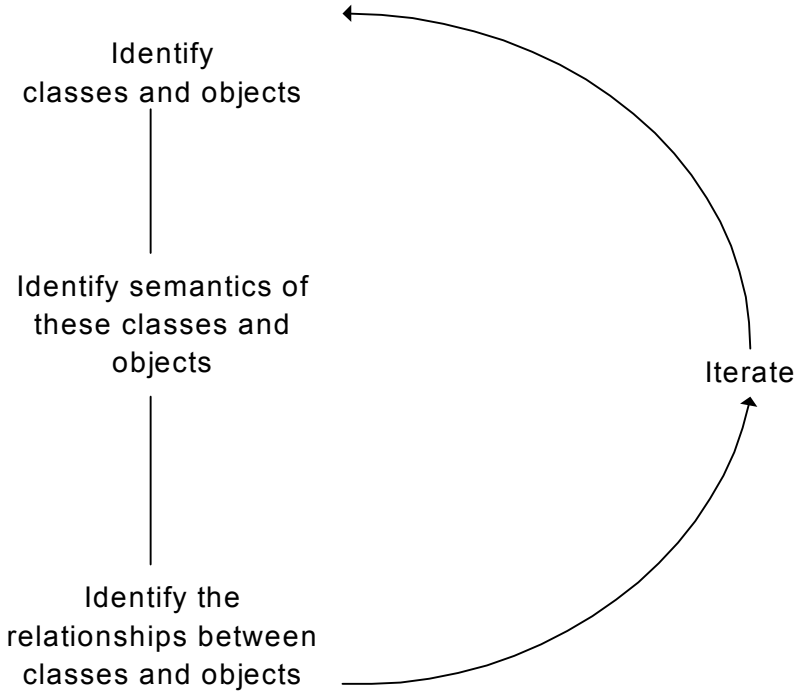


Fig 7.1 Analysis

Design

It is in the Design phase that we take the models from the Analysis phase, (i.e. class diagrams, etc.) and take them one step close to implementation. We have to factor in environment, constraints, non-functional requirements, cost, time-to-market, etc. We will produce a set of Design models that will be the basis for implementing the system.

Design is the next step in our journey toward implementation. As mentioned in Chapter 2, in the Design phase, we are specifying how the elements of the system that provide the functionality and satisfy constraints (non-functional requirements) will be implemented. In Design, we are creating the overall architecture of our application.

In order to accomplish these goals, we will need to refine and add more detail to our analysis models. This will entail repeating some of the activities that were performed in the earlier phase, but with a different focus.

Mechanisms represent patterns of behavior, i.e. interactions between collections of objects. These are typically design decisions.

During design, a developer must determine how instances of classes work together. Remember, an OO system is a collection or cooperating objects.

A Framework is a collection of classes that provides a set of services for a particular domain. A framework exports a number of individual classes and mechanisms that clients can use or adapt (e.g. Microsoft Foundation Classes (MFC)).

Our design process will include the following steps:

1. Refine classes and relationships
2. Identify environmental opportunities and constraints
3. Identify maximal levels for software engineering goals
4. Identify and employ useful patterns
5. Finalize the interface and implementation of the classes and objects

Refine Classes and Relationships

In Design, we start with the high-level models produced in the Analysis phase. As we attempt to create the application's architecture, we may find that there are some abstractions that need to be modified to properly fit the architecture[56]. The abstractions developed in the Analysis phase may need to have operations added, possibly to interact with the operational environment, etc. In some cases, as we attempt to organize our classes into modules, etc., we may notice weaknesses in the abstractions that need to be fixed before going further. These are but two of the possible scenarios that may lead to changing abstractions. It should be noted that the need to refine our abstractions is oftentimes closely related with the other activities listed below. It should not be viewed as having a single occurrence. As with everything else, it too is iterative in nature.

We need to finalize the interface and implementation of the classes and objects. This will require us to perform analysis to refine of existing abstractions sufficient to unveil new classes and objects at the next level of abstraction. Our design is a tangible representation of our abstractions, with all of the detail from our efforts to refine. We will make decisions about representation of each abstraction and the mapping of these to the physical model.

Identify Environmental Opportunities, Dependencies and Constraints

As mentioned earlier, every system has non-functional requirements that must be taken into consideration at the time of design. These non-functional requirements may represent a very diverse set. For example, some requirements force integration of many legacy databases and systems, making for a heterogeneous environment. These need to be incorporated into new development efforts. The method of interoperability selected may have an impact on the definition of our abstractions. Other requirements may dictate user-interface requirements. When confronted with this in our example, some abstractions were modified to add operations that better support the user-interface activites (`GetFirst()`, etc.). Yet other requirements may constrain our system in some way, or introduce dependencies. These must be examined for their impact on the overall design.

[56] The architecture of a system is a representation of its components their relationships, in addition to tactical considerations. See Chapter 9 for a detailed discussion of architecture.

Some environments provide opportunities for incorporating and leveraging existing classes and objects. Some of these may substitute classes already in the model. Some of these may change the interactions between some objects.

The overall environment, i.e. target platforms, protocols, libraries, development and languages will have an impact on the design as well. Again, the idea of the design is to create a set of models that represent the final step before implementation. They should describe in detail what is to be implemented and how it is to be implemented. Without factoring in the target language(s) and platform(s), there are detailed design decisions that we will be unable to make, that take direct advantage of the environment[57].

Identify Maximal levels for Software Engineering Goals

By now, we are familiar with some of software engineering's design goals, such as Reusability, Reliability and Extensibility. In order to have a good design, we need to attempt to reach these goals in some maximal way. This is because as we have to factor in environmental issues into our design. These issues may ultimately hinder how well we attain these software goals in absolute terms. In our attempt to maximize our success, we should focus on strategic decisions and attempt to limit or at least control the number of tactical decisions made.

As we saw in previous chapters, the characteristics of object-oriented development facilitate achieving our software engineering goals. Of course, this is true only if we have accurately and effectively selected abstractions, operations, interactions, how well we have employed encapsulation, etc. These will all have an impact.

Identify and Employ Useful Patterns

As we saw in Chapter 6, a design pattern is a solution to a problem similar to the one we're trying to solve. There are various design patterns that have been compiled over time, illustrating techniques and guidelines for solving many technical problems. Patterns exist at many different levels. There are patterns for solving technical issues such as managing a group of objects (containers). There are also patterns representing larger-scale issues such as architectures. Leveraging patterns starts us on the road to our solution. As we've seen, it should not be the expectation that patterns will be reused

[57] The "environment" being referred to is that which will "host" our system, once operational.

unchanged. Rather, the goal is to identify a similar pattern and make the modifications necessary in applying it to the problem at hand.

Note: The activities that we've outlined here in Design are a good representation of typical design activities. However, there may be others. One of the advantages of understanding and developing a process such as this is that it can be evolved and extended as necessary, as we grow in experience.

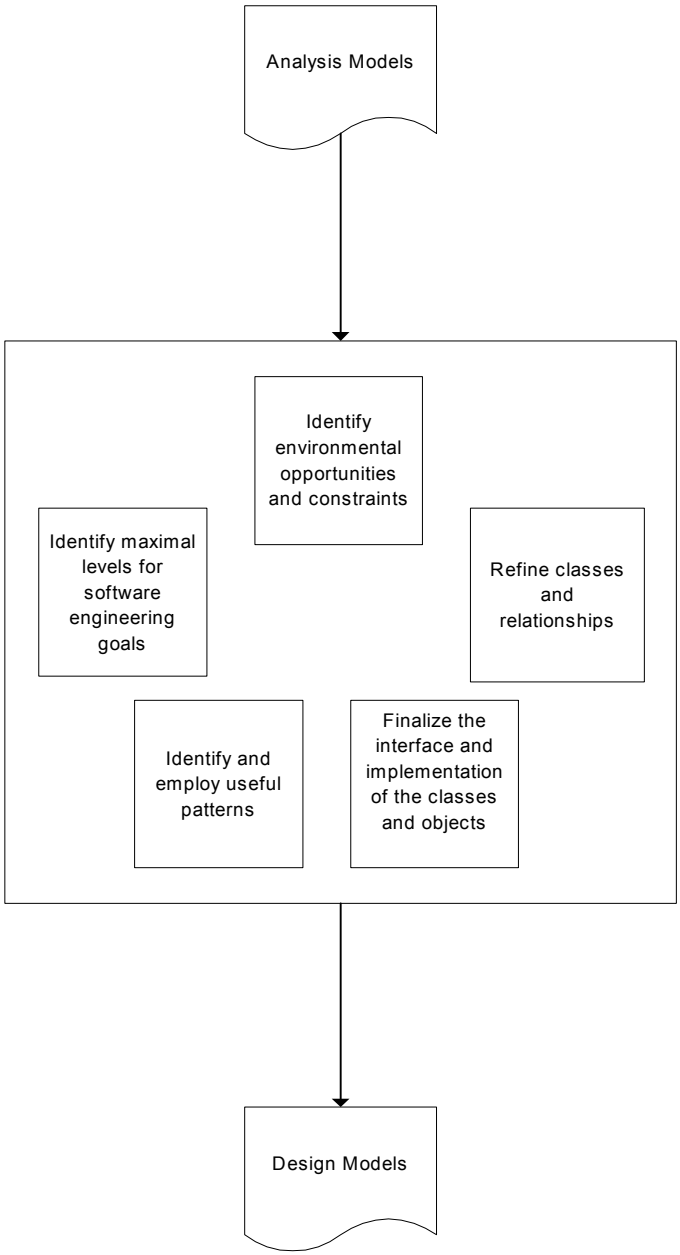


Fig 7.2 Design

Additional Development Phases

So far, we've created (and validated) the concept of the system, did our analysis and created our design. What's left? There are a number of activities that we have to undertake on the road achieving an operational system. Let us examine the additional phases below.

Implementation

The products of the design phases are models of the system that can be directly translated into programming code. In the Implementation phase, we perform this translation (and related activities) in anticipation of the next phase. In Implementation, we have to commit our detailed design to code, i.e. implement our design using one or more programming languages. For our discussion, the activities of implementation are below:

1. Implement the design: develop code, modules and components
2. Unit test
3. Maintain documentation

Implement the Design: Develop Code, Modules and Components

The primary activity of this phase is to develop code, modules and components, based on the design. These will be developed for the target environment using the target programming language(s). It is not uncommon for multiple languages to be used, leveraging the strengths of each. Indeed, in the era of web development, this is more the rule than the exception.

The architecture, (i.e. how the application is partitioned) and the prioritization will dictate where, i.e. which functional area of the system, development will logically begin.

Unit Test

A unit test represents the execution of a very isolated and localized set of test cases, usually done by one or more members of the development team. The goal is to have every unit, i.e. executable partition, of the system tested before it is used in a larger context, such as when integrated with other components or modules.

Maintain Documentation

Much earlier, the point was made that there was much iteration in developing object-oriented software. That is true here in implementation as well. Of importance is maintaining the documentation that was produced as a result of Analysis and Design.

Some of our implementation tasks may lead us to have to modify our existing abstractions and relationships. Other issues may arise that cause us to have to add completely new abstractions as well. Regardless, we must keep our models synchronized with the system being developed at all times. This is not necessarily a trivial task. To that end, some systems allow developers to reverse-engineer code to obtain models. As an aside, some systems also provide code-generation capabilities, based on the models and target language.

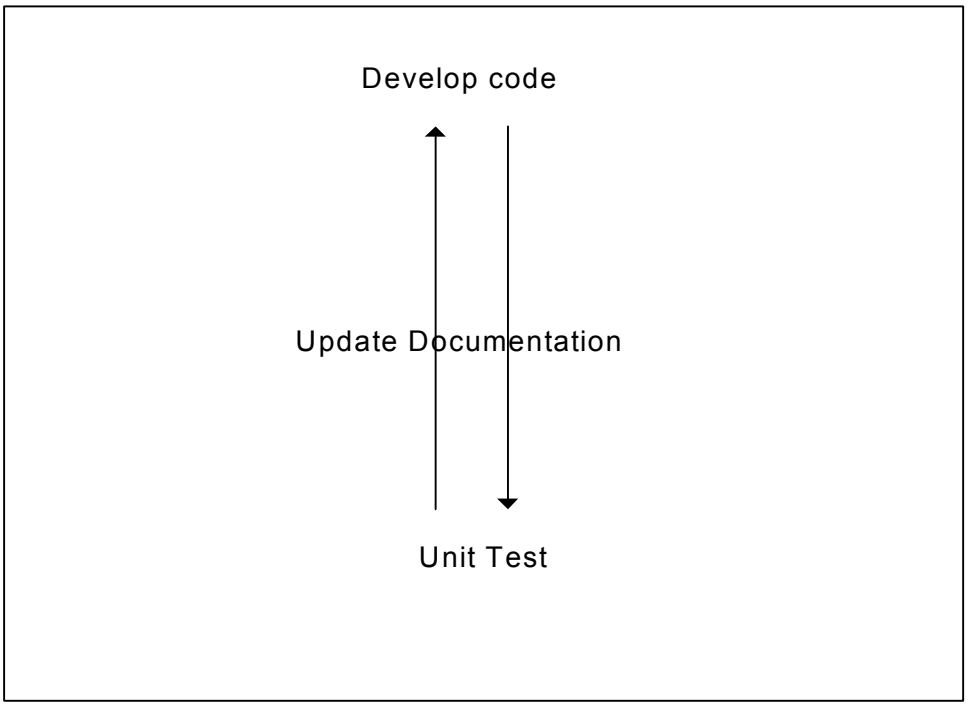


Fig 7.3 Implementation

Deployment

In the deployment phase, we move from our development efforts towards deploying our software in a production environment. Along the way, we have formal testing that must be executed, such as integration testing and user acceptance testing. Here, we are evolving the implementation through successive refinements. As before, the models should be kept in line with the system, or the documentation will be obsolete and thus much less useful.

Deployment also includes the physical installation of the software (and all necessary hardware). This will include the necessary authorizations and permissions, obtaining space in a data center, etc. This may include preparation and delivery of system documentation, including specific installation documents, architecture documents, runbooks,

manuals, etc. Strategies involving executing new and old systems in parallel, retiring old systems, etc. would be executed in the Deployment phase. At the end of this phase, we will have a fully operational system.

Maintenance

In the Maintenance phase, we are managing the evolution of an operational system (post-delivery). Once operational, there may be additional phases of development that have to be undertaken, issue resolution, brand-new requirements as a result of changing business environments, etc. The process of determining how to implement new requirements is the same as was done before for new development. However, the fact that the system is operational requires significantly more care, as compared to new development.

Why do we Need a Process?

In summary, our overall process describes various activities, partitioned into phases. These activities reflect those of an individual or small team those of the entire development team (larger scale), under the direction of a Project Manager or equivalent.

We need to define a process in order to have the outcome repeatable. We need to be able to define our activities in such a way as to be able to repeat and evolve them. As requirements and environments change, so should our processes evolve so that we can apply our experiences and any new design patterns.

We can look at our discussion as reflecting two ideas. The first is that we can partition an overall development effort into phases. The second is that within each phase, we have specific activities. In both cases, it is important for us to know how to partition and organize the effort required to develop software in an object-oriented way.

Our overall lifecycle is now partitioned into these phases, each of which embodies a different set of overall activities:

1. Conceptualization/Requirements Definition
2. Analysis
3. Design
4. Deployment
5. Maintenance

We could then use this phased approach to the lifecycle for overall project planning and project tracking.

In reviewing our software development process, we have decided to partition our process in this manner and use this terminology to express what each phase represents and also what activities occur in each phase. This is not nearly as "cut and dried" as it may seem. First, our overall process is inherently iterative. This means that even though we have seemingly sequential phases, this may not be the case in practice. This needs to be accounted for in the overall planning process. Secondly, it is much more important that there is an understanding of what activities are required for developing software, without being to "hung up" on the names used for each phase. This phased breakdown is very similar to other phased approaches we've seen – the terminology may be different. In addition, some have partitioned testing[58] into it's own phase. We have included it as a sub-phase of Deployment (in addition to installation, etc.). It is far better to understand that testing is important and decide logically where it should go, or understand why it was placed in a certain phase.

The figures below depict two views of a project plan, a Gannt chart and a tasks sheet. A project plan is a tool used to plan projects and manage timelines. The views are from a popular project planning application, Microsoft Project.

[58] With the exception of unit testing.

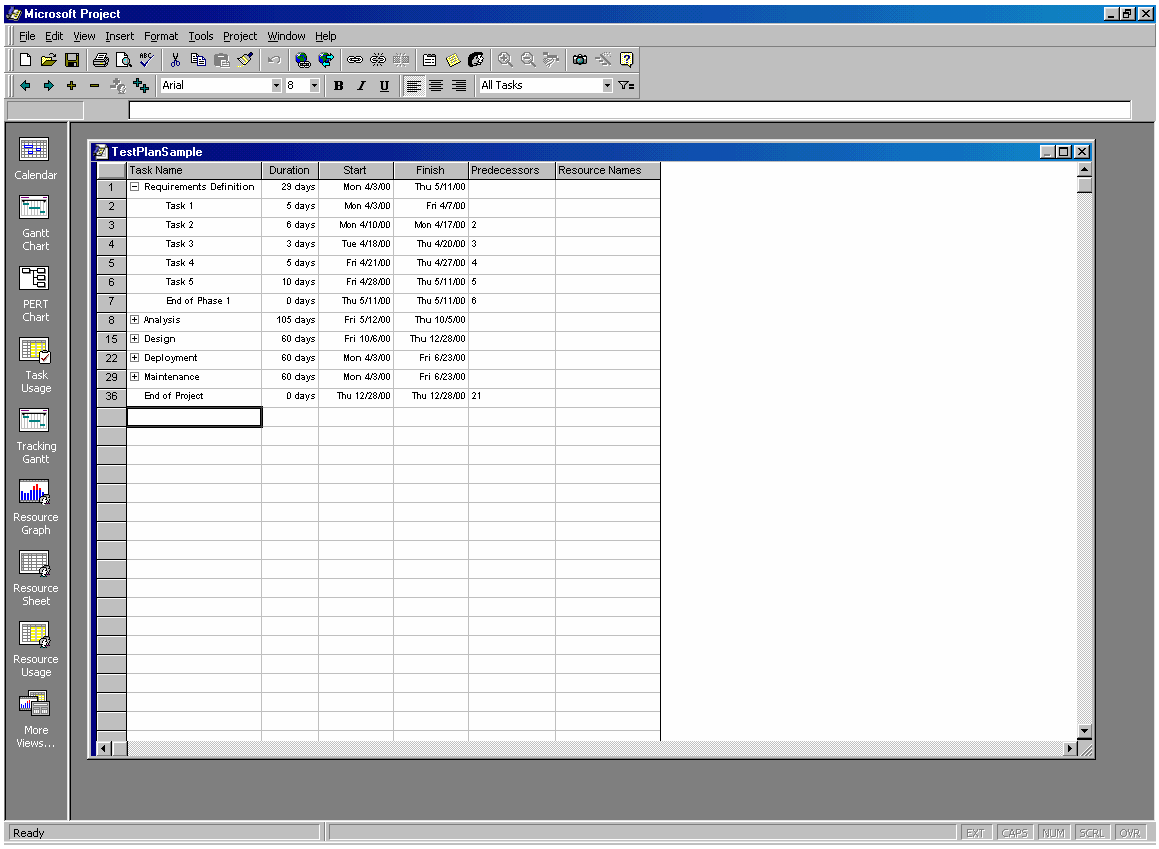


Fig 7.1 Task Sheet

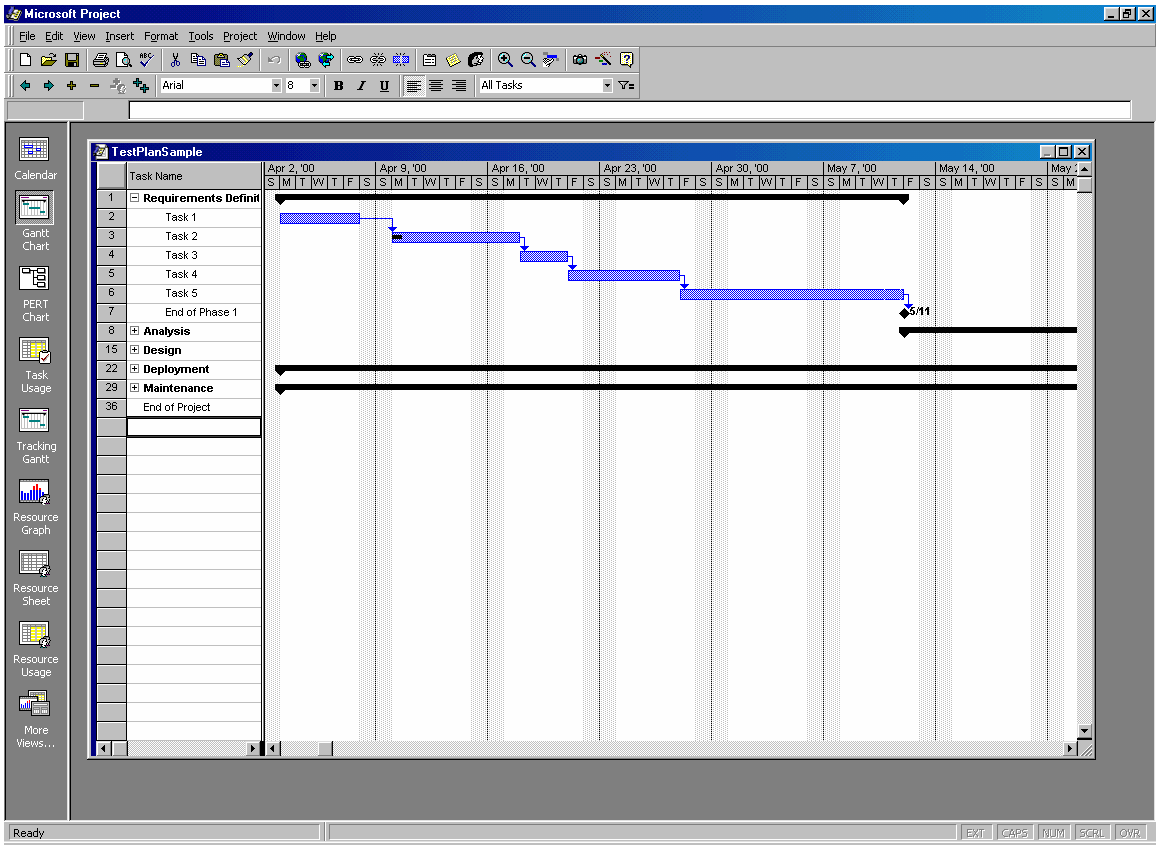


Fig 7.2 Gantt chart

Chapter Summary

- A software development lifecycle describes the activities involved in developing software.
- A development lifecycle may be separated into phases, such as Conceptualization, Analysis, Design, Implementation, Deployment and Maintenance.
- Object-oriented system development processes involve multiple levels of detail focused on selecting and refining abstractions and focused on the activities in the various phases of development, through implementation and testing.

Exercises

1. In your own words, provide a definition for a "phase" of a development cycle.

Chapter 8

Creating and Using Object Oriented Software Interfaces

We have already encountered the use of the word "interface", many times. In this chapter, we expand on that, extending the more general definition we saw earlier. In addition, we will see that interfaces can be generalized even further for use in distributed systems (chapter 9).

An interface is a set of methods that are defined in a contractual way. These methods represent a certain set of operations as required by the environment. So, an interface represents a named collection of publicly accessible methods. The interface offers no inkling as to how any of these operations is implemented. Implementation is not the job of the interface. This collection of operations (methods) is used to specify a service of a class or component. Briefly, a component (discussed in Chapter 9) is a physical part of the system comprising of one or more objects. A component implements one or more interfaces.

Practically speaking, an interface is a named collection of method signatures with the possible inclusion of constants and user defined types. The interface represents a contract that binds the client that uses the interface, i.e. invokes methods defined in the interface, and the server that provides an implementation for each method in the interface. The contract states that if you invoke a method in an interface and supply appropriate data types as parameters (input, output or either), as per signature, then you will receive appropriately typed values in return. This is all according to the signature of the methods.

This separation of interface and implementation should be familiar. Earlier in the course, we discussed encapsulation as one of the features of object-oriented development. With encapsulation, the implementation is completely separate and hidden, with the interface giving the only clues as to the capabilities of the object.

As we said earlier, the term interface is a general one. We will explore an interface as a specialized abstract construct. We will also examine an interface as it applies to the development of distributed systems. First, we will revisit the notions of interfaces and implementation.

Interfaces vs. Implementation

Earlier in the course, we defined an interface as comprised of the publicly accessible methods and fields of a class. To clarify further, this means the methods and fields that are publicly accessible by code in other classes, or more generally, code outside the class. By outside, we mean code not within any methods of the class. The interface of a class, as defined this way, is important to us because that is the only view of the class from the outside, i.e. externally visible. This is one of the advantages of the object-oriented paradigm. We are able to hide the implementation of our functionality inside our class. So, the only view of our class that others see is the set of public methods and fields that are externally visible.

Applications in OO Design

There is another view of "interfaces" in the object-oriented paradigm. This is a topic that will resonate with the Java knowledgeable among us. For the rest of us, it is a feature that we should be aware of and possibly add to our object-oriented arsenal. It is not covered in the text.

Suppose we remove the set (or a subset) of the public methods (names only) from a class place them in their own structure, similar to a class. In other words, suppose we create a new abstraction (similar to, but not equivalent to a class) that only has method names, but no implementation? What would this mean? This would mean this structure (our new abstraction) would imply some behavior. Remember, we said a class' behavior was "provided" by the methods of our class, i.e. the functionality implemented by the methods. This also means that any class that wanted to provide this functionality would implement the methods in our structure. So, we're saying, in addition to defining abstraction that become classes, we can also

create abstractions of certain behaviours of classes (i.e. methods) that form part (or all) of the class' overall interface.

In object-oriented languages that support interfaces in this manner, (such as Java), this is similar to a class definition. In Java, it is called an "interface" (surprise, surprise) and it introduces a new type to the compiler, as a new class does. However, unlike classes, the methods are not implemented in these structures. They are all abstract. This makes this structure very similar to abstract classes.

Why do we care about this kind of abstraction? There are a few benefits of being able to group sets of methods into structures such as these. For example, we can have unrelated classes all implement the same interface(s). As a result, these classes all share behaviours in common. In addition, as they introduce new types to the compiler, we may create references (not objects) which we may then use to manipulate the classes that implement these interfaces.

Hmmm. You may (correctly) say that this is similar to the polymorphic behavior we discussed earlier in the course. This is true. However, there are significant differences. For us to have the polymorphic behavior as discussed earlier, we must exploit the is-a relationship of inheritance.

Let's examine this further. Let's say we create an abstract base class with five (5) abstract methods, in addition to other stuff. As we discussed before, in order for us to be able to create any objects of our subclass(es), we must implement all five methods in the subclass. If we don't, the subclass will be abstract as well. Now, let's say we wanted to inherit from the superclass in a subclass but we wanted to (or really, only needed to) implement three of our five abstract methods. In order for us to be able to create objects of this superclass, we would have to provide an implementation for each of the other two methods, even though we did not need them implemented in our subclass. So, in order to leverage polymorphism, etc. as a result of inheritance, we have to be aware of inheriting stuff we do not need. This is really an issue of the design of classes and hierarchies. In languages that support interfaces, we have other options. We could define an interface that has the three abstract methods that are truly needed. The class we are trying to define (no longer a subclass, as these interfaces have nothing to do with inheritance) would then implement these methods if it need to. Once you decided to implement an interface, you would then have to implement all of the methods in the interface.

Interfaces, as new "types" have a few other features. Java does not support multiple inheritance but supports interfaces. C++ does support Multiple Inheritance, but does not support interfaces such as these. Hmmm again. Multiple inheritance could be a neat tool to have in one's arsenal. We haven't discussed multiple inheritance very much, other than an honourable mention early on. In multiple inheritance, we have more than one base class. The other aspects of inheritance stay the same. In Java and C#, we are allowed to implement multiple interfaces, not inherit multiple classes. In addition, as mentioned earlier, we are allowed to create variables that are references of the interface, and use this reference to manipulate objects of classes that implement that interface. This remains true even if our classes implement multiple interfaces. This gives us an alternative to Multiple Inheritance, without some of the pitfalls of inheriting from multiple classes.

Anytime Multiple Inheritance or multiple implementation of interfaces is employed, there are potential pitfalls that may arise. One of the issues that may arise with incorrect use of Multiple Inheritance is the following. Suppose you have a subclass that inherits from two superclasses, each of which has a method with the same signature defined. Due to the ambiguity, which method do we implement if they were defined as abstract? If the methods were not abstract in the super classes, which is actually called when we attempt to invoke a superclass method? However troublesome this might be, it isn't limited to Multiple Inheritance. A similar issue may arise with the multiple implementations of interfaces. If more than one interface has a method with the same signature and both (or more than 2) interfaces are to be implemented in one class, how do we resolve this issue? Don't forget, with these interfaces, all the methods in the interface are abstract. The implementation occurs in the class. Avoidance of these issues would seem to be the best defense.

Polymorphic Behavior and Interfaces

Above, we described interfaces as specialized abstract structures. While these structures are treated differently from classes (depending on implementation), we are able to take advantage of polymorphic behavior involving these structures.

In languages that support interfaces as separate structures, each definition of an interface introduces a new type to the compiler (as with classes). Thus, we are able to construct references of the

interface type that may be used to manipulate objects of classes that implement the interface. This allows polymorphic behavior as we may declare such a reference and use it in this way, without knowing the specific type of the object referred to. An example would be a function defined with its parameter being the reference of the interface type. Any reference to an object of a class implementing this interface could then be passed in as the parameter. Inside the function, the object reference could then be used to invoke any method that was defined in the interface (as in Java).

Interfaces themselves may also be part of an inheritance hierarchy. This means that an interface may inherit from another interface. The net of this is that the sub-interface (as opposed to the super-interface) would present the all of the methods defined in the sub- and super-interfaces. Any class implementing the sub-interface would have to implement all of the methods from both interfaces if it was to be concrete (i.e. able to be instantiated). As with an inheritance structure comprised of classes, we are able to exploit the polymorphic behavior that arises when we use a super-interface reference to manipulate objects of a class that implemented sub-interfaces.

Interfaces in UML

Here is an example of a simple hypothetical interface in UML:

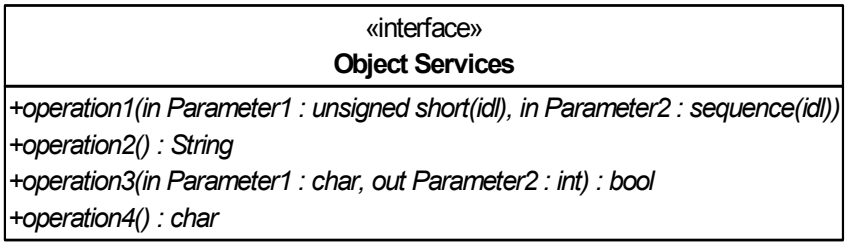


Fig 8.1 UML

In this example, we have an interface with four methods defined. Operation1 has two input parameters, Parameter1 and Parameter2. Parameter1 is defined as short, Parameter2 as sequence. These happen to be types defined as user defined types in the IDL (Interface Definition Languages (IDL) are discussed later in this chapter). Operation1 does not define a return value. Operation2 takes no parameters and returns a String. Operation3 takes two parameters: an input parameter (Parameter1 – defined as char) and an output parameter(Parameter2 – defined as int). Operation3 returns a

boolean (true/false) value. Operation4 takes no parameters and returns a character.

Here is an example in UML depicting an object that implements this simple hypothetical interface:

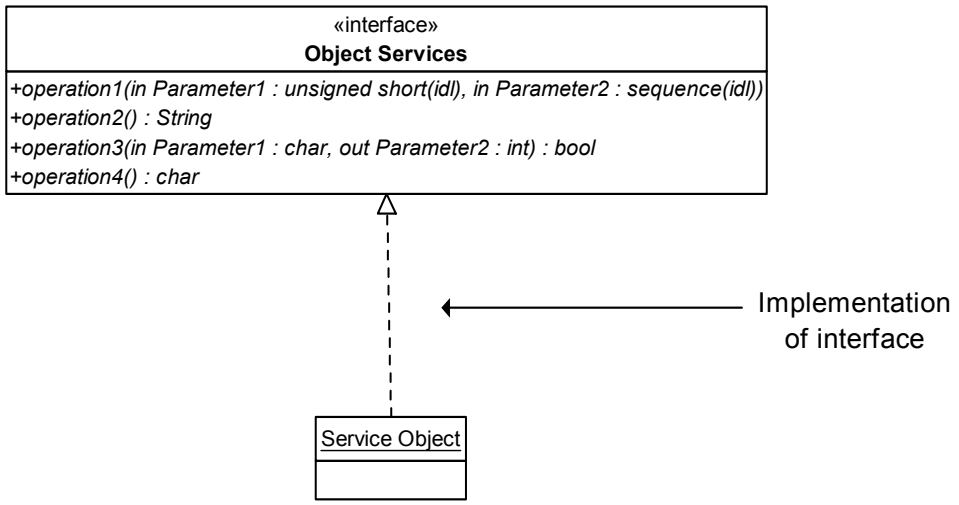


Fig 8.2 Interface illustration

In this example, the object Service Object implements the Object Services interface. The dashed line with the "open" triangle at the end indicates the realization of the interface, i.e. the implementation of the interface.

Support in OO Development

Interfaces as specialized, abstract structures, are supported by languages such as Java. However, as you've seen, interfaces are quite similar to abstract classes. In fact, in C++, which doesn't support interfaces in this way, but which does support Multiple Inheritance, we can define an abstract superclass with no implemented methods. Remember, for a class to be abstract, it only needs one abstract method. For our approximation to work, we will define a class with no implemented methods[59]. In addition, all of the methods in our superclass would have to be declared public[60]. We could then add this class as a superclass, using Multiple Inheritance. As a result, we would be able to create references (and pointers – available in C++) that we could use to manipulate objects of the classes that inherited

[59] All methods would have to be abstract to have equivalence with the Java interface construct.
[60] All methods in a Java interface are public by default.

from this additional superclass. This would effectively give us the same behavior[61]. Any class that needs to support this "interface" would inherit from our abstract superclass.

If the language supports the abstract construct "interfaces", you will be prevented from implementing any of the methods defined in the interface directly within the interface. The interface is completely abstract – you may only implement them in the class that implements the interface. In other languages, there is nothing language-specific which will prevent someone from implementing a method directly in the abstract class.

Some may argue that it may be useful to have some default behavior "built-in" to the interface definition and thus, an abstract class representing an interface, but with some methods implemented is fine. However, in order to make the correct determination, we must look at a few things. An interface in this context is a definition of various (one or more) methods to be implemented by a class. It does not represent the implementation of any of the methods. Keeping this in mind (and as discussed above), the decision will be based on the correct "factoring" of the methods and whether or not the cost of using a class hierarchy is justified, vs. an interface, whether as a separate structure or as an abstract class with no implemented methods.

Interfaces in Distributed Systems

By now, we've seen that the use of the term "interface" extends beyond its use with classes. Interfaces, with the same meaning, are also used by distributed system to define a contract between two objects or components. The term "distributed" as used here means that two objects or components, are communicating, are located on physically different hosts (i.e. different computers). In fact, these objects may be geographically distant. In fact, these object or components may have been developed in completely different languages, on completely different platforms. Regardless, as we've said before, distant or not, these object are able to collaborate to provide the overall functionality of the system. With encapsulation, we have the separation of implementation and interface.

[61] This is a limited equivalence. There is no enforcement of the class being completely abstract and all methods being public in C++, as our "interface" would be a regular class. Java's interfaces are completely different constructs and behave differently from classes. Java enforces the rules differentiating interfaces and classes.

As before, the interface defines the visible behavior of an object (i.e. public methods). Once the interface is defined, the implementation of the interface could proceed in parallel with the development of the client. The client need never know any details about the implementation of the operations defined in the interface. Indeed, as mentioned earlier, the development of the server (i.e. class(es) implementing the interface) could be in a language different from that used for the development of the client(s) and could be deployed on a platform different from the one used in deployment of the client. This is in keeping with the "spirit" of an interface. Systems incorporating distributed objects, such as those using CORBA (Common Object Request Broker Architecture – various languages, various platforms) and/or RMI (Remote Method Invocation - Java only) utilize such interfaces.

For interfaces utilized in distributed systems based on CORBA, each interface is specified using an Interface Definition Language (IDL), which is independent of the languages used for development. The IDL specifies each method, their parameters and return types. Optionally, the IDL for a particular interface may also include the definition of user-defined types. As with other interfaces, there is no implementation. Systems based on COM (Component Object Model – various languages, MS platforms only) also use an interface and an IDL to define the contract between components.
For interfaces utilized in distributed systems using RMI (Java only), the interfaces utilized are Java interfaces (as discussed earlier).

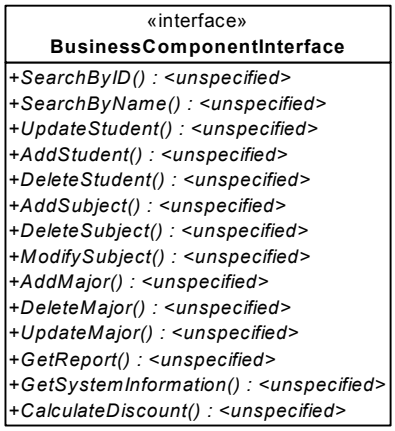
This was a brief introduction to a new concept that you should be aware of. In this course, we are creating a set of tools that will be at our disposal when we are required to perform in our role as object-oriented analysts, designers and developers. This view of interfaces introduces another tool in our toolbox. If our target language is Java, we will have a "direct translation" of this idea. If our target language is C++, we can approximate this behavior by inheriting from an abstract class that effectively defines our "interface" for any other class. We can then approximate the behavior observed in Java. These become additional design and architectural tools.

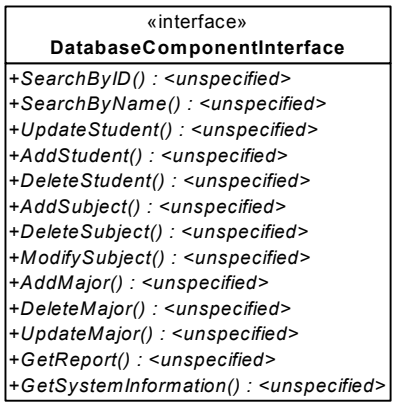
```
module FinancialInstitution {

        interface Customer {
                string GetCustomerNumber();
                ...
        };

        interface Account {
                string GetAccountNumber();
                ...
        };
        ...

};
```

Fig 8.3 CORBA IDL example

Sample Project

In the next chapter, we revise our architecture to make our example application component-based. Here's a sneak peek at the interfaces for two of our components. These interfaces describe the public methods of each component. As above, the interface does not supply any information regarding implementation.





These will be fully explained in the next chapter.

Chapter Summary

- An interface represents the visible behavior of an object, i.e. the public methods.
- Interfaces exist in distributed systems development also.
- Interfaces (as an abstract structure) support polymorphism.

Exercises

1. In your own words, explain what you would expect to happen if a class that implements an interface is:
   a) an abstract super class
   b) a concrete super class

What would the effect be on subclasses of each?

Chapter 9

Object-Oriented Software Architecture

Now that we have discussed how we identify and qualify candidate classes and conduct design, it is important that we look at these activities in a larger context. In this module, we will discuss areas such as software architecture, frameworks and reuse and how they impact (and exploit) our object-oriented approach.

In the last chapter, we discussed the term "interface". We mentioned that interfaces are also utilized in distributed systems, such as those built utilizing CORBA[62], DCOM[63], EJB[64] and RMI[65]. In this chapter, we will explore components and distributed systems, as part of an overall architecture.

What is Software Architecture?

The term "software architecture" means many different things to many different people. As a result, before we go forward, we must settle on a definition of this term that will serve as our backdrop.

We will define the overall software architecture as the collection of high-level views of the significant software components of the system. So, the architecture is comprised of different views. These views may include (not mandatory) the logical view, implementation view, process view and deployment view. Each of these depicts the structure of the system from a different perspective. Thus, software

[62] Common Object Request Broker Architecture
[63] Distributed Component Object Model
[64] Enterprise Java Beans
[65] Remote Method Invocation (Java)

architecture is also an abstraction, i.e. it describes system implementation in terms of its structure, functional decomposition into components (including their properties, etc.), interfaces, rules, constraints and communication (including protocols). In order to present this data, architects employ architecture diagrams. As an aside, the software architecture is obviously one of the architectural views for a system. For example, in addition to a system's software architecture, there is also its infrastructure architecture, describing the infrastructure components supporting the system. This depicts the hardware and communication aspects of the system, i.e. what hardware systems are deployed, connectivity (LAN, WAN, Internet, etc.), etc.

The difference between architecture and design is subtle. In architecture, we care about the interactions between our significant elements (such as components) with respect to overall scaling and performance, whereas in design, we are more granular, designing and paying attention to the detail of individual classes and components.

Object-oriented software architecture is therefore the activity described above, with respect to object-oriented systems. Unlike traditional architectures, object-oriented software architectures emphasize the placement of distributed objects and interfaces, components and interfaces, persistent objects and inter-object and inter-component communication.

Object-Oriented Architectural Elements

As outlined above, our software architecture will reflect the significant components of our system. This implies that not everything in our system is significant. In fact, in many cases, the elements of our architectures are not individual classes. In this context, some classes will be similar to "atoms", where our architecture is depicting "molecules". Obviously, the "atoms" (i.e. individual classes) are important parts of the "molecules", but if the "molecules" are the level of abstraction that our architecture is depicting, then they will be of greater interest to us.

We have used the word "component" above, in our definition of software architecture. A "component" is analogous to the "molecule" above. A component is an architectural element, i.e. it is something that is featured in (or included in) architectural diagrams and descriptions. A component is a stand-alone, (i.e. deployable), part of

a system's implementation. This means, a component is somewhat independent. A component is made up of one or possibly multiple object instances. However, as with "atoms" and "molecules" above, these object instances will possibly be omitted in the depiction of the system's structure. A component represents the collaboration of related classes, the aggregate of which provides some significant functionality of the system. As with classes, a component typically has an interface that describes the methods exported (i.e. made public) by the interface.

If we revisit the definition of object-oriented software architecture above, we can now say that the object-oriented software architecture depicts the placement of and interaction between the components of the system, each component being comprised of possibly multiple object instances.

Another term used frequently is "node". At run-time, a "node" is conceptually similar to a component. However, a node represents the hardware on which a component is deployed, i.e. a processor or device.

Designing with Components

In Chapter 6, we walked through the individual steps to get us to our object-oriented design of our system. However, good our design was, this was a trivial system for us to design. In the real world, systems are significantly more complex (understatement). In many cases, in order to provide the functionality, we need a construct that groups our classes, each of which provides some of the overall functionality. This construct that is required needs to be logical as well as physical. We need to have something that we can use to logically group classes that are collaborating. For deployment, we need to have a physical construct that we can use to manage this group of classes. Use of components also helps in organizing and deploying elements of the system. Components may be deployed in the same location, in geographically diverse locations. The use of components allows the classes that collaborate to be packaged effectively for deployment. A "package" is a container to manage elements, such as classes and components. Deploying in this way is termed "distributed". Note – we now have to expand our use of the term "object". The term "object" refers to instances of classes and components.

The objects within a component may also be grouped logically into layers. Each layer, composed of one or more objects, provides

services to its immediate outer layer. Conversely, each outer layer is a client of its immediate inner layer. In addition, layers can only communicate with their immediate neighbours, i.e. the immediate inner or outer layer.

Unlike layers, inside components, tiers are logical constructs that represent physically separate components. A client/server system has two tiers, the client and the server. In a three-tier system, the components take multiple roles. The second (or middle) tier is both a server to the first tier and a client to the third tier. In turn, each tier may also be comprised of multiple components. Generalizing this, we may have multi-tier systems, where there are multiple client/server pairs. Similar to layers, communication only exists between neighbouring tiers – we would not be allowed to "skip" or circumvent a tier.

Using Components

Components are important "building blocks" of our overall architecture. But how do we apply our knowledge thus far into designing and using components? In order to do this, we must note the following features of components:

1. Components are run-time (stand-alone) executables

2. A component is typically larger than objects. It could be an object, but it is typically composed of many objects.

3. A component has a well-defined, external interface that is distinct from the internal implementation. As we discussed in the last chapter, (encapsulation), the implementation is hidden inside the component. Encapsulation is a cornerstone of object-oriented development. All that is visible to clients is the interface. Also as discussed last time, the interface is defined in a contractual manner. Each of the component's methods is defined in terms of its signature. In addition to the operations, the interface may also include user-defined types.

4. A component is comprised of one or more objects. Each of these objects is an instantiation of a class. From a few sessions ago, we know that we may have multiple copies of an object, each with it's own state. However, the component is <u>not</u> something that we instantiate into multiple copies. We would not create multiple instances of a component, each with its own state the way we could with objects.

5. Components are usually designed with some consideration given to the environment in which the component will run. This is typically not the case when designing classes. Component designers have to consider what will "contain" their component. They may have to take advantage of services provided by the environment (or container). Component environments are typically standardized. This enables a designer to incorporate additional components into a design. This facilitates software reuse, as components may collaborate, yielding more functionally useful architectures. Components will also demand certain services from the environment and in turn have to provide certain services to the environment.

6. Components may support multiple interfaces. In fact, some components support an interface that allows other components to supply a query that returns the name of each interface implemented by the component.

7. Components may be considered lightweight or heavyweight. A lightweight component is one that relies on external software to accomplish its primary tasks. A heavyweight component includes all of the services it needs to operate in a given environment.

8. Components represent physical "packaging". Classes represent logical abstractions. Components live in the physical world, i.e. at run-time. Classes do not.

Given these features of components, which objects should be packaged in to a component? The answer depends on many, many variables. In earlier sessions, we discussed class design, i.e. what abstractions make good classes. As part of that exercise, we discussed the metrics such as completeness, primitiveness, etc. We said that the class should have a set of methods representative of the implied functionality and that there should be as many as required to give a complete view (cohesiveness and completeness). With components, we have to take a similar approach. A component is comprised of possibly many objects. However, a component is viewed from the outside as an architectural unit. Thus, a component provides a "block" of functionality, i.e. some particular behavior. As such, all of the objects that are included in the component should contribute to this overall functionality.

As you can see, this is similar to the evaluations that we have to do for objects. This is as it should be, as we should not have "weaker" object-oriented designs as a result of using components.

Components and Distributed Systems

As with objects, at run-time, components are also collaborating to provide the overall functionality of the system. However, with each component a stand-alone executable, we are able to deploy components on different platforms. Systems deployed in this manner are termed "distributed".

Earlier in this section, we mentioned that unlike objects, components have to interact with their environment. Thus far, we have discussed components in the context of providing an implementation for the operations in the interfaces only. However, in order for components to be deployed on different platforms and interact with other components and objects, there needs to be some lower-level services that are provided which would enable or facilitate this communication. Name and directory services, obtaining access to remote methods, translation and transmission of parameters and return values, security and license services, etc. Imagine these in the context of two platforms as different as a mainframe and a PC. Assume that there are network protocols and media to consider in terms of connecting the two platforms. In addition, other services may be necessary, such as distributed transaction processing, persistence and the ability to recover from failures (fault tolerance), etc. The need for these services arises from the increasing complexity of modern day systems and the need for these systems to be reliable. These are services typically provided by the environment in which the component runs. If each component designer had to implement all of these services, a distributed architecture would be near impossible to achieve, not only because of the added complexity of each component and the added work of creating the component. In addition to these, each component designer could possibly implement these in a proprietary way. If they did, there would be no interoperability.

Each component designer is more able to focus on the functionality the component is provided, instead of that and everything else. In fact, the expertise that would be required to develop "everything else" is decidedly non-trivial. Since the environment provides services such as these (and others), the environment needs to be able to communicate with each component. The environment, to some degree, has to keep tabs on each component, registering the component when it starts,

being aware of its lifetime and its destruction. The environment may allow a component to be started once it recognizes a request for an operation defined on an interface implemented by the component. The environment needs to interact with the component. As a result, there may be additional, separate interfaces that need to be implemented by the component. These additional interfaces (one or more) could be comprised of methods called by the environment. In addition, there may also be methods provided by the environment that have to be called by the component to keep the environment aware of the component's state. The environment may provide "factories", which are able to launch a component once a request for an operation on that component arrives.

The environment may utilize various strategies to achieve the goal of secure and reliable component execution. Enterprise Java Bean (EJB) containers (EJB = Java component) are an example of an environment in which components execute. Servers such as BEA WebLogic and IBM WebSphere are EJB containers. Based on the EJB container specification published by Sun Microsystems (provided a standards based environment), EJB containers such as these provide the services mentioned above (among others) to each component they host. This frees the component designer to focus on the business functionality they need to implement. CORBA implementations, such as Iona Orbix provide similar services for CORBA components written in many different languages. In each case, the "container" or environment is based on a standard. This allows the reuse of components developed by others, as long as they conform to the standard.

In addition to the design of the components and their interactions with their environments, there is another aspect to consider. Components are deployed onto nodes. The deployment strategy could have impact the system's overall performance greatly. Decisions have to be made about what processor or device configurations to use, in addition to where geographically they should be located. There is also potential for the strategy to include component or node clustering. In clustering, we have multiple instances of the same components on different nodes. When a request comes in, one of the nodes satisfies it, based on a random selection, a round-robin selection, where each node is selected one after the other in a cyclic fashion or load-balanced, where the node with the least workload is chosen. Each of these options has positives and negatives. The environment may also supply these or similar services. Again, decisions made regarding these areas may impact system performance greatly.

Components in UML

As described above, software architecture depicts one or more views of a system. This depiction is typically via diagrams. In UML, we can use a deployment diagram to model our architecture. This diagram will describe the components and nodes of the system with the connection between nodes and components. A simple deployment diagram is as follows:

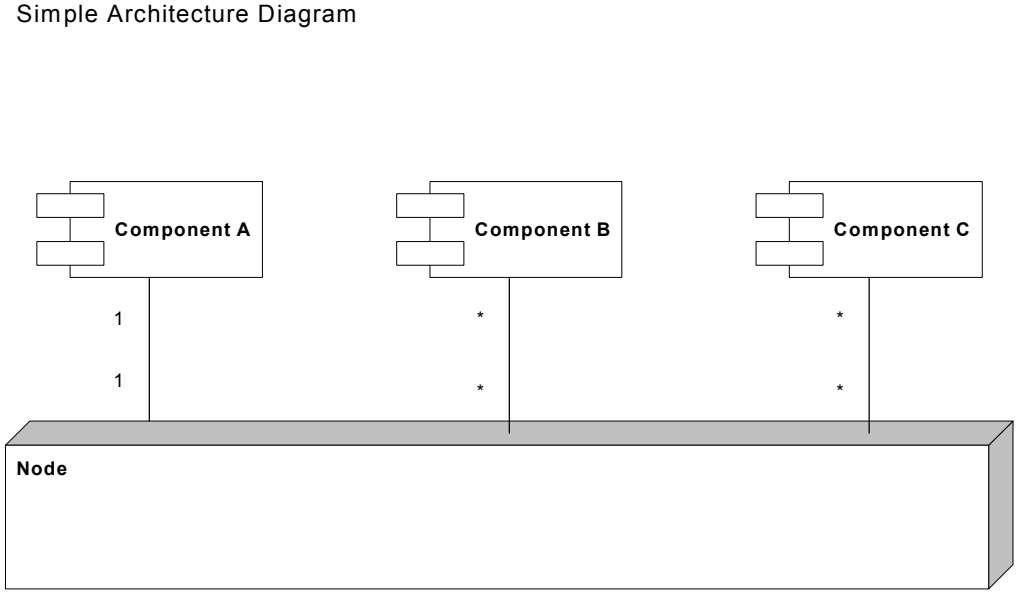


Fig 9.1 UML components

In this example, we have one node (a "server") and three client components.
Even though we have essentially equated "node" and "component" for the purposes of this chapter, strictly speaking, a node may consist of many components. A node represents a processor or device on which components may be deployed. To capture the case in which multiple components are deployed on the same node, we could modify the previous diagram as follows:

Simple Architecture Diagram with multiple-component node

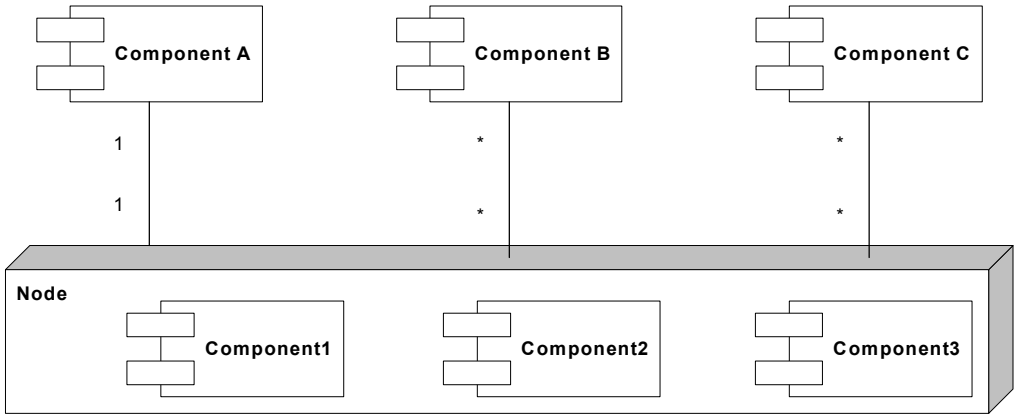


Fig 9.2 Component interactions

The node here could represent an EJB server ("container") or some hardware on which multiple CORBA or COM+ components were deployed.

Let us modify the example from last time, changing the implementer of the interface from an object to a component as follows:

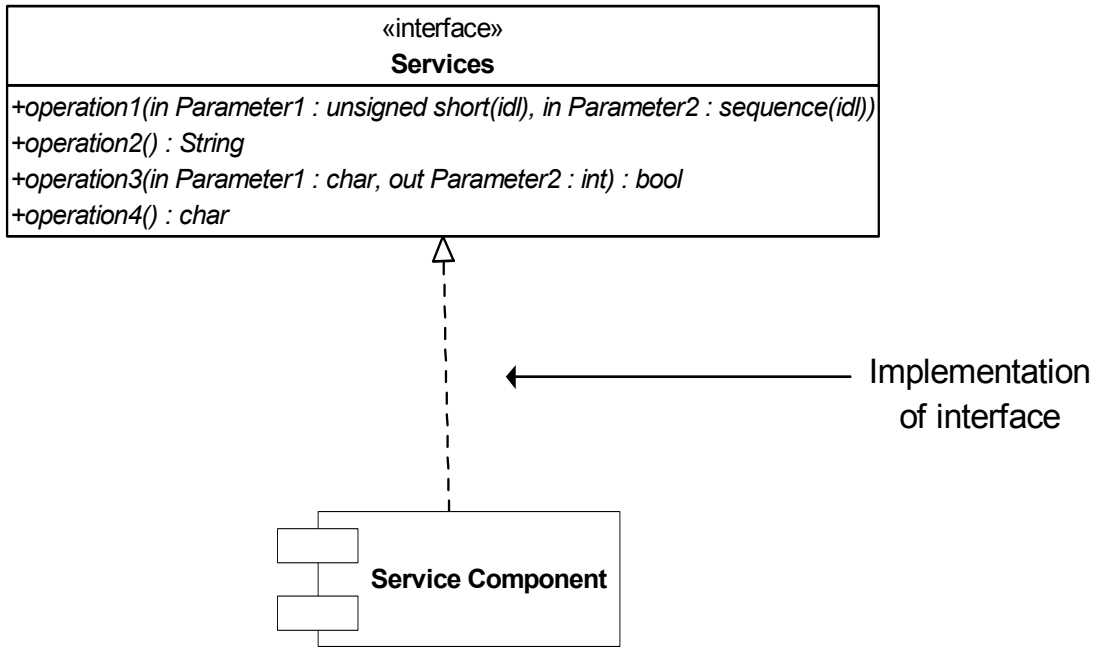


Fig 9.3 Component interactions

In this example, the component "Service Component" implements the interface "Services". The operations are the same as presented in Chapter 8.

There is a "shorthand" version of this modeling that may also be used. We may capture the API (application programming interface) of a component as follows:

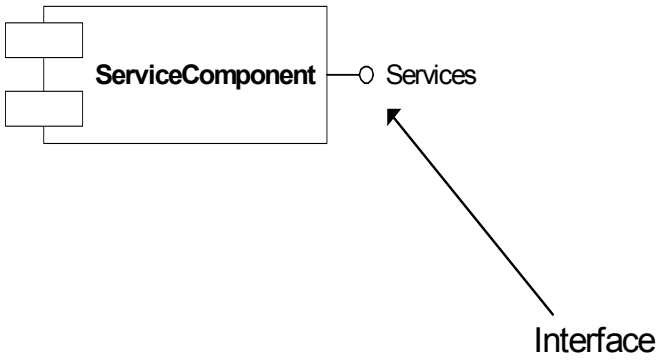


Fig 9.4 Component interfaces

The horizontal line with the circle at the end represents an interface that is implemented by the component. Obviously, the details of the interface, i.e. the operations contained therein, are not apparent using this form.

Using this shorthand method, we may model a component that implements multiple interfaces as follows:

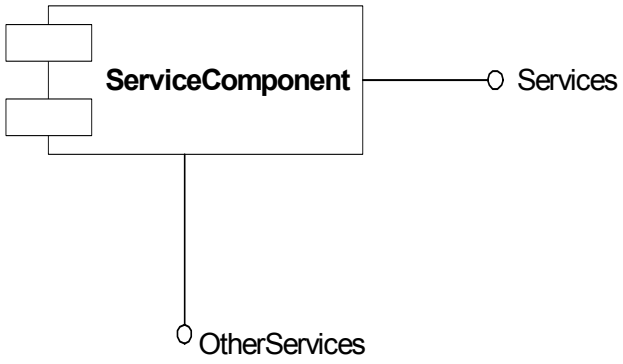


Fig 9.5 Multiple interfaces

Sample Project

In Chapter 6, we progressed through our design phase with the underlying assumption being that we were building a single executable running on one platform. Everything would be in one executable, with the exception of the actual data in our relational database. Another realization is that everything would be running in the same process, including the calls to our database.

Suppose we decided to "component-ize" our simple example. How would we change our existing architecture to partition our system into components? Where would we draw the line?

Informally, we already have three software layers at work in our application[66]. The user interface is one layer (presentation layer), the validation rules, etc. are another (middle layer) and the database interactions are a third (database layer). As a result, we could physically separate our applications into three distinct parts, running on three separate platforms. Each of the three parts would correspond to one or more components. We would redesign our application to become distributed.

One of the major activities when designing with components, other that deciding what components will exist, is defining component interfaces. From our discussion above, we know that the interface will represent the functionality provided by the component. We also know that a component may include many classes, so the component's interface may not "match" any single class' interface. Let us revise our architecture, identify our tiers and discuss the functionality and interfaces.

[66] Since everything is in one executable and not distributed, we use "layers" instead of "tiers", as discussed earlier.

Architecture

Let's begin by describing the tiers included in our architecture.

Tier 1
The first tier would be for presentation. This tier is responsible for presenting the user interface. This means, this tier would be responsible for communicating with the user. Tier 1 would be a client of tier 2, our middle tier that has our "business" logic. As we mentioned above, Tier 1 would only communicate with Tier 2.

Tier 2
Tier 2 contains all of our "business" logic. This includes all of our validation routines, calculation routines, etc. This layer accepts inputs from tier 1 and translates them into calls to Tier 3. This is reversed for the trip back. Tier 2 is a client of Tier 3 and provides a service for Tier 1.

Tier 3
This tier is provides all database services. This remains the abstraction of the physical relational database.

Interfaces

Tier 1 - Presentation

We do not need to define an interface for Tier 1. Tier 1 will not have any clients. Tier 1 will be a client of Tier 2.

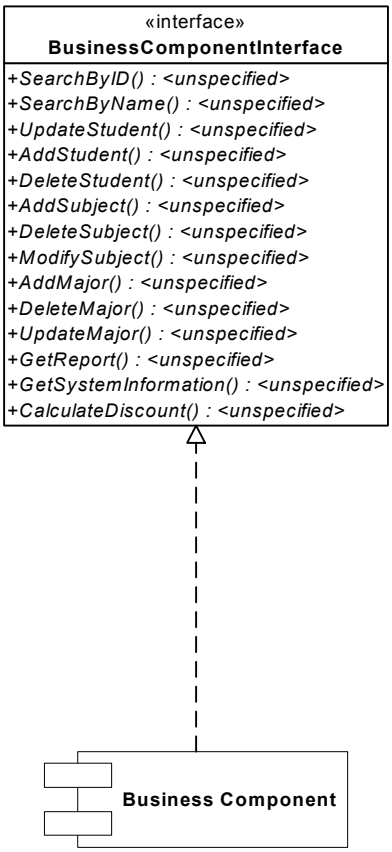
Tier 2 - BusinessComponent

The interface to Tier 2 (BusinessComponent) needs to support all of the methods that the presentation layer would need to invoke in order to present information to the user and to accept information from the user. Let us look at the objects that currently service our presentation layer. Tier 1 can only interact with Tier 2. So, the interface for Tier 2 needs to be able to support all of the requests needed by Tier 1. This includes the following:

- Adding new students
- Modifying students' information
- Deleting students
- Searching for a student by name
- Searching for a student by ID
- Obtaining data for reports
- Maintaining the overall list of subjects

- Maintaining the overall list of majors
- Obtaining system information

Even though our example is simple, there is a bit of work left to do to refine the interactions between the objects in our system. This will obviously have an effect on our new architecture as well. With that in mind, let's define some general methods in our interface for our component. We can diagram our interface for Tier 2 as follows:

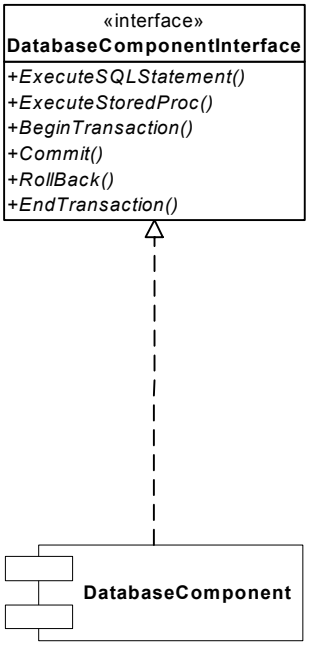


The return-types of the methods are left as <unspecified> as this is a level of detail to which we have not progressed sufficiently.

Tier 3 - DatabaseComponent

In the example interface for Tier3 (DatabaseComponent), we see that the methods defined there are defined to support the activities undertaken by the BusinessComponent.
For the sake of the example, we'll take a very simplistic view of our component and its interface. Let's assume that the interface supports executing various SQL statements, in addition to managing transactions. This means, the BusinessComponent would be responsible for creating the SQL statements that would be executed by

the DatabaseComponent. We've also included a method "ExecuteStoredProd" to indicate that we may implement some of our SQL operations as stored procedures physically on the database server. The interface to the component is below.



Here as well, the return-types of the methods are left as <unspecified> as this is a level of detail to which we have not progressed sufficiently.

Chapter Summary

- Architecture describes the collection of high-level views of the significant software components of the system.

- Components are the building blocks of an overall architecture. They represent one or more objects and run as stand-alone executables.

- Components are well encapsulated. They have a well-defined interface that is distinct from its implementation.

- Components may be deployed on different platforms.

Exercises

1. When would an instance of a component be created?
2. When would you have multiple instances of a component?
3. Compare two objects collaborating in the same program, running on the same platform to two components running on different platforms. List (with a brief explanation for each) three issues that the component-based system will have to overcome that the "single-program" system will not face.

Chapter 10

Object-Oriented Methodology in the Industry

In the last nine chapters, we have discussed object oriented analysis and design. In the industry today, object-oriented development has a strong foothold. There are now many environments that support object-oriented or object-based development. In addition, as we will discuss below, a specialized process for developing object-oriented development has been produced and is available – the Rational Unified Process.

Object-oriented development does not exist in a vacuum. In the typical enterprise, there are many platforms – one or more mainframe systems, a few to many midrange systems which could be Unix based (any one of the flavours) or AS400 based. In addition, there could also exist a plethora of microcomputer-based systems, running various versions of Windows, with the possibility of some Macintosh systems as well. To this mix is usually added various databases.

Increasingly, new system requirements dictate that there needs to be software that is able to aggregate the various pockets of business intelligence that exists on the different platforms. In many cases, the enterprise is also looking to further leverage their existing platforms for better returns on their original investments. This increase in complexity affects software development in another way as well. Software is constructed based on requirements. Another way of looking at this is to say a system will only be as good as its requirements were defined. One of the largest problems facing software development efforts is the issue of incomplete or ill-defined requirements. If the requirements are ill-defined or incomplete, then the project runs the risk of delivering software that is unusable to

some degree. We've all seen the old cartoon that depicts the sharp contrast between what the users wanted and what the systems development team delivered. In addition, if the requirements are incomplete, then the scope of the project could keep growing (scope-creep). This will severely limit what is delivered, if anything, for the project will have to keep expanding to address new requirements. The strategies for combating these issues could be a book in itself, as there are many, many variables involved in managing the requirements for a non-trivial project. Indeed, there are books dedicated to this topic.

Coupled with the increasingly shorter development lifecycles (they always want it yesterday), you can see what a distressing state of affairs could result. Nowadays, leveraging Information Technology (IT) (representative of development, infrastructure etc.) is seen as a competitive advantage. It is no longer seen as only a business expense. In order to provide a better customer experience, reduce costs, improve turnaround time, speed time to market, etc., enterprises are increasingly putting pressure on the IT departments to produce. For this reason, IT managers are looking for ways to improve the IT return on investment (ROI) and to make their senior management happy. This includes addressing issues of interoperability, leveraging existing platforms, more easily maintainable code, to name a few.

Object-oriented development methodology (inclusive of component-based systems methodology) can come to the rescue. We list a few specific examples below.

Requirements Gathering

This is not object-oriented per se. However, utilizing use-cases may help to alleviate some to the issues that arise when other requirements gathering strategies are used. A "use-case" is a scenario from the user's perspective. This means that it uses the language familiar to the user and describes some aspect of the overall functionality. So, the user requirements for the entire system will be comprised of many use-cases, each of which describes some facet of the system. The details of use-cases are beyond our scope here. However, as each use-case details some aspect of the system (including positive and negative cases) in the language of the user, it may be easier to define the appropriate scope, as opposed to using a functional specification document. In many cases, it is sometimes difficult to get an overall view of the behavior of the system from the

functional specification. By its very nature, a use-case gives you a view of what the expected behavior of the system should be.

Code Reuse

There are many aspects of object-oriented development that have an impact on code reuse. Let's think about this. From Chapter 1, we know that the major aspects of object-oriented development are the following:

- Abstraction
- Hierarchy (inheritance and polymorphism)
- Encapsulation (information hiding)
- Modularity
- Persistence

Let's look at the major aspects again, in the context of code reuse, as follows:

Abstraction

An Abstraction is a representation of a more complex structure by a simpler one that emphasizes only the elements of the more complex structure that are deemed relevant in the context of the particular system being designed. We've seen where it is critical to choose the correct abstractions for our systems. These abstractions become the classes that are the "building-blocks" of our design. We've also seen that it is critical that we evaluate the quality of our abstractions. How does this apply to software reuse? We may reuse our code in a few ways. For example, if we have classes at lower levels of abstraction, meaning they provide relatively narrowly scoped functionality, we may use objects of these classes to provide this functionality wherever it is needed. If we design relatively monolithic classes that include "everything but the kitchen sink", i.e. at a higher level of abstraction, it is less likely that we will effectively reuse these classes as we may not need exactly the same behavior again. We may need a slightly different behavior, which we would be unable to easily obtain because of the design of the class. So to promote reuse, we would want to concentrate on appropriately scoped classes. Since we have a collaboration of objects at run-time, not necessarily one or two, we can appropriately factor the overall functionality into appropriately sized classes.

Hierarchy

Hierarchy and code reuse go together very well. If you remember, this aspect of object-oriented development covers Inheritance,

Composition, Aggregation and Association. The underlying idea utilizes the discussion of appropriate abstractions (correctly scoped) outlined above. If the abstractions (i.e. class definitions) are at the correct levels of abstraction, then we may create new classes based on inheriting, aggregating or associating previously defined classes (or any combination of the above). With Inheritance, we directly extend previously defined functionality, adding the specific functionality that we need. We have access to the functionality of our super class (or classes) in addition to the functionality of our sub-classes. We can create increasing complex inheritance hierarchies. Of course, with polymorphism thrown in, we may use existing code with new subclasses. If we do not want to use Inheritance, we may still employ Aggregation or Composition. With these, we may use the "restaurant menu" approach, i.e. pick objects from existing well defined classes that we need, bringing them together and assembling them to get the functionality we need. Whichever approach is chosen, well-defined classes are the key. From a testing standpoint, we also benefit, as if each of our "building blocks" has been tested and certified, the testing will center on either our sub-classes (if we used Inheritance) or the class that results from assembling these objects (if we used Aggregation or Composition). These are definitely very high on the reuse scale.

Encapsulation

Encapsulation describes the separation of the interface from the implementation. As we saw in Chapter 8, this applies equally to objects and components. How does this facilitate software reuse? There are two perspectives to consider – the client that invokes an operation defined in the interface and the object (or component) that implements that operation. With code reuse, we are looking to be able to reuse or re-deploy previously developed software. As we've discussed previously with abstractions, we have a similar approach we have to take with interfaces as well. Interface, as with abstractions, have to be well defined. Having multiple interfaces with similarly defined operations is not the only issue. Having to have multiple similar implementations is a result of this. The interfaces need to be well defined so that the objects or components that implement their operations will not be redundant.

Modularity

Modularity would seem to have an obvious impact on code reuse. Modularity is the ability to decompose a system into a set of collaborating objects. As before, each object is a specific instance of a class. The ability to have an operational system comprised of cooperating objects is Modularity. Modularity, as it applies to code

reuse, describes the ability for use to have multiple objects collaborating, even if that included multiple instances of the same class, each of which was acting as a server providing the same functionality, but to multiple clients. There would be no conflict between these object (static variables not withstanding).

Components and Reuse

Everything we have discussed thus far may be applied to components as well. If we have a well-defined component that provides some functionality, we would be seeking to employ this component wherever we required this functionality. As with objects, it is therefore critical that components, though consisting of one or more objects, be at the correct level of abstraction as well. A relatively monolithic component will tend to be less effectively reused than one that is defined more granularly.

The important thing to remember is that a component is independently executing. Components should be designed to preserve this fact. In addition, the components should have clearly defined functionality. The "containers" that support component-based development also support multiple client connections, again promoting reuse.

Code Maintenance

Code maintenance has been the topic of many a debate. In a nutshell, systems are rarely static structures that never change. On the contrary, in many cases, there are significant changes that may arise over time. These may be due to changing market conditions or regulatory requirements, to name two. As a result, software needs to evolve. Unfortunately, many people attempt to address this fact of life when the changes need to happen. By then it is usually too late. Maintainability needs to be designed in. Object-oriented development has some features that facilitate this. Let us revisit the major aspects of object-oriented development in the context of code maintenance.

Abstraction

The selection of quality abstractions will impact code maintenance as well, for essentially the same reasons as listed above. If the functionality of the system is "factored" correctly into classes, then changes to one class should not affect other unrelated classes in the system. In fact, this should be a design goal. Coupling, as we discussed in Chapter 6 will have an impact. Loosely coupled classes will be less affected by changes made to one class. Unnecessary coupling and dependencies between classes lead to code that is less easy to maintain.

Hierarchy

Some of the features of Inheritance facilitate code maintainability, while some do not. For example, if I have an inheritance hierarchy in place and I need to accommodate a new sub-class, then I only have to add my new sub-class, not change any of the existing classes. In fact, if I am exploiting polymorphism, I may not have to change any existing code at all. In addition, if I have to add new functionality that is to be available to all classes in my inheritance hierarchy, I could easily add this new functionality to the super-class. In general, once it is in the super class, it is available to all sub-classes. However, if I have to make changes to a super-class or re-factor the functionality in any of the existing classes in my inheritance hierarchy, the task becomes very different. This is because inheritance hierarchies are inherently tightly coupled. This means changes to super-classes may cause existing code to break. Before, we noted we could take advantage of adding code to the super-class. This is a "double-edged sword". Such changes must be made judiciously. In essence, inheritance hierarchies need to be well-defined also. The correct decisions must be made about the definition of the super-classes or super-classes. If not, the maintainability of the system may be compromised. Aggregation and Composition, being loosely-coupled do not have the same issues. With Aggregation and Composition, we seek to be able to make modifications by changing the assembly of the building blocks with as little effect as possible.

Encapsulation

The separation of the interface from the implementation has many direct benefits. Encapsulation dictates that "clients" will never have access to the implementation details of the "server", whether object or component. Therefore, once we define the interface, "clients" may include code to invoke the operations defined in the interface. However, due to Encapsulation, we may change the implementation of these operations at any time with no effect on the clients. Keeping the interface constant means not removing or changing the signature of any methods defined in the interface, or removing the interface itself.

Modularity

Here, we see the benefit of Modularity in terms of having multiple objects with concurrent lifetimes, as they are aggregated etc. into providing the newly required functionality.

Components and Maintainability

The ideas expressed above may be carried over to components also. We may look at maintenance in terms of how easily we are able to

maintain the objects that comprise the component. We may also look at maintenance in terms of what functionality the component "serves". In either case, the discussion above, if extended to components, holds.

Object-Oriented Technology at Work

Object-oriented technologies are showing up in many places these days. There are many tools and technologies that have been developed, each of which is becoming more accepted in the industry. We highlight some of these below.

Rational Unified Process

Development methodologies are not uncommon, in the least. However, in recent years, one development process has come to the fore in the industry. It is a process developed by some of the people at the forefront of object-oriented development. It has many aspects taking into account the detailed development processes, as discussed in Chapter 7. What follows is a brief introduction to the Rational Unified Process.

Taken as a whole (and if applied correctly), we see that the object-oriented approach to systems development is an effective tool that may be used for complex systems development, in addition to having features that may address some of the largest challenges that we face when undertaking non-trivial software development.

Rational Software as put forward a software development process termed the Rational Unified Process (RUP). It seeks to address many of the deficiencies in general software development methodology, resulting in the achievement of process goals as described in our discussion of the development process in Chapter 7. The RUP outlines the following:

- Guidance for ordering team activities
- Specification of which artifacts should be developed and when
- Tasks for individual developers
- Tasks from the team perspective
- Criteria for monitoring and measuring a project's products and activities

The RUP is geared to producing quality object-oriented software using a well-defined process that is repeatable. It employs an interactive

developmental approach, as opposed to the traditional "waterfall" approach.

The RUP is developed and maintained by Rational Software and integrated with its suite of software development tools. The RUP also represents a process framework that can be adapted and extended to suit the needs of an organization.

The RUP embodies many of the concepts and software best practices we have discussed thus far. These include iterative development (as we've discussed earlier), requirements management, architecture and the use of components. In addition, the RUP uses a use-case centric approach, where use-cases define the behavior of the system[67]. The RUP also includes quality of process and product, change management, process configuration and tools support A large part of the process is the development and maintenance of models of the system, using UML. UML is used to express the artifacts required by RUP.

Rational Rose

Rational Rose is Rational Software's tool of choice to capture, manage and display the models created in the RUP in UML. Rose also allows the generation of code from models and the generation of models from code, thus making it easier to keep your code base synchronized with your design. This will also allow your system to be able to evolve from the code or from the design or both.

Let us look at some areas in which object-oriented technology has a presence.

Object-Oriented Databases

In Chapter 6, we integrated a relational database management system into our object-oriented system. While this is obviously possible (and done every day by developers), it is requires us to map our objects into a two-dimensional set of tables. As we have seen, there isn't always a direct correlation between classes in our model and tables in our database. In addition, relational databases have certain rules. In order to produce a join, there must be keys in common, i.e. the primary key of one table must be a foreign key in the other, etc. This is not a constraint of the object model. This is a constraint of the relational database. How may we efficiently persist and query object

[67] Use cases are not required.

systems without having to do this translation between object and relational models? Is there a better way?

Object-oriented databases were introduced in the mid-1980's to address these problems. They were designed to store and manipulate objects optimally. Instead of focusing on the data model, as all relational systems do, object-oriented database management systems focus on the object model. Based on this, object-oriented databases are able to manage very complicated models with complicated relationships. They are able to manage many different kinds of objects, as defined by the model.

Generally, a library of routines is supplied with a particular vendor's relational database management system (RDBMS). This library allows us to interact with the database. In general, we are able to invoke those routines, passing them SQL statements[68] for execution. Some of these libraries may be vendor and database specific, applying only to that particular database. These may accept extensions to SQL that are specific to that vendor. Others, like ODBC, provide support for a standard way of querying a relational database. They are still vendor specific, but the only support the SQL syntax that is "portable" across relational databases.

With object systems, we have a similar situation. Object databases are optimized for the transfer of objects between client and server. This access is provided transparently through an object manager supplied by the vendor. This provides the necessary navigation and management. The object manager interface includes operations to manage transactions, execute queries, etc.

As with relational databases, object-oriented databases have standard languages defined for them as well. Object-oriented databases have Object Definition Language (ODL) to specify how an object model is defined in the database, Object Manipulation Language to specify the application-object manager interactions to manipulate objects, and the Object Query Language to specify how applications query object-oriented databases. These are analogous to relational databases' Data Definition Language (DDL), Data Manipulation Language (DML) and SQL for querying.

[68] In many cases, the ability to create tables, drop tables, etc. are also allowed via these routines.

Chapter Summary

- Object-oriented development seeks to remedy many of the issues in the industry, such as software quality and code reuse.

Appendix 1

Use Cases[69]

Simple statements are useful for capturing and presenting performance, hardware, deployment and usability requirements, etc. However, it is difficult to use simple statements as the only means of capturing functional requirements. These requirements describe how the system behaves in response to user and external system inputs[70]. Use cases are an excellent way to capture and express a system's behavior.

Use cases are a formal way to capture the interaction between those providing inputs to the system (actors) and the system itself. Actors may be (but are not limited to) users of the system. An actor's role represents some functionality of the system that will be exercised.

A use case is a description of the interaction between an actor and the system. It contains at least one narrative description of a scenario. A scenario is an example of specific usage, one in which the actor supplies the input and the system demonstrates an observable output. Use cases are not required for object-oriented development, but the may provide important insights into the relationship between the functionality and areas such as testing. A testing scenario could be developed from a use-case scenario in a straightforward way.

A use case may have many scenarios. There is usually one main scenario and possibly many alternate scenarios. The alternate scenarios in the use case may represent exception handling or other

[69] First introduced by Ivar Jacobsen.
[70] Functional requirements are more dynamic and typically require accompanying details in order to understand then

options presented in the main scenario. Thus, the main scenario may be looked at as the "positive" case or path, compared to the others.

A use case is written to express what the expectations of the system are, i.e. what the system is expected to do. This is a view of the behavior of the system from the outside. A use case is not concerned with implementation details. Rather, a use case is concerned only with the inputs to and outputs from the system, without describing how the inputs are transformed into the outputs.

Use cases are written in the language of the problem-space (domain). This means we would not expect to find technical jargon in the use case document, per se. A use case is a way of capturing requirements and the requirements express what the expectations of the system are. For all involved, the scenarios in the use case documents should be clear and easily understood.

Use Case Models

Relationships between use cases are captured in a use case diagram. The collection of the use case diagrams is termed the use case model.

A use case is depicted by an oval with the name of the use case underneath. An actor is a stick figure. As above, the actor represents the role that a user[71] may have with the system. The arrow from the actor to the use case shows the actor providing input to the use case. Therefore, the arrow denotes input.

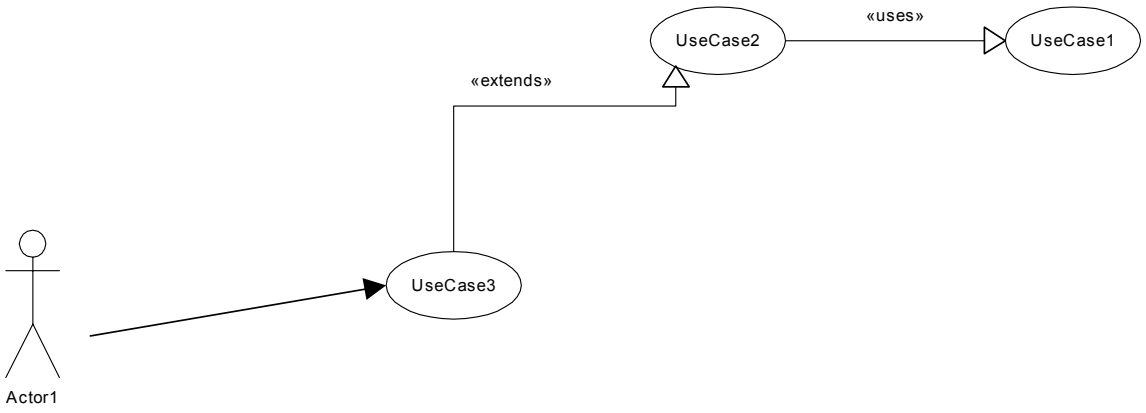


Fig A1.1 Use case example

[71] Remember, a user is not necessarily a person

Use cases may be extend other use cases or may include other use cases. These describe association between use cases. If case A extends case B, the actor invokes case B and can go on to case A. If case B includes case A then the actor invoking use case B also invokes use case A automatically.

Use case diagrams only show structural relationships between use cases. They do not capture dynamic relationships.

Once the initial information is captured in the use case diagram, other diagrams may be brought to bear on the problem. Other diagrams such as sequence diagrams[72] may be used to depict a more detailed flow of information from actor to system, with respect to a timeline.

Of course, use cases must also be analyzed. Use cases are used to capture requirements. The analysis of use cases allows us to identify classes and objects that will provide the functionality described in the use cases. In addition, we are also able to determine the responsibilities, attributes and associations, look at certain architectural structures and mechanisms, etc. The outputs are the realizations[73] and models depicting the static view of the system.

[72] These sequence diagrams are similar to those used to capture dynamic information about objects.
[73] A use case realization is a special use case that provides a description based on, and in terms of the system's architecture.

Appendix 2

Brief UML Reference

Introduction

Any non-trivial application needs to be designed in such a way that its structure facilitates design goals such as scalability, reliability, extensibility, etc. Their structure (i.e. architecture) must be clearly defined to allow unambiguous interpretation. This will allow relative ease of maintenance (among other benefits) as the application evolves.

A good structure benefits applications of any size, but, it is particularly useful for larger, more complex applications. This is true because looking at the structure of the application will provide a quicker grasp of the capabilities of the application. In turn, understanding the capabilities of the application and how it is structured will allow designers to leverage code reuse, as it will be easier to grasp how the application is organized into modules and/or components, each having the responsibility of some specific area of the overall functionality.

There are various ways of describing the structure of an application. One could review all the code in the application and derive an understanding of the application. While unambiguous, this method is surely the most tedious method that one could apply. Another suggestion would be to use words to describe the structure of an application. Someone could create text descriptions of the application that could be as detailed as necessary. While potentially easier to manage than reading the code itself, one would still have to read all

this documentation to grasp the structure of the application. This activity would still be very tedious and time consuming. Instead, we use modeling, to depict the structure of an application. As they say, a picture is worth a thousand words.

The Unified Modeling Language (UML), is designed to help you communicate the specifications of software systems. These specifications include their structure, design and behavior. With UML (especially via various available UML tools), you can perform an analysis of the system's requirements and create a design that satisfies them, using UML's visual language to communicate the results.

This visual language is comprised of a standard set of elements. The importance of standardization is that everyone looking a UML diagrams consisting of these elements will have the same understanding as to what each element means. This fact alone greatly improves the level of communication one can achieve using UML.

UML has many standard diagram types that can be used to create models of systems of widely varying architectures, on varying hardware and software platforms. This inherent flexibility allows you to use UML to model single-tier, multi-tier, web-based, or just about any architecture. You can also use UML with any of the popular development languages such as C++, Java, C#, VB.Net, etc., including even non-object oriented languages.

With UML, you can create platform independent models. In addition, you can also create methodology-independent models. There are various methodologies for software development. Software development methodologies define the formal steps one should take in developing software, from the gathering and analyzing requirements to the design and deployment of the solution. One important characteristic of UML is that UML's diagrams are used to help analyze and to communicate the architecture of the system, regardless of how you performed the analysis or created the architecture. Plus, since the diagramming elements of UML are standardized, various tools can interpret the system's specifications. UML's role is to communicate the results of our efforts in analysis, design and deployment of our system.

As this is a brief overview of UML, we'll concentrate on many "filling out" the UML diagrams and notations introduced in the text. As a result, this appendix will not cover every aspect of UML notation.

UML Diagram Types

UML defines many different diagram types. We can group these into different categories. With UML, we can look at the static structure of our application. We can also use UML to look at the behavior of our application, i.e. how it behaves at run time, or behavior scenarios (dynamic diagrams). Finally, with UML, we can also look at how we group areas of our application for organization. Let's examine each of these categories below.

Application Structure Diagrams

These diagrams represent the static structure of an application. The specific diagram types included in this category are as follows:

- Class Diagrams
- Object Diagrams
- Component Diagrams
- Deployment Diagrams

Application Behavior Diagrams

- Sequence Diagrams
- Activity Diagrams
- Collaboration Diagrams
- State Diagrams
- Use case Diagrams

Organization Diagrams

- Package Diagrams
- Subsystem Diagrams

As this is a brief overview of UML, we will look at simple examples to each, along with a description and overview of its usage.

Class Diagrams

Class diagrams show the static structure of a class. Since a class is the blueprint for an object, the class diagram allows us to see what the internals of the object looks like.

In the class diagram, we specify the name of the class, in addition to the methods and fields contained in the class. We can identify the access levels of the methods, in addition to their parameter lists and return types.

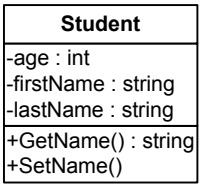
Class diagrams are also used to communicate the relationships between classes. These relationships may be associative (Association)

or hierarchical (Inheritance, Aggregation or Composition). Within the class diagram, we use various elements to indicate where a relationship exists, what type of relationship it is and any other information relevant to the relationship, such as the cardinality of an associative relationship.

Diagram Elements and Syntax

We use class diagrams to depict the structure and relationships of classes in our architecture. Some examples of these are as follows:

Basic Class Structure



Basic class element

In this structure, we see that the class element is separated into components for the name, attributes, methods (operations) and responsibilities (not shown). In class diagrams, attributes, operations and responsibilities are the most common features you'll use to depict abstractions (classes). Each class is named – the name of the class appearing in boldface followed by an underline (the first compartment). The *attributes* of a class, if included, are below the name of the class in the next compartment. The names of the attributes are preceded by symbols which denote their visibility (access level):

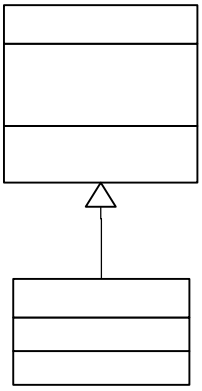
+: public
- : private
#: protected

The type of the attribute may also follow the name of the attribute, preceded by a colon (see examples above).

In the next (separate) compartment is the list of operations (methods) of the class. As above, each operation is preceded by the symbols denoting access level, and may be followed by its return type.

The last compartment of a class element is used to communicate the responsibilities of the class. Class responsibilities are basically free-form text.

Inheritance

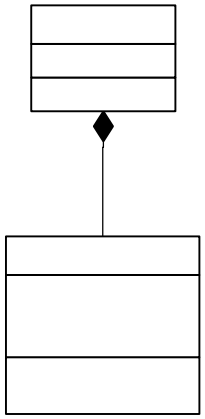


Student

-age : int
-firstName : string
-lastName : string
+GetName() : string
+SetName()

To depict inheritance relationships, the open arrow with the solid line is used. The arrowhead is placed at the superclass, with the arrow pointing from subclass to superclass.

Composition



Transfer Student

Inheritance relationship

In a composition relationship, there is usually a rule which governs the validity of the container with respect to the contents. Sometimes this rule has to do with the number and/or type of objects that are contained. If this rule is violated, the relationship is void. To depict this relationship, we use a solid diamond as an arrowhead. The diamond is placed at the class that represents the container, i.e. the arrow is pointing away from the class representing the contents (aggregate).

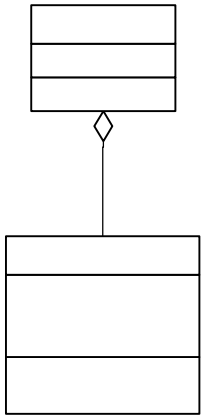
Students

-count
+Count() : int

0..1

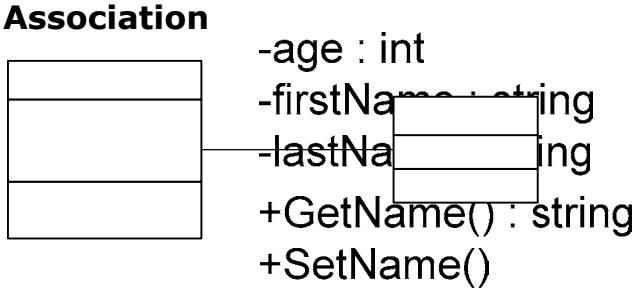
*

Aggregation



Aggregation and composition are quite similar, obviously. In Aggregation, there are no constraints governing the validity of the relationship. For example, we could have zero or more of the contained objects in our container – zero would be valid, as would any other number. To show this, we use an open diamond as our arrowhead. As above, the diamond is placed at the class that represents the container, i.e. the arrow is pointing away from the class representing the contents (aggregate).

Association

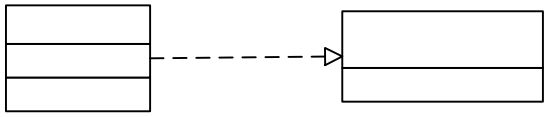


Aggregation relationship

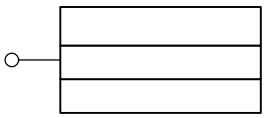
An associative relationship refers to a dependency between classes, the strength of which can vary. In the above case, we're depicting a "many to many" relationship, between Student objects and Account objects. Each link between classes can include symbols to denote the cardinality (multiplicity) of the association. Examples of these symbols are as follows:

1 : Only 1 (could be any numeral)
0..1 : 0 or at most 1
1..1 : At most 1
1..* : At least 1
.. : Many to many

Interfaces
As you create more complex architectures, you may want to explicitly communicate the separation of the interface from the implementation details. In UML, you can use interface elements to communicate this separation. Interface elements appear as follows:



This diagram reflects that the account class *implements* the AccountManager interface. A class implementing an interface may also be depicted as follows:



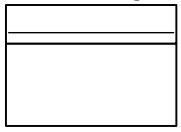
Note: classes may implement (realize) many interfaces.

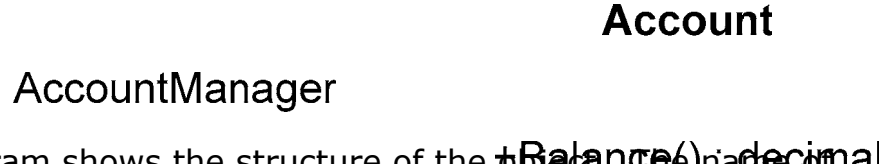
Object Diagrams

Object diagrams show the static structure of the object, but they don't stop there. Objects exist at runtime, not at design time. So, in addition to the structure of the object, object diagrams are used to communicate the behavior of the object and how objects interact with each other. Each object's behavior is based on the methods contained in that object. Objects interact with each other by invoking (calling) the public methods that are defined in other objects. The object diagram provides a visual representation of what methods are being called by which objects, depicting system behavior at runtime, though without definition of order (sequence) or time.

Diagram Elements and Syntax

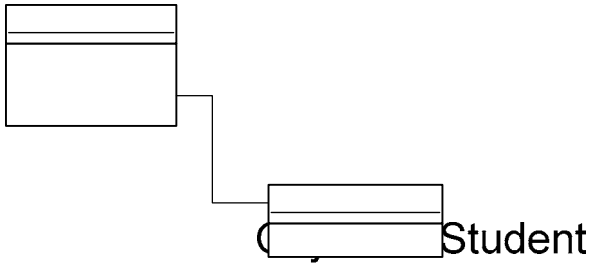
Basic object diagram





This simple diagram shows the structure of the object. The name of the object is in the first compartment, followed by the attributes in the next. Note, in object diagrams, the name of the object is underlined, unlike in class diagrams.

Object Interaction



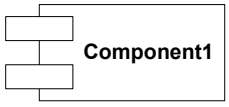
In this diagram, we're depicting the interaction between two objects, one of class Student and the other of class Account.

Component Diagrams

A component is a stand-alone, (i.e. deployable), part of a system's implementation. A component can be looked at as a runtime "container", comprised of the objects of various classes. The component provides some functionality that is based on the functionality "contributed" by the included classes. A component is also an element of a distributed system. As our architecture becomes distributed, we find that we need some way to look at our architecture from the standpoint of which components are interacting with each other, i.e. at a higher level than that of individual classes and objects. This higher level view allows us to more quickly obtain information about our system than would be obtained otherwise.

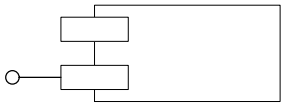
Diagram Elements and Syntax

Basic Component



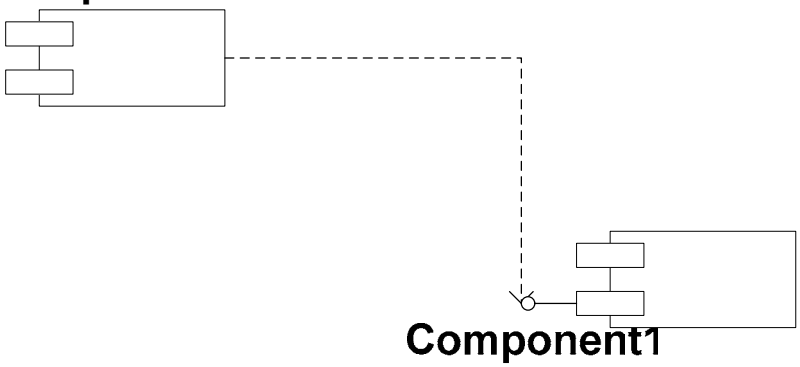
Here we see the component element, showing the name of the component

Component with Interface



As with classes, components can also implement interfaces, as shown here

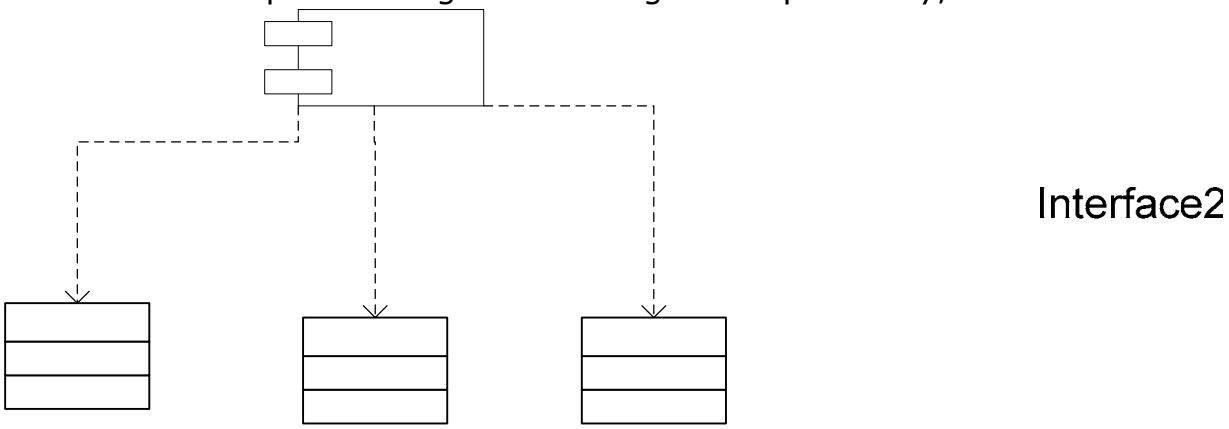
Component interaction



This diagram expresses the interaction between two components, via Component2's interface

Depicting Components and Classes

A component's functionality is provided by its classes. We can say a component is dependent on its classes. To show this relationship, we can create a component diagram showing this dependency, as follows:



Deployment Diagrams

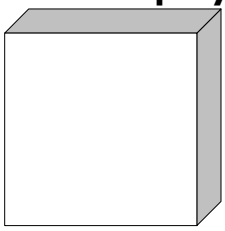
Many of the diagrams we discuss serve to help us visualize the structure and behavior of our software systems. Deployment diagrams help us visualize the physical layout of our systems. It shows the relationships between the software and hardware elements of the system.

Deployment diagrams consist of nodes that reflect a unit of computation such as a piece of hardware. The lines between nodes represent links that show communication pathways between nodes. Within each node of a deployment diagram are components (see component diagrams above).

Deployment diagrams are useful when you need to show how components in your system will be or are deployed. This information is obviously different from the logical information reflected by many of the other UML diagrams we've seen.

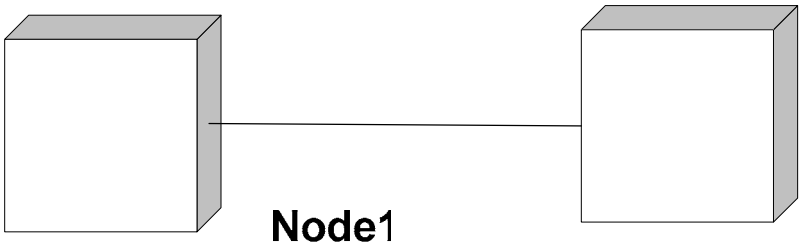
Diagram Elements and Syntax

Basic Deployment Element



In this example, we see the node element including the name

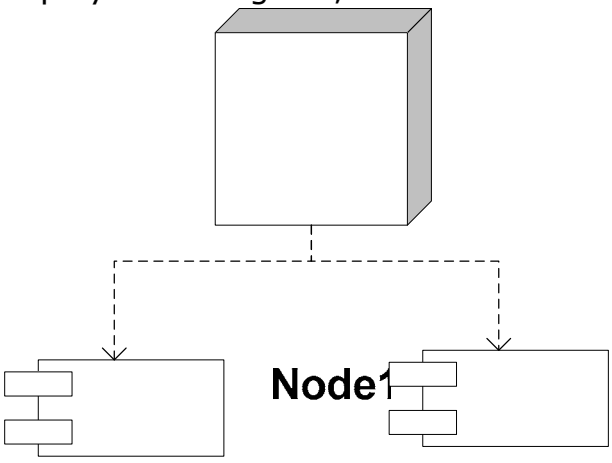
Node Interaction



Each of the lines between the nodes represents the physical communication (i.e. connection) between the nodes.

Nodes and components

Nodes are composed of components. To show the dependency of the nodes to their constituent components, we can use a variation of the deployment diagram, as follows:



* *

Sequence Diagrams

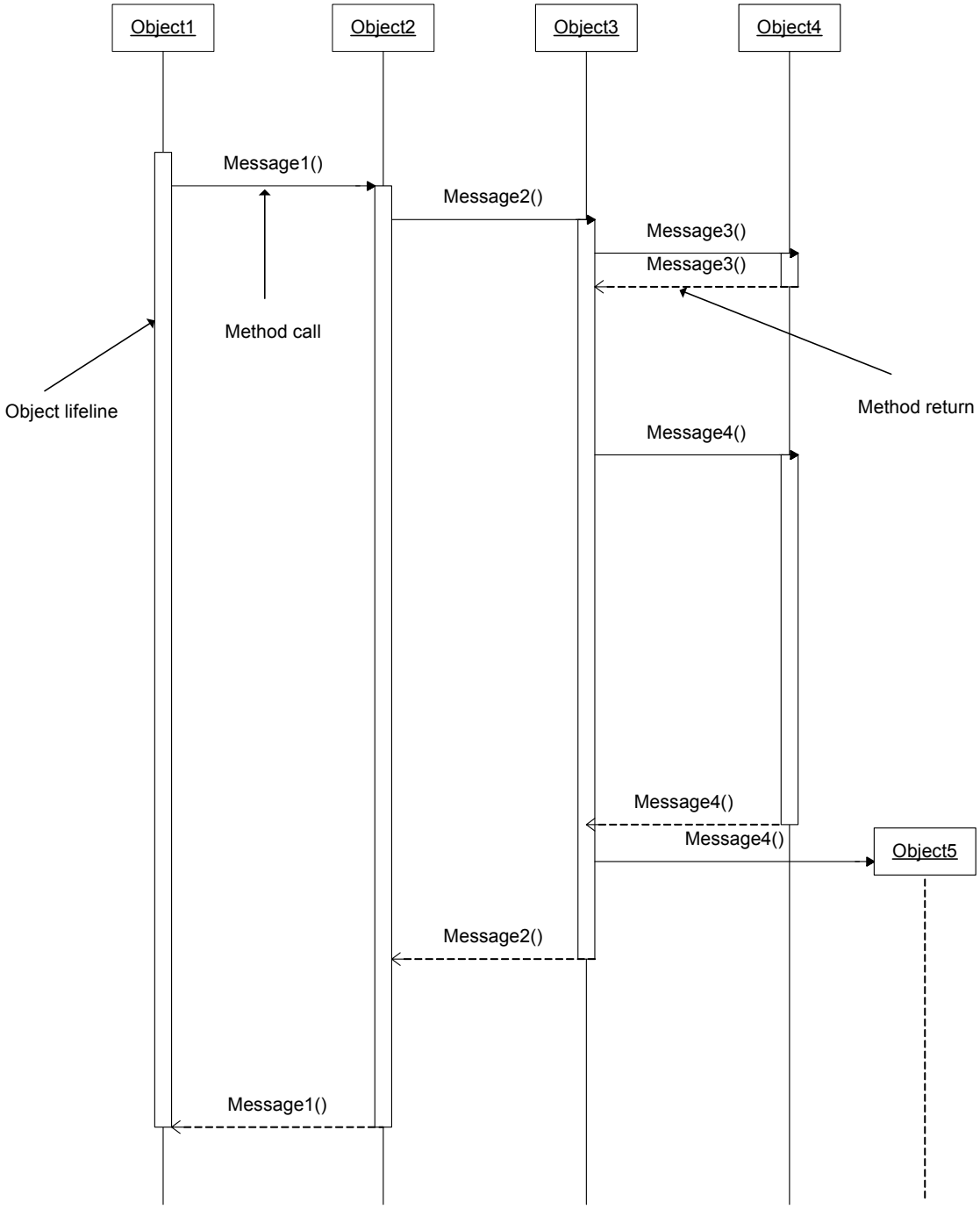
A sequence diagram conveys the sequence of messages (method calls) for a group of objects, as those objects collaborate to satisfy some requirement of the system. Another way of characterizing a sequence diagram is to say that it depicts the explicit sequence of methods calls for a group of objects as those objects collaborate to satisfy a use case.

In a sequence diagram, there is an explicit start point and end point. These correspond to the start of the scenario and the end of the scenario. As the scenario progresses, objects are recruited for their functionality via message passing (i.e. method calls). The sequence diagram makes it easy to identify the individual steps taken, from one object to another.

Because a sequence diagram explores the "path" through a number of objects to satisfy a use case, the sequence diagram is not intended to show all of the methods that are defined on each object in the scenario. It serves only to highlight those methods that are explicitly called to satisfy the use case. The focus of the sequence diagram is the "sequence" of the messages only. It shows the flow of logic as we traverse through the scenario, which allows us to record and validate the logic.

Diagram Elements and Syntax

Basic Sequence Diagram



Each object in the sequence is represented by the object element at the top of the diagram. Extending down from each object is its lifeline, i.e. the line downward represents the object's lifetime. Each long rectangle that is positioned on the lifeline represents the "activation"

of the object, i.e. the time in which a method on the object is executing.
Each message (i.e. method call) is represented by the arrows going from one object's activation to another. Each line implicitly represents the method's call and return. However, a method's return can be explicitly denoted by using a dashed line in the opposite direction to the invocation. This dashed line would be placed at the bottom of the object's activation rectangle.

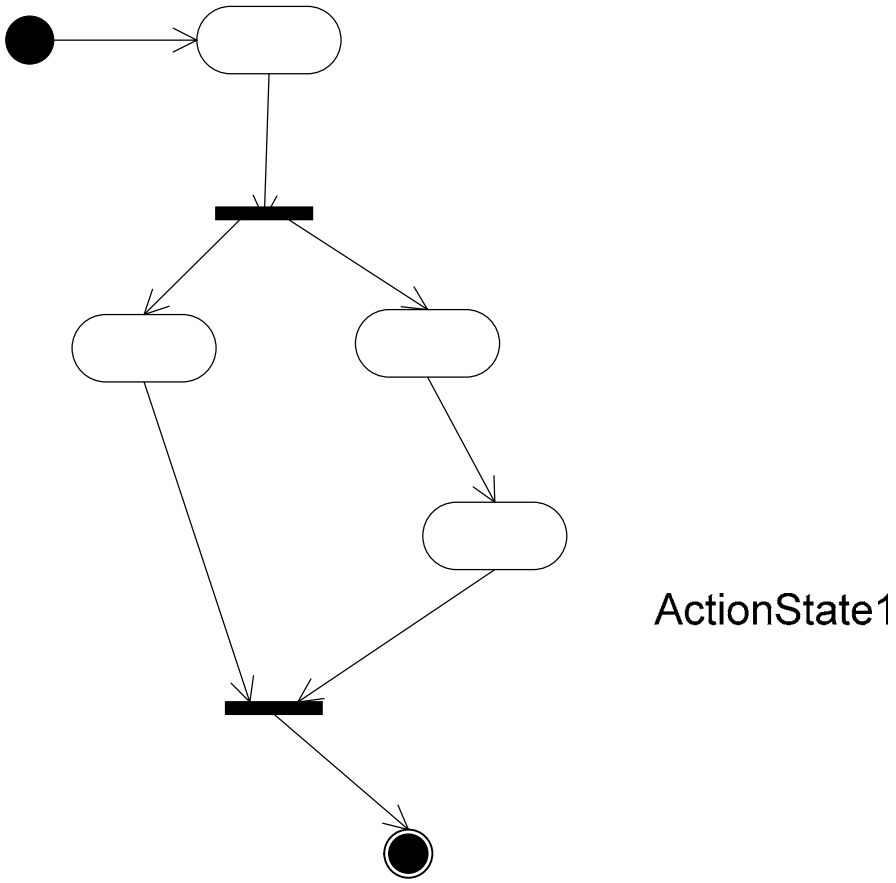
Sequence diagrams can also depict asynchronous method calls (synchronous by default). In the case of asynchronous calls, instead of a "full" arrowhead, a "half" arrowhead is used for each "half" of an asynchronous message.

Activity Diagrams

An activity diagram is similar to flow charts, except they include the ability to model parallel behavior. Activity diagrams are useful when depicting the behavior of multithreaded applications, or other complex processes.

Diagram Elements and Syntax

Basic activity diagram



In the activity diagram, as in the State diagram, the starting point is the initial node, a filled in circle. While not required, the presence of this node makes it easier to comprehend the diagram. The filled circle with an enclosing circle signifies the ending point of the diagram.

Between the starting and ending nodes in the diagram is where we find "activities", "flows", "forks" and "joins". An activity is something that occurs (an action). When something occurs, that places our object in another state, during which something else occurs. Transitions between one action state and another is depicted by the arrows between states.

Sometimes, as a result of an occurrence, there are two "paths" that may be followed simultaneously, i.e. parallel activities. The horizontal bar with one arrow coming toward it and two or more arrows going away to other action states indicates a fork, i.e., the start of parallel processing. At the end of parallel processing, a join is executed. The join is a horizontal bar with two or more arrows leading toward it, with one arrow leading away.

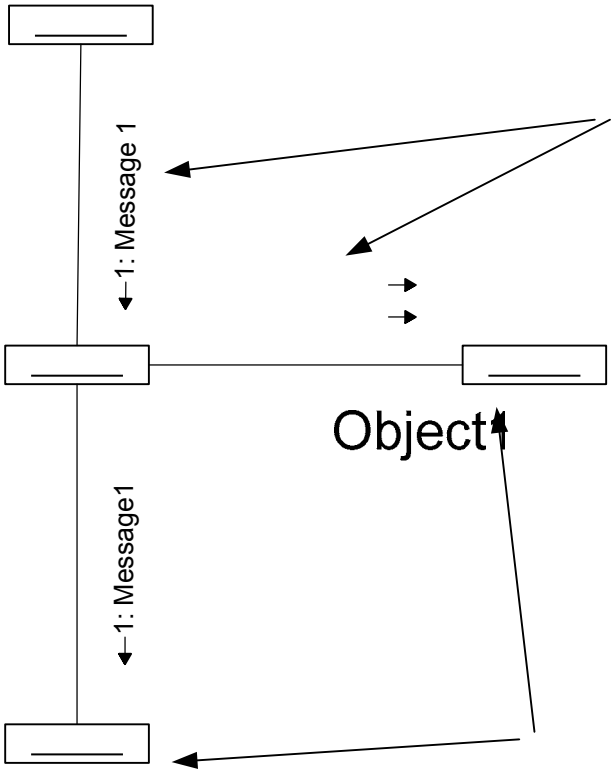
Collaboration Diagrams

Collaboration diagrams show the interaction of objects in an application, i.e. the message flow (method calling). They provide a view of a collection of collaborating objects. This view allows us to see the functionality provided by each object, aspects of the flow of logic between objects and also roles that objects can assume within their lifetimes.

Unlike sequence diagrams, collaboration diagrams focus on the relationships between objects. The name alone implies that we would use such diagrams to help us understand how a group of objects collaborates to accomplish some goal.

Diagram Elements and Syntax

Basic Collaboration Diagram



In Collaboration diagrams, each rectangle represents an object. The lines between the objects in the diagram are links that represent relationships (association, aggregation or composition) between objects. The arrows indicate method calls and their respective directions. The numbers associated with each arrow indicates the sequence in which the methods are called.

1: Message1
2: Message2
Object2
Objec

State Diagrams

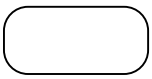
As the methods defined for an object are invoked, the values of the object's internal data might change. Each of these changes, in response to a method call, constitutes a different state that the object is in. Throughout an object's lifetime, it may inhabit many states, with the change from one state to another being a "transition".

As designers, we may want to focus on an object and look at what it takes to move from one state to the next, what the values of the data

are in various states and how the object interacts with other objects in our system as we travel from state to state. The UML diagram that allows us to do this most easily is the UML State diagram. With the state diagram, we use UML elements to depict the various states of the object, as well as the state transitions. A state diagram combines states and method calls in order to depict all possible object states during its lifetime. The state diagram will help us visualize how the object responds to invocations of its methods.

Diagram Elements and Syntax

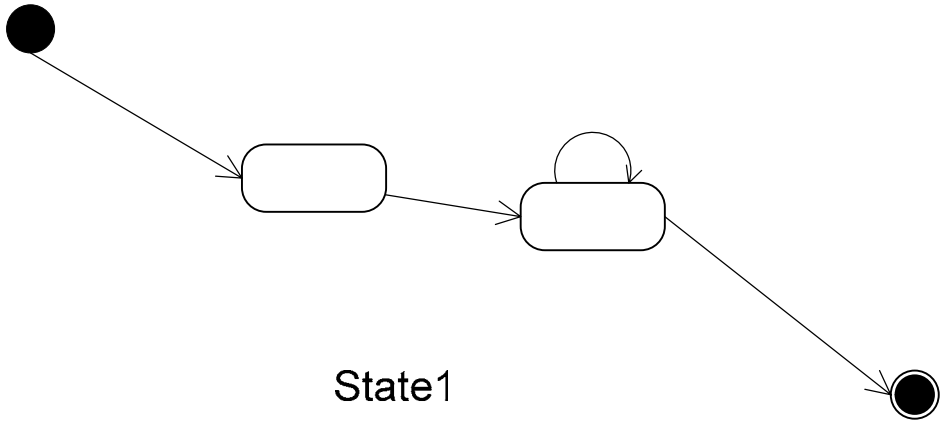
Basic State Element



Each "state" is represented by this element. The name of the state is included within.

State Transitions

As the object progresses through its lifetime, its state changes. This transition from state to state is represented by lines between states, as shown below:



Each state diagram begins with the object in an initial state, as denoted by a solid circle. Each diagram also ends with the object in a final state, denoted by a solid circle, enclosed within another circle.

Use Case Diagrams[74]

Use case diagrams provide a generalized view of how a system will be used. Each use case represents a specific usage scenario that is a sequence of interactions between user and system, the goal being the satisfaction of some requirement. The collection of use cases would thus provide all of the requirements for a system. One of the benefits

[74] For a more complete discussion of use cases, see Appendix 1

of use case diagrams is that it allows anyone to see a visual representation of the intended usage and functionality of the system.

Package Diagrams

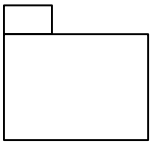
A package is a fundamentally group of related classes. The grouping is used to organize classes into something more manageable. The relationship between classes in a package is not only limited to the class relationships we've seen before, i.e. Association, Inheritance, Aggregation or Composition. The "relationship" that exists between classes in a package may be simply that they share responsibility for providing some aspect of a system's functionality. For example, you could create a package of utility classes. Each class provides some "utility" functionality that is used in one or more places in the system. Another perspective is that a package is a logical grouping of classes, i.e. there is some "logic" to why they're grouped together.

Many languages, such as Java and the Microsoft .Net languages extensively use packages to organize classes. When the functionality that exists in one or more classes in a package is required, the package is utilized. In so doing, the class name is prefaced by the package name.

A package diagram is a diagram that shows groups of classes and the dependencies between them. A dependency between packages implies that there is a dependency between two or more classes in each package.

Diagram Elements and Syntax

Basic Package Diagram



This diagram shows the package element, including the name.

Package Interaction

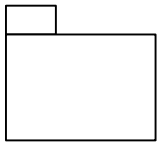
As we mentioned before, if one class in a package is dependent on a class in another package, then the first package is dependent on the second package. The example above shows this relationship between Package1 and Package2.

Packages and Classes
Packages represent a group of classes. This relationship between packages and classes can be represented by drawing the contained class elements within the package element.

Subsystems
In general terms, the *system* is the solution you're designing. Depending on the complexity of the system, a system may be viewed as being a collection of sub-systems.

UML provides a set of elements that we can use to depict as system as a group of subsystems. Diagrammatically then, a system is depicted as an aggregation of subsystems, using the same aggregation elements as with class diagrams.

Diagram Elements and Syntax
Basic Subsystem



As you can see, the subsystem element is very similar to the package element. The difference is that the subsystem element contains a *stereotype*, which is the text in the diagram between the << and >> symbols. The stereotype is used to indicate whether the element represents the system (as a whole) or a subsystem.

System/Subsystem Relationship
Systems are comprised of subsystems. This relationship is represented as a composition of subsystems, using the composition relationship element as described earlier (class diagrams).

«subsystem»
SubSystem1

Appendix 3

Object-Oriented GUI Design Elements

Graphical User Interfaces (GUI's) have been around for many years now. They were brought into the mainstream by Apple Computer with its Macintosh line and further popularized by Microsoft's Windows environment.

GUI frameworks have also become popular. These frameworks are presented as part of an Integrated Development Environment (IDE), which allows users to write code, debug and build software. There are many commercially available GUI frameworks in IDE's. Some of the more popular are Microsoft's Visual Studio for Windows, X Windows for Unix, CodeWarrior and JBuilder for Java. These frameworks exist to simplify the construction of a user interface for an application.

A GUI framework consists of various abstractions representing tools used in building user interfaces. These tools consist of text boxes, scroll bars, buttons, sliders, checkboxes etc., many of which are displayed in the two figures following. Each abstraction has various operations and attributes defined on it based on the appropriate semantics. For example, the value of a checkbox would not be the same as the value of a text box. A checkbox may have two or three states, depending on implementation. Those states would be on, off, indeterminate (or true, false, indeterminate). The value of a textbox would be the current contents of that textbox. Each element of a GUI is an object, i.e. an instance of the class in which various operations and attributes are defined.

GUI frameworks allow you to "draw" elements (or controls) directly onto windows, specify windows (dialog windows, etc.), define menus and menu items, etc. Typically, this graphical construction generates the code corresponding to the GUI elements. This is far simpler than

trying to construct and place these elements from the bottom up. In many GUI frameworks the elements share a common superclass element, such as a superclass representing a generic control. In this way, the framework and developers can exploit Polymorphism in the workings of the framework and in the development of applications.

There are various mechanisms available in GUI frameworks. One of the most important is the event handling mechanism. The elements of a GUI respond to events. Events are triggered by a variety of sources. Some of these are as follows:

- Mouse events
- Keyboard events
- Menu events
- Window activation and deactivation events
- Window resizing events
- Initialize and terminate events

This event-handling mechanism allows us to use the mouse to navigate among the various controls, as each movement, click, drag, etc., translates to an event. Each event is then handled as appropriate, based on the context in which the event occurred.

GUI frameworks, as part of their overall capabilities, give developers the opportunity to implement custom code in the event-handling routines. It should be noted that GUI frameworks typically allow developers to add custom code to handlers for events that are already defined. You are typically not allowed to define new events or handlers. Each control will have a set of events appropriate for it. For example, a text box will have handlers for events that buttons will not have.

The following pages show some of the more common GUI design elements, as found in the more popular IDE's.

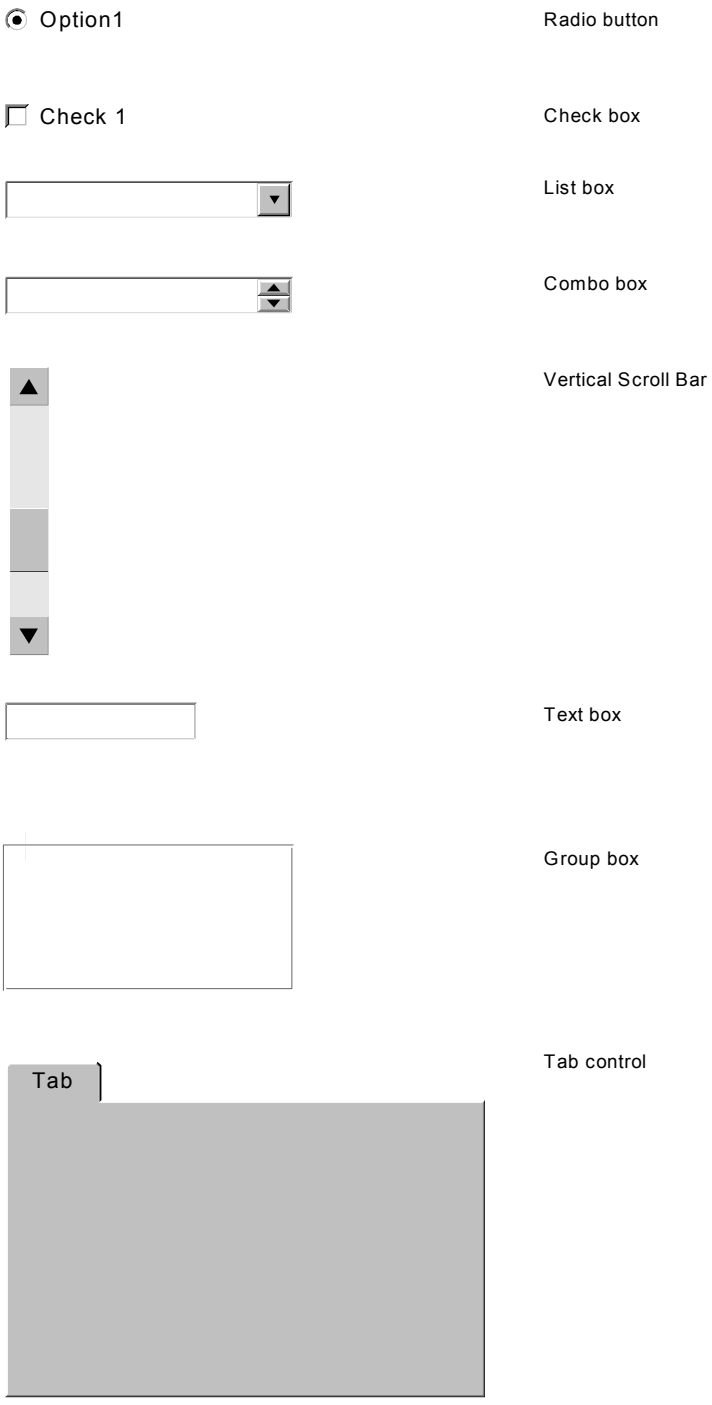


Fig A2.1 Some GUI elements

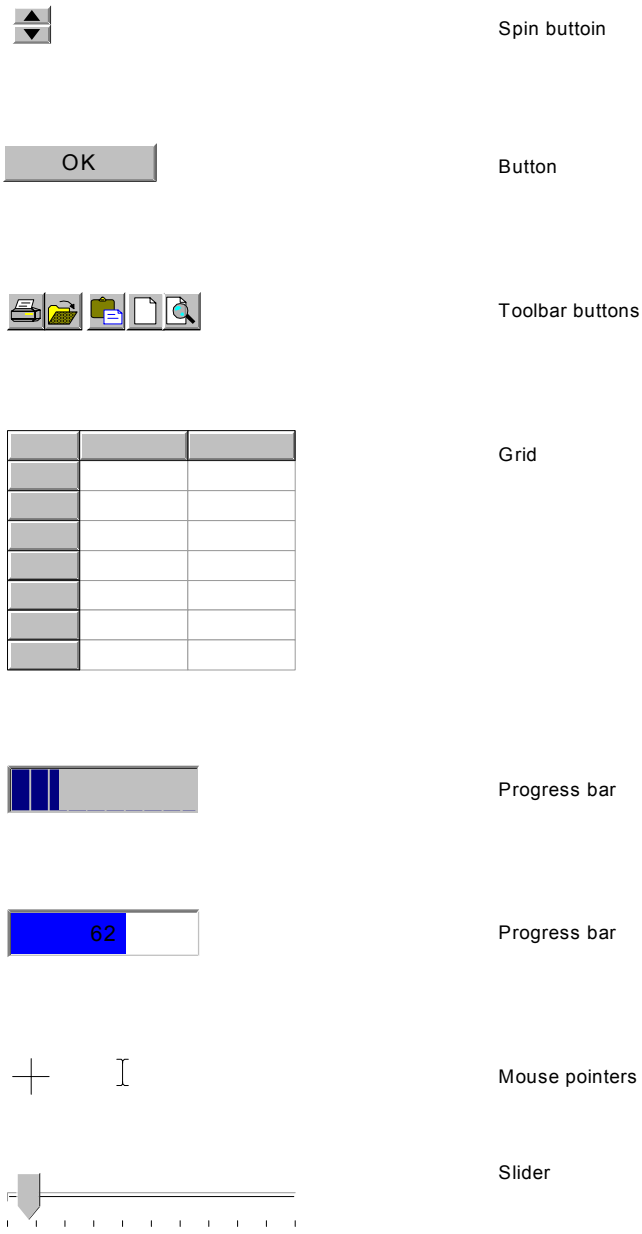


Fig A2.2 Some more GUI elements

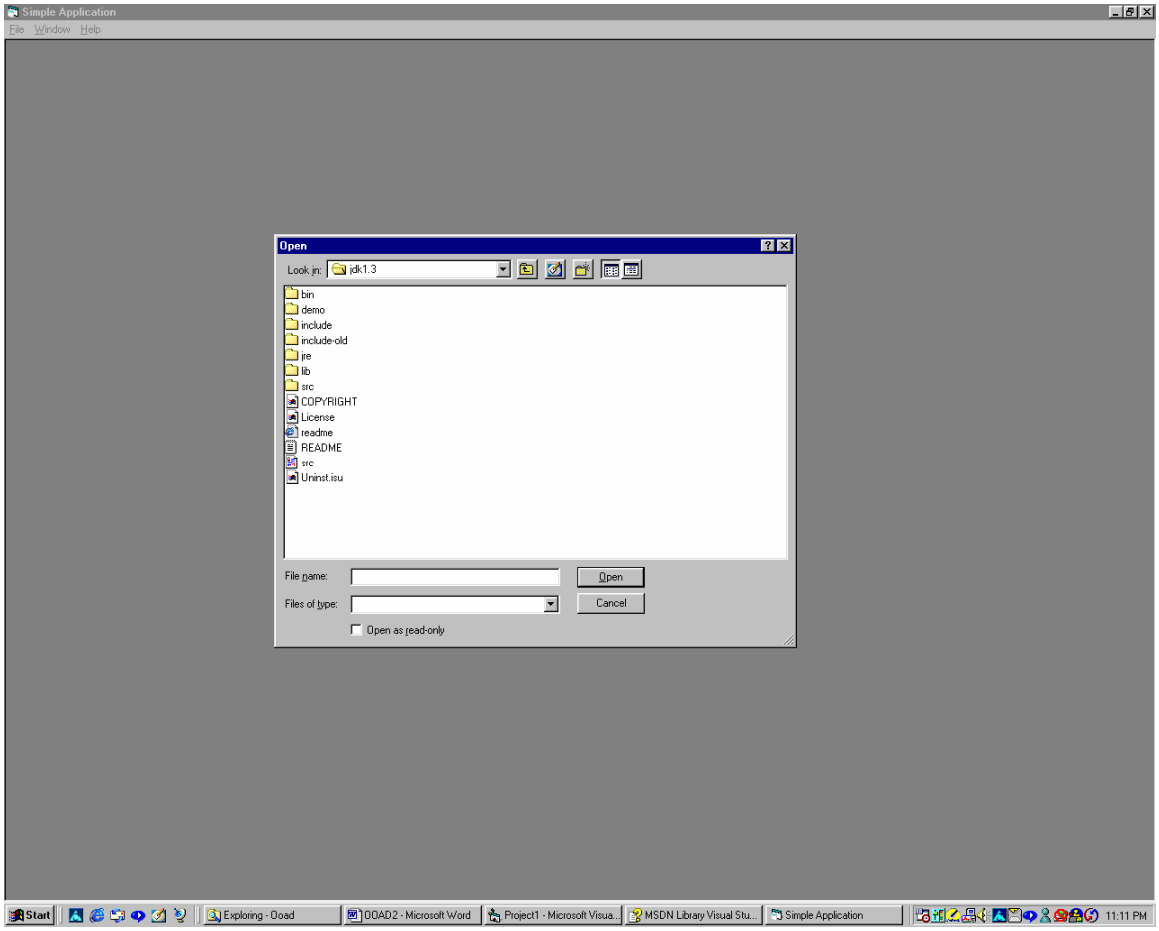


Fig A2.3 Simple Windows application

Glossary

Analysis
A phase of the overall software development process (lifecycle) in which high-level models of the system (based on the system requirements) are created. These models are built with an understanding of the functionality the system is to provide, again based on the system requirements.

Design
A phase of the overall software development process (lifecycle) in which the high-level models from the Analysis are made more concrete by factoring in the environment, constraints, non-functional requirements, cost, time-to-market, etc. The output of Design is a set of models that are the basis for writing code.

Procedural Languages
Computer languages in which problems to be solved by a computer are broken into more manageable pieces and each piece is solved with a unit of code.

Hierarchical Decomposition
Also called "algorithmic decomposition," hierarchical decomposition is the method by which procedural languages break complex problems into manageable pieces. Hierarchical decomposition is the cornerstone of top-down design methodology in Computer Science, and its advent allowed complex problems (some orders of magnitude greater than those previously undertaken and solved) to be dealt with more routinely.

Abstract Data Types
User-defined types, e.g. structs in C, records in Pascal, etc. They allowed programmers to create types that were abstractions of elements of the problems they were trying to solve. These abstractions further allowed the aggregation of primitive types (i.e. integers, characters, etc.) in a way that was more meaningful to human designers and coders. They also resulted in more readable and organized code.

Classes
Blueprints for object.

Objects
Instances of a class. They may also be viewed as the members of a class.

Attributes
An object's data are its attributes.

Methods
The means to manipulate an object's data. Methods provide the functionality of the object.

Abstraction
A description of a system that does not focus on all details, only on those that are relevant, like a simple schematic drawing.

Hierarchy
A hierarchy is an ordering of items. Inheritance, in which the "is a kind of" relationship is examined, is one way to develop a hierarchy.

Encapsulation (information hiding)
The ability to separate the interface of a class from the implementation

Object-Oriented Decompsition
The ability to model an operational system based on cooperating objects, which is closer to reality that the earlier hierarchical decomposition would allow.

Persistence
Storing the value of an object for later use. An object's class and current state may be saved for later use.

Concurrency
Ability of objects from the same class to have simultaneous existence.

Typing
Classifying variables by the kind of data they hold (integers, strings, etc.).

Object-Oriented Analysis

An analysis of the system requirements that is based on object-oriented thinking. This is an analysis based on the object-oriented decomposition, as opposed to the top-down hierarchical decomposition of structured analysis.

Object-Oriented Design
Design that is also based on object-oriented thinking. In the design phase, we are concerned with making the models developed in the analysis phase more concrete and refined, readying them for development.

Object-Oriented Programming
The development of programming code also based on object-oriented thinking, using an object-oriented language and environment (C++, Java etc.).

Bibliography

Booch, Grady: Object-Oriented Analysis and Design with Applications 2nd Edition. (1994), Benjamin/Cummings Publishing Company, Inc., California

Booch, Grady, Jacobson, Ivar and Rumbaugh, Joseph: The Unified Modeling Language User Guide (1999), Addison Wesley Longman Inc.

Weisfeld, Matt: The Object-Oriented Thought Process (2000), Sams Publishising Company, Inc.

McConnell, Steve: Code Complete (1993), Microsoft Corporation

Krutchten, Phillippe: The Rational Unified Process An Introduction, 2nd Edition (2000), Addison Wesley Longman Inc.

Harmon, Paul and Morrissey, William: The Object Technology Casebook (1996), John Wiley and Sons Inc.

Hofmeister, Christine, Nord Robert and Soni, Dilip: Applied Software Architecture (2000), Addison Wesley Longman Inc.

Eckel, Bruce: Thinking in C++ (1995), Prentice Hall Inc.

Lau, Yun-Tung: The Art of Objects Object Oriented Design and Architecture (2001), Addison Wesley Longman Inc.

Kafura, Dennis: Object-Oriented Software Design and Construction with Java (2000), Prentice-Hall Inc.

Page-Jones, Meilir: Fundamentals of Object-Oriented Design in UML (2000), Addison Wesley Longman Inc.

Treese, G Winfield and Stewart, Lawrence C.: Designing Systems for Internet Commerce (1998), Addison Wesley Longman Inc.

Conallen, Jim: Building Web Applications with UML (2000), Addison Wesley Longman Inc.

Fowler, Martin and Scott, Kendall: UML Distilled Second Edition (2000), Addison Wesley Longman Inc.

Index

Abstract data Type, 15, 134
Abstraction, 10, 21, 25, 27, 28, 34, 35, 36, 37, 38, 45, 50, 51, 52, 54, 55, 71, 75, 99, 100, 101, 133, 134, 135, 137, 138, 139, 140, 141, 145, 147, 149, 150, 151, 175, 181, 202, 203, 209, 210, 211, 212, 213, 216, 228, 229, 240, 250, 257, 258, 259, 287
Activities, 210, 211, 213
Additional Considerations, 149
Address, 56, 72, 74, 75, 77, 98, 100, 101, 102, 123, 124, 125, 155, 156, 157, 161, 163, 164, 165, 169, 170, 171, 172, 175, 176, 177, 183
Aggregate, 45, 46, 89, 92, 111, 152, 169, 177, 178, 179, 183, 197, 201, 241, 255
Aggregation, 16, 44, 45, 46, 59, 80, 81, 89, 91, 92, 105, 111, 136, 140, 144, 147, 197, 209, 212, 258, 260
Algorithmic Decomposition, 15
Analysis, 1, 21, 23, 25, 27, 51, 52, 53, 60, 72, 131, 186, 208, 209, 214, 215, 216, 219, 221, 225
Architectural Elements, 240
Architecture, 9, 10, 133, 141, 207, 215, 216, 219, 220, 234, 235, 239, 240, 241, 242, 244, 246, 249, 250, 251, 253, 262, 267
Associations
  Mandatory, 91
  Optional, 91
Associative, 89, 90, 91, 103, 168, 169, 202, 213
Associative Relationships, 89
Attributes, 10, 25, 32, 33, 34, 39, 40, 60, 84, 93, 96, 97, 98, 99, 100, 114, 123, 124, 127, 133, 141, 142, 148, 150, 156, 159, 160, 163, 164, 167, 169, 170, 171, 175, 182, 201, 202, 207, 210, 213, 267, 287
Availability, 12, 23
Base class, 81, 82, 87, 95, 159, 160, 168, 202, 229, 230
Behaviour, 29, 30, 32, 33, 39, 41, 43, 87, 89, 93, 108
Behaviour Analysis, 52, 58
Benefits, 35, 47, 50, 51, 71, 88, 141, 229, 260
  Benefits of Class Modeling, 71
C++, 9, 10, 20, 25, 34, 56, 61, 65, 66, 67, 73, 85, 107, 142, 230, 232, 233, 234
Capturing object behaviour at run-time, 114
Cardinality, 90
Class, 9, 13, 22, 23, 29, 30, 31, 32, 33, 34, 35, 38, 39, 40, 41, 43, 46, 47, 50, 51, 55, 58, 60, 61, 62, 63, 64, 65, 66, 67, 71, 72, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 103, 104, 105, 106, 107, 108, 109, 111, 115, 123, 124, 125, 126, 127, 129, 130, 131, 132, 133, 135, 136, 137, 138, 140, 141, 143, 145, 147, 148, 149, 150, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 201, 202, 203, 205, 209, 210, 211, 212, 213, 215, 227, 228, 229, 230, 231, 232, 233, 234, 238, 242, 243, 249, 257, 258, 259, 260, 287
  Class and Object Interactions, 66
  Class Associations, 89

Class diagram, 22, 23, 72, 80, 104, 115, 130, 167, 213, 215
Class field, 106, 107
Class hierarchies, 33, 81, 141
Class Modeling, 67, 79
Class semantics, 32
Class Structure, 60
Class-Object Relationships, 111
Identifying classes, 51, 182
Identifying classes Identifying Classes and Objects, 51
Instance field, 106, 129, 130
Instance field vs. Class fields, 106
Instances, 30, 33, 34, 47, 58, 66, 67, 79, 85, 98, 105, 106, 107, 108, 115, 119, 129, 130, 148, 153, 154, 182, 186, 254, 258, 287
Responsibilities, 53, 55, 75, 76, 96, 97, 99, 137, 202, 210, 211, 212, 267
Classes and Objects, 9, 29, 47, 51, 52, 53, 55, 60, 68, 85, 132, 133, 147, 209, 210, 212, 213, 215, 216, 217, 267
Client, 12, 140, 147, 148, 149, 227, 234, 242, 246, 250, 258, 259, 263
Client/Server, 242
COBOL, 9, 14
Code, 14, 15, 18, 20, 23, 24, 25, 46, 50, 58, 61, 66, 71, 88, 89, 92, 131, 144, 146, 153, 219, 220, 228, 256, 257, 258, 259, 260, 262, 264, 287, 288
Code Maintenance, 259
Cohesion, 202
Collaboration, 115, 118, 119, 129
Collaboration Diagram, 118, 119, 129
Collaborations, 204, 213
Collection Classes, 152
Collections, 152
Lists, 152
Maps, 152
COM, 234, 247
Comments, 11
Completeness, 109, 138, 203, 205, 243
Components, 15, 40, 46, 47, 58, 89, 142, 151, 216, 219, 233, 234, 235, 239, 240, 241, 242, 243, 244, 245, 246, 247, 249, 253, 254, 258, 259, 260, 262
Components and Maintainability, 260
Components and Reuse, 259
Components in UML, 246
Composite, 111
Composition and Aggregation, 44, 89
Differences between Aggregation and Inheritance, 89
Composition and Aggregation Composition, 43, 44
Composition and Aggregation Composition, 45
Composition and Aggregation Composition, 46
Composition and Aggregation Composition, 59
Composition and Aggregation Composition, 80
Composition and Aggregation Composition, 81
Composition and Aggregation Composition, 89
Composition and Aggregation Composition, 89
Composition and Aggregation Composition, 89
Composition and Aggregation Composition, 89
Composition and Aggregation Composition, 89
Composition and Aggregation Composition, 91
Composition and Aggregation Composition, 92
Composition and Aggregation Composition, 99
Composition and Aggregation Composition, 105
Composition and Aggregation Composition, 111
Composition and Aggregation Composition, 144

Composition and Aggregation
Composition, 167
Composition and Aggregation
Composition, 201
Composition and Aggregation
Composition, 258
Composition and Aggregation
Composition, 260
Conceptualization, 208, 221, 225
Concrete, 23, 25, 27, 47, 131, 210, 231, 238
Constraints, 18, 21, 22, 23, 27, 111, 131, 186, 215, 216, 240, 262
Containers, 147
Controls, 93, 287, 288
CORBA, 234, 235, 239, 245, 247
Costs/Benefits, 91
Coupling, 136, 202, 259
CRC Cards, 55
Data dictionary, 210, 213
Data Management Methods, 181
DCOM, 239
Decomposition, 15, 21, 22, 25, 27, 34, 48, 51, 145, 209, 240
Default Constructor, 108, 110
Dependencies, 213, 216, 259
Deployment, 220, 221, 222, 225, 244, 258
Derived Class, 94, 95
Design, 1, 17, 23, 25, 27, 56, 57, 60, 72, 131, 132, 133, 146, 148, 149, 205, 208, 215, 216, 218, 219, 221, 225
Design Elements, 148
Design Guidelines, 132
Design Patterns, 146, 149, 205
Designing for Interoperability, 149
Designing with classes and objects, 131
Design time, 60, 105, 107, 111, 132
Destructors, 110, 163, 182, 211
Development Phases, 219
Development Process
Purpose, 209, 210, 212
Distributed Systems, 9, 10, 12, 142, 227, 228, 233, 234, 237, 239, 240, 241, 244, 249
Documentation, 21, 22, 23, 210, 219, 220
Domain Analysis, 52, 53, 58
Drawbacks, 50, 88
Encapsulation, 25, 27, 34, 35, 38, 39, 40, 41, 50, 51, 58, 93, 133, 139, 140, 141, 147, 148, 175, 217, 228, 233, 242, 257, 258, 260
Information Hiding, 93
Enterprise Java Beans, 239, 245, 247
Environment, 18, 23, 25, 72, 88, 91, 108, 131, 148, 153, 215, 216, 217, 219, 220, 227, 243, 244, 245, 287
Environment
Environmental opportunities, 215
Environment
Environmental opportunities, 216
Evaluation, 202
Event-handling, 53, 67, 154, 288
Example, 56, 72, 94, 96, 106, 123, 153, 235, 249
Adding a Student, 189
AllMajors, 102, 123, 126, 127, 158, 162, 163, 166, 167, 178, 184
DBClass, 182
Student Majors, 102, 123, 126, 155, 163, 166, 175, 183
Student Subjects, 102, 123, 126, 155, 163, 166, 183
StudentMajors, 158, 162, 163, 166, 173, 177, 178, 201
Student-Related Data, 174
Students, 159, 163, 167, 179, 181, 182, 183, 184, 201
StudentSubjects, 158, 162, 163, 167, 173, 175, 177, 178, 201
Subjects, 102, 123, 126, 127, 158, 163, 166, 184
System, 75, 78, 99, 101, 102, 123, 126, 154, 158, 162, 163, 166, 169,

179, 181, 182, 183, 185, 187, 200, 201, 202, 207
System Help, 200
System Prototype, 187
Extensibility, 145, 217
Features, 22, 27, 228
First-Generation languages, 14
Function, 40, 52, 61, 95, 144, 147, 152, 180, 183, 201, 231
Functional, 21, 22, 23, 27, 47, 131, 153, 186, 208, 215, 216, 219, 240, 256, 265
Functional Requirements, 22
Graphical User Interface, 134, 148, 186, 287
GUI, 134, 148, 186, 187, 287, 288, 289, 290
GUI Design Elements, 287
GUI Framework, 134, 287, 288
Hierarchy, 27, 34, 35, 41, 43, 44, 46, 50, 51, 58, 62, 65, 81, 89, 97, 133, 136, 140, 141, 143, 144, 159, 176, 231, 233, 257, 260
High-Level languages, 9, 14, 15
History, 13, 56, 68, 73
Identity, 25, 33, 58, 108
Implementation, 27, 57, 64, 81, 91, 142, 148, 149, 162, 174, 176, 183, 201, 206, 219, 220, 221, 225, 227, 228, 229, 230, 231, 233, 244, 245, 252, 258, 288
Informal English, 54
Inheritance, 34, 41, 42, 43, 44, 59, 62, 63, 71, 80, 81, 82, 84, 86, 88, 89, 91, 95, 96, 98, 99, 103, 111, 136, 141, 144, 147, 152, 160, 167, 168, 176, 202, 209, 212, 213, 229, 230, 231, 232, 257, 260
Instances, 66, 67, 85, 106, 110, 111, 115, 153, 201, 215, 241, 242, 245, 254, 259
Integrated Development Environment, 287, 288
Integration, 216, 220
Interactions, 52, 81
Interfaces, 9, 10, 14, 24, 32, 39, 40, 47, 52, 57, 58, 92, 93, 95, 103, 132, 133, 134, 138, 139, 140, 147, 149, 150, 152, 182, 183, 186, 187, 194, 199, 201, 203, 205, 211, 215, 216, 227, 228, 229, 230, 231, 232, 233, 234, 235, 237, 238, 239, 240, 241, 242, 243, 244, 245, 247, 248, 249, 250, 251, 253, 258, 260, 263, 287
Interface of a class, 92
Interfaces in UML, 231
Interfaces vs. Implementation, 92
Interoperability, 149, 216, 244, 256
Java, 9, 10, 20, 25, 34, 44, 61, 62, 65, 66, 67, 85, 87, 107, 142, 228, 229, 230, 231, 232, 233, 234, 239, 245, 287
Key Abstractions, 54
Keyboard, 288
Language Features, 151
Layers, 133, 134, 241, 242, 249
Legacy Systems, 51, 147, 150
Lifecycle, 10, 17, 21, 221, 225
Maintain, 219
Documentation, 47, 57, 101, 152, 169, 198, 259, 261
System data, 219
Maintenance, 198, 221, 225, 259, 260, 262
Mechanisms, 215
Menu, 187, 188, 189, 194, 198, 199, 200, 258, 287, 288
Method Overloading, 138, 152
Method Overriding, 64, 65, 66, 88, 172, 176
Methodology, 15, 25, 27, 67, 256, 261
Methods, 17, 20, 25, 34, 39, 40, 49, 50, 54, 60, 61, 62, 63, 64, 65, 66, 67, 68, 74, 81, 82, 83, 84, 87, 88, 92, 93, 94, 95, 105, 106, 107, 108, 109, 123, 124, 125, 126, 127, 129, 133, 138, 139, 140, 142, 152, 154, 155, 156, 157, 158, 159, 163, 164, 165, 166, 167, 174, 177, 181, 182, 183, 184, 197, 201, 202, 203, 210, 211, 212, 227, 228, 229, 230, 231, 232, 233, 234,

235, 237, 241, 242, 243, 244, 245, 250, 251, 252, 260
Microsoft, 10, 34, 134, 222, 287
Model, 21, 23, 27, 30, 50, 51, 53, 55, 68, 104, 114, 115, 131, 147, 148, 161, 170, 174, 179, 198, 205, 209, 210, 211, 213, 216, 217, 234, 239, 246, 248, 262, 263, 266
Modeling Activities, 72
Model-View-Controller, 148
Modular, 16, 47, 48
Modularity, 15, 25, 27, 28, 34, 35, 46, 47, 48, 51, 58, 133, 257, 258, 260
Modules, 18, 40, 46, 47, 48, 213, 216, 219
Mouse, 288
Multiple Inheritance, 44, 100, 230, 232
Nodes, 241, 245, 246, 247
Non-functional Requirements, 22
non-object-oriented, 12, 20, 47, 48, 149
Oak, 20
Object-Oriented
Design, 228
Development, 232
Object-Oriented Databases, 262
Object-oriented languages, 9, 19, 44, 47, 49, 61, 92, 107, 152, 229
Object-Oriented methodology, 10, 12, 17, 27, 255
object-oriented methods, 20
Object-Oriented Technology, 261
OOA, 17, 25, 29
OOD, 17, 25, 29
OOP, 17, 25, 29
Paradigm, 24, 34, 39, 50, 60, 62, 93
Object-Oriented Design Goals, 143
Objects, 1, 9, 10, 12, 13, 16, 17, 19, 20, 21, 22, 24, 25, 27, 29, 30, 32, 33, 34, 35, 39, 40, 44, 45, 46, 47, 48, 49, 51, 52, 53, 54, 55, 56, 57, 58, 60, 61, 65, 66, 67, 68, 72, 73, 79, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 98, 99, 101, 103, 104, 105, 106, 107, 108, 109, 110, 111, 114, 115, 118, 119, 121, 123, 124, 126, 129, 130, 131, 132, 133, 134, 135, 136, 137, 139, 140, 141, 142, 143, 144, 145, 147, 149, 150, 152, 153, 154, 155, 156, 158, 159, 160, 161, 162, 163, 164, 166, 169, 170, 174, 175, 176, 177, 178, 179, 182, 183, 186, 187, 197, 201, 202, 203, 205, 207, 209, 213, 215, 216, 217, 219, 221, 225, 227, 228, 229, 230, 231, 232, 233, 234, 237, 239, 240, 241, 242, 243, 244, 247, 250, 251, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 267, 287
Objects
Object Oriented Analysis, 17
Objects
Object Oriented Analysis, 17
Objects
Object Oriented Development, 19
Objects
Object Oriented Programming, 56
Objects
Object Oriented Programming, 73
Objects
Object Structure and Relationships, 105
Objects
Object Initialization, 108
Objects
Object De-initialization, 110
Objects
Object Interactions and Relationships, 114
Objects
Object Diagram, 115
Objects
Object Diagram, 119
Objects
Object Diagram, 130
Objects
Object Oriented Analysis, 133
Objects
Object Factories, 147

Objects
Object Diagram, 204
Objects
Object Diagram, 213
Objects
Object Diagram, 213
Objects
Object Oriented Software Interfaces, 227
Operations, 10, 22, 38, 40, 46, 49, 52, 54, 92, 132, 138, 139, 140, 141, 147, 149, 150, 152, 153, 175, 179, 180, 181, 201, 202, 203, 210, 211, 212, 213, 216, 217, 227, 234, 242, 244, 248, 252, 258, 260, 263, 287
Operator Overloading, 152
Packages, 47, 207, 241, 243
Parameterized Classes, 153, 201, 213
Partitions, 46, 58, 219, 221, 222, 249
Pattern Scavenging, 212
Patterns, 131, 146, 147, 149, 207, 212, 213, 215, 217, 221
Persistence, 25, 27, 35, 48, 49, 51, 58, 133, 141, 142, 169, 244, 257
Persistence and Data Management, 169
Phases, 23, 25, 51, 53, 60, 72, 151, 186, 210, 215, 216, 219, 220, 221, 222, 226, 249
Platform, 26, 148, 217, 234, 249, 254
Pointers, 18, 20, 85, 103, 232
Polymorphism, 82, 87, 88, 91, 92, 99, 103, 105, 160, 229, 237, 257, 258, 260, 288
Defined, 82
Polymorphic behaviour and interfaces, 230
Primitiveness, 203
Private, 18, 93, 94, 95, 111, 123, 132, 140, 150, 154, 201
Process, 9, 21, 24, 54, 131, 132, 133, 135, 141, 187, 207, 208, 210, 211, 212, 218, 221, 222, 239, 249, 255, 261, 262
Products, 210, 211, 212
Protected, 93, 94, 95, 111, 140, 201
Prototyping, 186, 187
Public, 10, 49, 93, 94, 95, 111, 123, 132, 140, 147, 150, 155, 163, 177, 183, 201, 228, 232, 233, 234, 235, 237, 241
Rational Rose, 10, 262
Rational Unified Process, 10, 255, 261
References, 9, 85, 103, 183, 229, 230, 232
Refining Class Selections, 53, 134, 205, 215, 216
Relational, 57, 142, 147, 149, 150, 169, 170, 174, 249, 250, 262, 263
Relational Databases, 149
Relationships, 81, 266
Reliability, 23, 40, 47, 141, 143, 144, 148, 205, 217
Remote Method Invocation, 234, 239
Reports, 198
Requirements, 22, 56, 179, 208, 221, 256
Requirements Gathering, 208, 256
Reusability, 143, 205, 217
Reuse, 257
RMI, 234, 239
Robustness, 23
Run-time, 23, 47, 48, 49, 60, 88, 92, 105, 111, 114, 130, 132, 142, 203, 241, 242, 243, 244, 257
RUP, 261, 262
Scenario Walk-Through, 211
Searching for Students, 194
Second-generation languages, 15
Security, 20, 23, 33, 100, 244
Sequence, 115, 116, 118, 129
Sequence diagram, 115, 118, 129
Sequence Diagram, 116
Serialization, 48
Server, 147, 148, 149, 227, 234, 242, 246, 247, 252, 259, 260, 263
Sets, 29, 152
Shared memory, 24
Shlaer/Mellor, 51

Simula, 19, 20
Single inheritance, 41
Single Inheritance, 41
Smalltalk, 20
Software Architecture, 239
Software Development Process, 207
Software Engineering, 9, 12, 17, 68, 132, 215, 217
  Goals, 132, 215, 217
  Maximal levels, 215, 217
Specification, 105, 213, 245, 256, 261
State, 25, 34, 48, 49, 51, 58, 77, 107, 108, 121, 122, 125, 129, 130, 142, 157, 161, 165, 170, 171, 174, 180, 204, 242, 245, 256
State diagram, 121, 129
State Diagram, 122
Static, 21, 107, 108, 114, 115, 132, 163, 259, 267
Static Views, 21, 114, 267
Strategic Decisions, 148
Strategic Decisions
  Strategic vs. Tactical Decisions, 148
String Classes, 152
Structure of an object, 105
Structured Analysis, 54
Subclasses, 31, 43, 64, 65, 82, 84, 88, 92, 103, 160, 229, 230
Sufficiency, 203
Sun, 10, 20, 245
Superclass, 43, 44, 63, 64, 65, 66, 82, 88, 92, 103, 136, 140, 141, 176, 229, 230, 232, 288
System Development Processes, 207
Technology, 10, 256, 262
Tier 1, 250
Tier 2, 250, 251
Tier 3, 250, 251
Tiers, 242, 249, 250
Trademarks, 10
Typical classes, 30
Typical objects, 33
UML, 9, 10, 67, 68, 69, 70, 71, 72, 79, 84, 115, 116, 118, 119, 120, 122, 129, 231, 232, 246, 262
UML Notation, 68, 69, 70
Unit test, 219
Unix, 255, 287
Usability, 23, 27, 186, 265
Use case, 22, 53, 54, 115, 118, 262, 265, 266, 267
Use Case Models, 266
Use cases, 54, 266, 267
Use Cases, 265
Use-Case Analysis, 53
User Interface, 186
User-defined type, 16, 66, 85, 234, 242
Virtual, 65
Visual Basic, 34
Visual Studio, 287
What is an object?, 105
Windows, 10, 255, 287, 291
Wrappers, 147