1 Introduction

This is the first set of lecture notes that will supplement the course. Since our only text book is on Java, we will have to rely on these notes for studying principles of object oriented (OO) system analysis and design.

The OO concept emerged gradually in the early seventies both in the area of design of hardware architecture and in software development. The main reason for this emergence was to cope with complexity of systems. We need to state exactly what we mean by complexity as it is used in many contexts for different meaning.

2 What is complexity?

In the area of software engineering the notion of “complexity” pops up in at least two distinct contexts. Two better understand this let me set up a scenario:

Example 1 (The currency arbitrage firm problem) The ACME currency arbitrage company has worldwide offices with interconnected computers distributed in each of these offices. The central office of the company is in New York. Every office at a given moment knows the exchange rate of the local currency with respect to all other currencies that the company has offices in. Also local taxes for currency exchange and transaction fees are known. The main job of this firm is to figure out if, starting from dollars in New York, it can find a sequence of currency exchanges so that it ends up with more dollars than it started with. Each transaction involves a buying office and an executing office. For example the New York office using dollars may wish to buy yen. Then New

\footnote{Arbitrage means making something from nothing with no assumption of risk whatsoever.}
York is the buying office and the Tokyo office will be the executing office; it takes the dollars wired by New York and and buys equivalent amount of Yen. These transactions have to take into account all taxes and fees. The operation will be managed by a software system which on the one hand has real time access to the current exchange rates in all locations and on the other hand processes and sends the orders through the company’s network to local computers.

The description above is fairly vague with many details missing. This is typical of initial project specification which is usually formulated by managerial and not technical staff. But even in its current form one can discern two kinds of difficulty.

### 2.1 Algorithmic complexity

On one hand the gist of the problem, stripped away form all “unessential” detail, can be formulated as follows: We have a network with nodes corresponding to currencies and with links from one currency say $A$ to another currency, say $B$ with a label indicating the exchange rate. See figure 2.1. The problem then is

![Figure 1: A modeling of the arbitrage firm problem. numbers on arcs indicate exchange rates](image)

to find if there is a **directed path** in the network from a specified currency, say dollar, which after a series of transactions leads back to the original currency with higher value. This formulation abstracts away from many of the details and focuses on the essential core of the problem. Nonetheless the solution of this problem is not quite obvious and the problem from **algorithmic point of view** has a certain level of complexity. Algorithmic or computational complexity deals with the amount of resources, mostly time and storage, that is required to solve a problem as a function of the size of the problem—in this case the number of different currencies that we are working with. In this case, it turns out, that
the problem is an instance of the generalized flow problem which is studied by computer scientists and the best known algorithms for it run approximate in the order of cube of the number of currencies, we say the time complexity of such algorithm is $O(n^3)$.

### 2.2 System related complexity

Another aspect of complexity in this problem is exactly in the “unessential” details. These include issues such as synchronization of activities of each office (they are not all open at the same time), the design of relationship among various aspects of the components especially in the distributed environment, interface to users in particular decision on who and at what level can alter the transactions, graphical user interface, mechanism of changing various applicable fees and taxes, and similar types of problems. Software that deal with these kinds of questions often contain many components with complex interactions among them. In addition software where there are a complex interface with users or other software fall into this category. We refer to this form of complexity as system-wide complexity. The focus of our course is on this latter type of complexity and less on the former, algorithmic, variant.

**Object oriented system design** defines a collection of rules and methodologies that should aid us cope and manage system wide complexity. Though it can be used in almost all forms of software design, it is most effective in understanding and design of large systems with many components having complex interactions.

### 3 The principles of object orientation

In the **object model** a system is considered as a collection of cooperating objects. Objects themselves are units that contain both the data and the operations to be performed on the data. This model is quite distinct from structural programming, where data and operations are completely separated. For instance suppose you plan to write a structured program using a programming language such as Pascal or C. The usual way—and the one encouraged by these languages—is to clearly define your data first and then think about all the functions and procedures that are to be applied to these data. In this view data and operations are completely separate items. Objects on the other hand are modules within which both data and operations on data are defined. Let us look at an example.

**Example 2 (maintaining a telephone book)** Suppose you want to maintain a telephone book where people and business names, telephone numbers and addresses are stored. Furthermore you wish to be able to look up a name, a telephone number, or an address in this phone book. You also want to be able to enter new items to it and delete outdated items. In structured programming approach you would for instance implement the telephone book itself as a data structure such as a balanced tree or a hash table, or some other structure that
3.1 Abstraction

can support the operations of lookup, insert and delete efficiently. Then you
go ahead and write up the code for these operations. This is diagrammatically
sketched in figure 2. Object oriented approach would define an item, an object,

![Data section]

```
Data section

Hashtable TBook;
```

![Operation]

```
Operation

insert();
delete();
lookup();
```

![Procedural or structured design]

```
Procedural or structured design
```

![Data fields]

```
Data fields

Hashtable TBook;
```

![Methods]

```
methods

insert();
delete();
lookup();
```

![Object based design]

```
object based design
```

This is more than a cosmetic difference. The outside world now, understands
this object simply as a telephone book which can be looked up, inserted in and
deleted from. No knowledge of details of implementation or data structuring
is required. In other words the outside world, that is whomever who wants to
use the telephone book object only needs to know what it does and not how
it does it. This emphasis on what rather than how is the pillar of the object
model. When applied correctly and diligently it facilitates coping with system
wide complexity.

Objects are instances of classes, or put another way, classes define behavior
of their objects. Thus for instance you may have a class called rectangle with
attributes such as height, width, position and methods such as getArea() and
isSquare(). Then particular instances of the rectangle class may be, say A, B
and C. Here Then A has its own height and width, which is distinct from other
instances.

The principles and rules of OO design can be stated in a fairly succinct
manner: we outline them below.

3.1 Abstraction

How many times has this occurred to you: You write a piece of code to do
something and a while later you need code very similar to the one you just wrote
except that the type and nature of data is changed, so you have to essentially write the same code for the new types?

Abstraction refers to the ability to include those properties of an object which are essential to and separate it from other objects. As part of practicing abstract design one has to delay as much as possible specifying details or committing to certain implementations, and allow those details to be implemented by objects that are extending the object under consideration. The following definition (and all other quoted definition in this section are from Booch’s text [?])

An abstraction denotes the essential characteristics of an object that distinguishes it form all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the user.

As a very simple and common example suppose you are trying to implement a sorting algorithm. Now a typical algorithm needs to compare data and rearrange them so that the result is in order. This procedure really does not depend on the kind of data you are trying to sort. Its logic is valid whether you are trying to sort integers, doubles, Strings or complex structures in databases with multitude of fields. Abstraction principle requires you to write your code in such as way that your code will not depend on the specific type of input and is applicable to all “sortable” set of data. To accomplish this you want to avoid specifying the types and even the details of the comparison operation. All you need to know is that some comparison operation will be defined in due time.

3.2 Encapsulation

Encapsulation is the principle of hiding information from users of an object. As such encapsulation is complementary to the abstraction principle. While abstraction avoids specifying the unessential information of the object, encapsulation hides them. Consider for example that for certain project we need the stack object. This object contains a collection of data plus the following operations:

1. access the top element of the stack (and no other element),

2. push a new element to the stack, so that the new element replaces the top

3. and pop the top element from the stack.

Such a data structure is quite useful and appears in many applications in computer science. Exactly how each of these operations are implemented should be hidden form all those objects that are going to use the stack object. Whether the data in the stack are implemented as an array or linked list or any other structure, or what is the exact implementation of access, push and pop operations is irrelevant to an object wanting to use stack. What is relevant is the expected behavior of stack and operations it supports. Encapsulation makes
the details of implementation only the business of the internals of the stack object inaccessible to outside objects. The advantages of encapsulation are many. First, by creating a barrier between the user of an object and its internals we can prevent inadvertent (or even malicious!) modification of data. Second, if in some future time we decide that we like to change implementation of stack we can be rest assured that no other objects need be modified, as long as we keep the interface of the stack to its user consistent.

### 3.3 Modularity

Modular property is the decomposition of a system into a collection of pieces, modules if you will, that are cohesively and loosely coupled with each other. Modules are thus created by packaging related objects together. Of course there are many different ways of putting objects together. The trick is to do it so in a manner that allows modules to be designed and revised independently of each other. Also a rule of thumb is that modules should be small enough to be fully comprehended. Ideally you wish to design your modules so that after possible modification only the objects and classes of that module is recompiled and linked. This means that you should avoid changing the interface of a module as much as possible. Changing the interface may necessitate changing and recompiling all other modules that re dependent on that interface which may cause changing modules that were depend on them and so on.

As an example suppose you are designing a calculator with a graphical user interface (much the same way the calculator accessory in Windows or the xcalc routine in X window system). Here there are many objects but we can recognize at least two broad modules. First is the collection of objects that realize the operations of the calculator such as add, multiply etc, entering data and various functions like sin, cos and such. Another module is the graphical objects such as the buttons, their colors the font of the labels on the buttons, and the like. Now if we decide to change the design or arrangement of buttons on the calculator graphical user interface (GUI) that would not have much of an affect on the objects that implement the mathematical objects of the calculator. Conversely if we decide to add new functions to the calculator, the graphical part should not be affected that much. Of course you may need to add new buttons for the new functions, but the point is that adding a new function has virtually no effect on how the button object is implemented.

### 3.4 Hierarchy

In abstraction we attempt to leave unspecified the details as much as possible. The details however, have to be specified sooner or later. The principle of hierarchy is the act of revealing more and more details in steps. To give an example take the general category of employees of a company as an abstract object. Suppose for each employee we like to know their name, gender and date of hire. Now, at a different level we need to know a bit more we may have three categories of employees: managers, technical and administrative. Now each of
these categories are special kinds of employees and as such they already have employee attributes like name, gender and date of employment. In addition for managers we may list the office number, the secretaries working for them, and departments they manage. Technical employees may have cubicles location, department they work for and the current projects they are involved with. Administrators may have also departments, and may be assigned to duties such as payroll, secretarial, custodial and the like. We can push this further. Managers may have categories of CEO, VP’s middle level and so on, each with their own new attributes. Technical staff may be categorized further into programmers, system administrators, scientists, chemists etc. The point is that at each level we only add or modify properties of the object needed at the level, and we inherit all the other properties of the parent object.

What we just described is a special kind of hierarchy principle called inheritance. This principle is used when we want to encode the IS-A relationship. For instance a manager is an employee; a scientist is a technical staff which in turn is an employee.

Another form of hierarchical design is called composition, sometimes called aggregation. Here we create new objects by wholly or partially putting together other objects. In our example above, suppose that we also have another object which is called compensation which may contain information such as the salary amount, benefits, bonus, whether it is monthly, biweekly or hourly and so on. Then we may wish to add to the employee class another attribute the compensation object. The date property of the employee object may be itself a date object with its own properties and operations. Composition represents the IS-PART-OF relationship among objects.

### 3.5 Polymorphism

Polymorphism is the ability to request that the same operation be done by many different types of objects. However the way the request is handled depends on the object it is being sent to. For example consider our sorting object mentioned above. A sorter may request a comparison of two items. If this request is sent to integers then may be a less than or equal to test is performed. If it is sent to strings then the comparison may be alphabetical order. We could also write the comparison to implement a greater than or equal to rather than a less than or equal to relationship. The requesting object, here the sorter, really does not care how the comparison object goes about doing its business. It only needs the results of the comparison. Together with abstraction, encapsulation and hierarchy, polymorphism is responsible for the power of object oriented design in organizing, making sense of, and implementation of a large and complex system.
4 The software life cycle

The software life cycle refers to various phases of software development from its inception as an idea all the way to its release for use. We focus on the part that the software is actually being built. At this stage we have the following tasks:

4.1 Object Oriented Analysis (OOA)

This is the conceptual modeling stage. At the analysis level the users and designers of the system come together and precisely articulate the functionality of the system. For instance you may wish to design a complex web site for buying merchandise over the net. At the analysis level you specify things such as, the security requirements of the site, Unique ID, password and other identification information, registration mechanism for new clients, a shopping cart mechanism for multiple purchases, payment mechanism, possible escrow arrangements and the third party administering the escrow, possible automatic tracking of the stage of product processing (for example is it still being ordered, is it being built in the factory, or is it already delivered, say to UPS, with so and so tracking number). A good analysis sticks with the requirements and functionality of the system and will avoid design decisions (see below). While there are many forms of analysis for large systems the most relevant for our purposes is the object oriented analysis. In this method, requirements are presented in the form of objects and classes and the relationship among these objects.

4.2 Object Oriented Design (OOD)

In the design phase we emphasize the proper structuring of the system. At this stage we focus on proper decomposition of the system and we model the proper logical and physical and static and dynamic aspects of the system. In object oriented design we use classes and abstraction to logically structure the system. Object oriented design is probably the most important aspect so the software life cycle. It is at this level the system takes its shape.

4.3 Object Oriented Programming (OOP)

Object oriented programming is the method of implementation in which programs are organized as cooperative collection of objects, each representing a class, which in turn are organized in hierarchy of classes united by inheritance. Generally OOP is done through a language that facilitates this practice. It is possible to adhere to OOP principle by using non object languages such as Pascal or C. However this may take an exceptional effort. On the other hand languages such as Java, C++, Smalltalk, and Eiffel have the necessary facilities to enable OOP. These include support for classes and object, abstraction, overwriting and overloading of operations and dynamic typing.
4.4 The life cycle

Generally speaking one starts with analysis. Then the output of the analysis will serve as a model for design. Finally the output of design is sent to implementation phase. In reality however, things don’t work quite neatly like this. Rather, at the design stage one can discover shortcomings or even inconsistencies of the analysis phase. Similarly at the programming level, one can find difficulties with the design level which may in turn reveal further problems with the analysis stage. This give-and-take may continue all the way to the completion of the project. The software development lifecycle, then resembles more realistically like figure 4.4. Notice that two other stages are present. The preliminary stage of business planning, where one has to justify, budget, allocate manpower, and develop a schedule. Finally the last stage is to deploy the software to the users. Even at this stage interaction with previous stages may be required: bugs may be found, or flaws in design and user interface may surface, or after deployment it may be realized that certain functionality need be added or completely redesigned.

Figure 3: Software development lifecycle