Introduction

There are two very basic questions which one seeks to answer throughout a curriculum for computer science.

- How to write the programs that give correct answers for all inputs.
- How to make sure that no better program could written.

These questions have direct relevance to good software designs. They constitute what may be termed as the high-level software design goals.

Introduction

- The first question is of generality of correctness. Obviously, we don't want to design such software that can only work on specific predefined inputs. If that is the goal then one may some how painstakingly assemble the solution once and use it for all the times.
- So, if a software were to be usable then basic high level design goal should be that it produces correct output for any valid input.

Introduction

- It is not always easy to find a complete answer to the second question. This refers to issue of efficiency of the software.
- Efficiency is closely related to computational resources.

  For example, multiplication may be more costly than shift operation on some computer. Then multiplication should be realized in terms of shifts.

Introduction

- It is not always easy to find a complete answer to the second
- The second question also relates to types of valid input which have to be processed.
- So in a very high level of abstraction, efficiency raises issues of constraints and optimizations.
- How to evolve a framework which can assure that the design indeed works most efficiently than other possible approaches.

  The most fascinating aspect of an applicable mathematical framework to the study of computer science known as *complexity of computations*.

Introduction

There is also a another set of less basic yet equally important questions. It includes

- how to write portable programs,
- how to write scalable programs,
- how to write reusable programs,
- how to write readable programs.
- how to write big programs,

These together can be termed as aspects of low-level software design goals.

Low-level Software Design Goals

- Program adaptability, reusability, readability, and modularity mainly help to manage large software projects and collaborations.

  The point we like to emphasize through the above set of questions is maintainability of software begins with design initiation and not after the implementation of software. In fact, the *legacy* and the *reverse engineering* problems would not arise if these aspects are considered with desired importance.

- Portability and scalability contribute to robustness which means program behaves correctly in any platform with any input size and on any hardware. In a way robustness is correctness plus something more. The subtle difference lies in portability and scaling.

Reusability

- Reusable programs make our life lot easier. We don't have to write a new program *ab inito* every time.

  For example, we could make an abstraction of generic requirements of a program and write the reusable code for these generic abstractions.

- Essentially, it forces us to decompose program in form of a collection of a set of generic components.

- As the specification of components is domain independent, it enables us to reuse substantial pieces of code developed for an entirely different application with minimal effort.

- Clearly, reusability also addresses scalability, adapability and modularity.

Object Oriented Paradigm

- *Object oriented programming* paradigm is introduced to take care of all these issues.
- Object oriented design is introduced with motive to instill a sense of clean programming practice and to address the requirements of code reusability, modularity and maintainability among others, as discussed earlier.
- The terminology of clean programming practice needs to be amplified. Basically, object oriented programming paradigm formulate a set of ground rules and principles for organizing a software design.

Object Oriented Paradigm

These can be broadly described as follows:

- `Abstraction.` Method of filtering out the details from operational semantics required for a software design.
- `Encapsulation.` A way to hide implementational details of one component from the other.
- `Modularity.` This refers to organizing software into functional units based on hierarchical relationship.

Abstraction

Filters out the details to expose operational semantics.

- Consider the design of a document processor. *Search and replace* seems more intuitively obvious operation from a user prospective.
- But implementation of this operation quite involved.
- High level abstractions tend to make the design and decomposition of software intuitively clear
- Helps additionally to isolate encapsulation features.

Encapsulation

Information hiding.

- Abstraction reveals the component decomposability of the software into components.
- the principle of encapsulation tells that implementation details of each component should be kept as much within itself as possible.

Encapsulation

Information hiding.

- The component interaction may require a component to demand change in internal state of one or more components. But there is no need to let the interacting components to look at each others state.
- They may behave like negotiating parties. Each maintaining its own state and changing it if the the state change demanded by the other is agreeable to itself.
- This means each component should expose only that much which is required for interfacing with the other components. Therefore encapsulation is information hiding.

Modularity

Organizing S/W into functional units based on hierarchical relationships.

- This refers to organizing software into functional units based on hierarchical relationship. Two different types of hierarchical relations are possible:
  1. *is_a* and
  2. *has_a*.
- *Has_a* relationship expresses the *composition hierarchy*. For example, a computer *has_a* CPU.
- Whereas *is_a* relationship is used to express *membership hierarchy*. For example, a CPU *is_a* processor.

Modularity

The advantages of modularity and hierarchical design are

- Common functionality can be defined at the most general level
- Specialized behaviour or extension of common functionalities can be derived from general functionalities by inheritance and polymorphism.

Object Oriented Design Terminology

All object oriented programming languages like JAVA or C++ include techniques and constructs that help in viewing the components of abstractions as objects. The basic constructs are

- `Objects and Classes.` Classes define abstract specification of an object. Or equivalently an object is a specific instance of a class. For example, the set of all computers can be defined by a class called *Computer*. But an object called `my_Computer` may represent just an instance of the class *Computer*.
- An object is completely specified by its *state* and *behaviour*. The state is specified by the values of the internal variables. The behaviour is expressed by a set of methods or functions.

Object Oriented Design Terminology

- Objects interact by message passing. For example, object o_1 interacts with o_2 by requesting the latter to
  - Print itself.
  - Change its state.
  - Return its specific data fields.
- Object o_1 can directly change object o_2's state if the latter has explicitly allowed such an access.

Object Oriented Design Terminology

- `Interfaces and Strong Typing.` Each class must present an Application Programming Interface (API) for other objects (belonging to other classes or the outside world) to interact with objects of the class.
- API is presented in form of a collection of methods in the class definition.
- Each method will have arguments with specific types.
- Interpreter or compiler enforces rigid type checking at runtime. Programmer has to take care of the issue of strong typing at the time of writing the code.

Object Oriented Design Terminology

- `Inheritance & Polymorphism.` Help in managing composition and membership hierarchies. So they help in abstraction (generalization), implementation (specialization) and thus reusability.
- A `base class` or super class is the most general class appealing closer to intuitive operational semantics.
- A `derived class` is a class which inherits all the properties of a `base class` and includes more methods and/or state variables than the `base class`.
- `Polymorphism` literally means *many forms*. A polymorphic object is an entity that can hold different types of values during the course of execution.

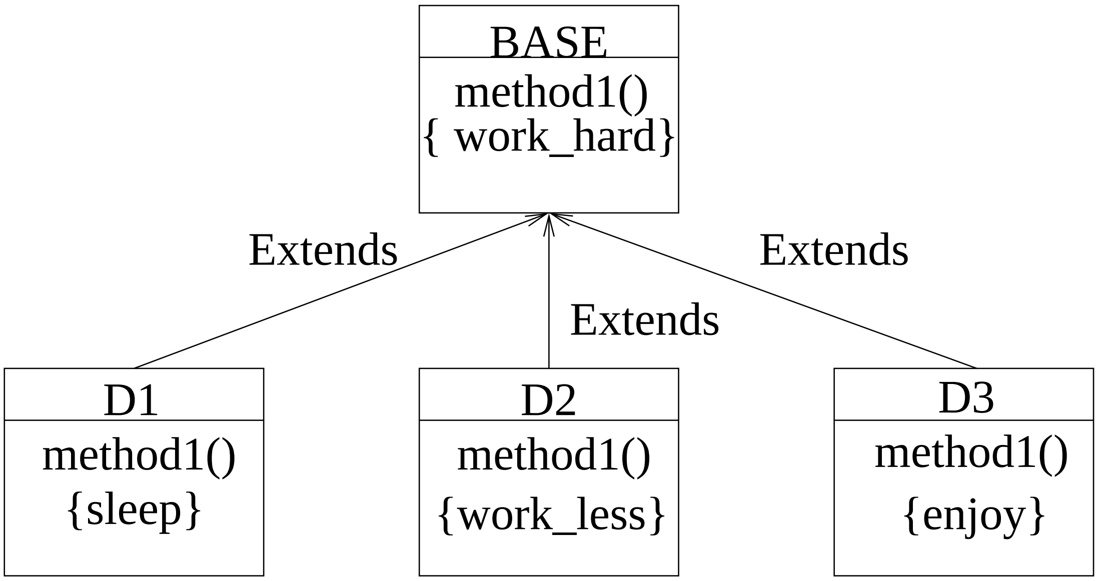
More about Polymorphism

- Strongly typed language exhibit polymorphism by *overloading* and *virtual functions*.
- Pure polymorphism occurs when a single function can be applied to arguments of arbitrary type.
- A strongly typed language requires the argument types to be specified for each form of a single function in order to exhibit polymorphism. This kind of polymorphism is known as ad hoc polymorphism.
- Essentially a number of functions with same name but a varying number of arguments and argument types are written. The function instance called is recognized by its *signature*. This mechanism is called *overloading*.
- A function name is said to be overloaded if there are two or more function bodies with same name.

Overriding

- It is a mechanism of *refinement and replacement*
- A derived class may override some method of the base class by using the same name for it, but replacing the arguments as well the method body by a different logic.
- The overriding is transparent to user of a class. In that respect it is similar to overloading. In other words, overloading is the mechanism by which both overriding and polymorphism is exhibited in JAVA.
- Figure shown next illustrates polymorphism and overriding.

Polymorphism Example



- Depending on which class `D1`, or `D2`, or `D3` the object o refers to `sleep_more`, or `work_less` or `enjoy` is executed.
- On the other hand if o refers to an object of class `Base` then only `work_hard` can be executed.

Polymorphism Example contd.

- It means the method of most restrictive class is executed. Therefore, caller of *o*.`method1()` does not need know to whether object *o* refers to class `D1`, or `D2`, or `D3` or `Base`.
- It implies that the same method assumes different forms at different situations.
- Advantages of a specialized class are, it
  1. inherits generic methods of base class, and
  2. redefines some of the methods of base class to take care of its additional properties.

Abstract Data Structures

- Abstract Data Structures (ADS) close the semantic gap between intuitive program constructs (programmer's view) and program constructs in a programming language.
- In a sense it provides a virtual programming language. For example, JAVA class definitions provide Abstract Data Types (ADT) on which ADS can be built.
- There is a common set of operations on each ADT. The list is provided in the next transparency.

Common Set of Operations

- `IsFull()`
- `IsEmpty()`
- `Size()`
- `PositionOf(x)`
- `ValueAt(i)`
- `InsertValue(x,i)`
- `ReplaceValueAt(i,x)`
- `DeleteElement(i)`
- `DisplayADT()`

JAVA Tutorial: Class & Object

```
[<class_modifier>] class <class_name>
    [extends <superclass_name>]
    [implements <interface1>, <interface2>, ... ]
    {
        //class methods and state variables
    }
```

Class modifiers are options to modify the scope and the status of a class.

- `abstract` describes a class that has abstract methods.
- `abstract` methods have signature but empty body.
- class consisting of only `abstract` methods is called an `interface`
- `abstract` class is a mix of both *actual* and `abstract` methods.

Class, Object & Instance Variable

- `Final` class modifier is used when no class can be derived from a class.
- `public` class modifier describes a class that can be instantiated any where in the same package or anything that imports the class.
- All `public` classes are declared in separate files called `<class_name>.java`.
- A new object is created by `new` operator. It returns a reference or handle to an object of the required class.
- Default modifier interprets the class as `friendly`. All classes in the same package can instantiate and use it.
- Classes can define *instance variables* or *fields*. They represent state of an object when the latter is instantiated.