CITS2210

Object-Oriented Programming

Topic 2.
Introduction and Fundamentals:
Abstraction

Summary: This topic considers the fundamental concept of abstraction, and how object-oriented languages support abstraction.

[These slides are based on those supplied by Tim Budd to complement chapter 2 of "An Introduction to Object-Oriented Programming".]

Outline

Roadmap

In this chapter we will consider abstraction, which is *the* most important tool used in the control of complexity.

We will examine various abstraction mechanisms

We will present a short history of the development of abstraction tools.

Abstraction

Abstraction is the purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structure.

Information Hiding

Information hiding is the purposeful omission of details in the development of an abstract representation.

Information hiding is what allows abstraction to control complexity.

Abstraction in an Atlas

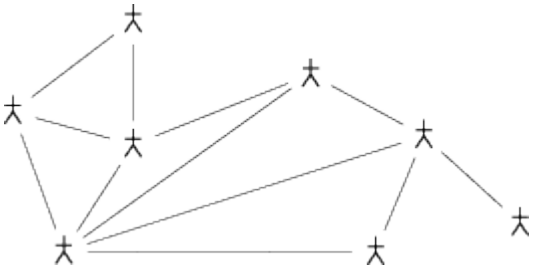
Think of an atlas, and the various different levels of maps

- A map of the world, contains mountain ranges, large political boundaries
- A map of a contenent, contains all political boundaries, large cities
- A map of a country, contains more cities, major roads
- A map of a large city, roads, major structures
- A map of a portion of a city, buildings, occupants

Each level contains information appropriate to the level of abstraction.

Levels of Abstraction in OO Programs

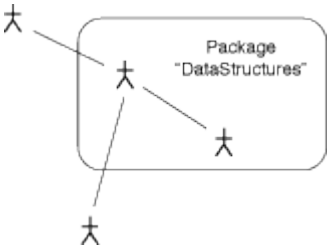
At the highest level of abstraction we view a program as a community of interacting objects.



Important characteristics here are the lines of communication between the various agents.

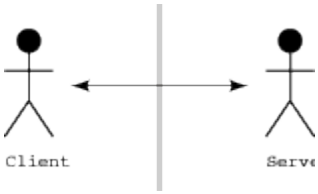
Abstraction in OOP -- Packages and Name spaces

The next level of abstraction is found in some (but not all) OO languages. A package, Unit or Name Space allows a programmer to surround a collection of objects (a small community in itself) with a layer, and control visibility from outside the module.



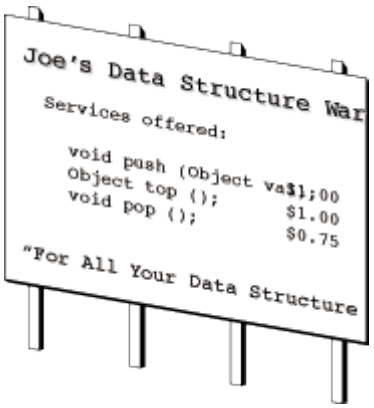
Abstraction in OOP -- Clients and Servers

The next level of abstraction considers the relationship between two invidual objects. Typically one is providing a service, and the other is using the service.



Abstraction in OOP -- Description of Services

We can next examine just the person providing a service, independent of the client. We define the nature of the services that are offered, but not how those services are realized.



Levels of Abstraction in OO -- Interfaces

Interfaces are one way to describe servies at this level of abstraction.

```
interface Stack {
        public void push (Object val);
        public Object top () throws EmptyStackException;
        public void pop () throws EmptyStackException;
}
```

Levels of Abstraction -- An Implementation

Next we look at the services provided, but from the implementation side:

```
public class LinkedList implements Stack ... {
        public void pop () throws  EmptyStackException {
           ...
        }
        ...
}
```

Concern here is with the high level approach to providing the designated service.

Levels of Abstraction -- A Method in Isolation

Finally, we consider the implementation of each method in isolation.

```
public class LinkedList implements Stack ... {
        ...
        public void pop () throws  EmptyStackException {
                if (isEmpty())
                        throw new EmptyStackException();
                removeFirst(); // delete first element
        }
        ...
}
```

Every level is important, and often you move quickly back and forth between levels.

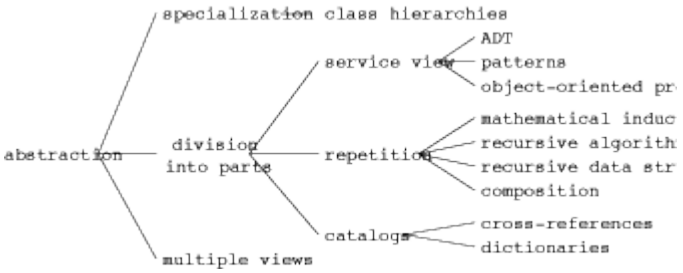
Finding the Right Level of Abstraction

A critical problem in early stages of development is to determine what details are appropriate at each level of abstraction, and (often more importantly) what details should be omitted.

One does not want to ignore or throw away important information

But one does not want to manage too much information, or have the amount of information hide critical details.

Forms of Abstraction



Is-a and Has-A abstraction

Two of the most important types of abstraction are the following:

- Division into parts -- Has-a abstraction
- Division into specialization -- Is-a abstraction

Has-A Abstraction

Division into parts takes a complex system, and divides into into component parts, which can then be considered in isolation.

Characterized by sentences that have the words ``has-a''

- A car has-a engine, and has-a transmission
- A bicycle has-a wheel
- A window has-a menu bar

Allows us to drop down a level of complexity when we consider the component in isolation.

Is-a Abstraction

Is-a abstraction takes a complex system, and views it as an instance of a more general abstraction.

Characterized by sentences that have the words ``is=a''

- A car is a wheeled vehicle, which is-a means of transportation
- A bicycle is-a wheeled vehicle
- A pack horse is-a means of transportation

Allows us to categorize artifacts and information and make it applicable to many different situations.

Encapsulation and Interchangeability

An important aspect of division into parts is to clearly characterize the connection, or *interface*, between to components.

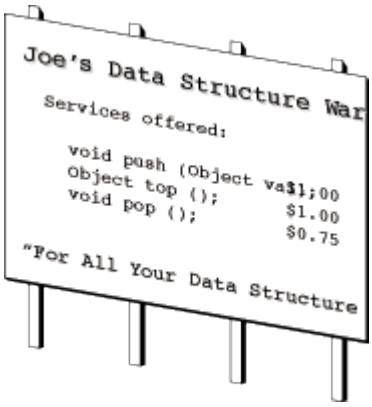
Allows for considering multiple different *implementations* of the same interface.

For example, a car can have several different types of engine and one transmission.

The Service View

Another way to think of an interface is as a way of describing the *service* that an object provides.

The interface is a contract for the service--if the interface is upheld, then the service will be provided as described.



Other Types of Abstraction -- Composition

While is-a and has-a are two important types of abstraction, there are others.

Composition is one example; a form of has-a; characterized by the following

- Primitive forms
- Rules for combining old values to create new values
- The idea that new values can also be subject to further combination

Examples include regular expressions, type systems, windows, lots of other complex systems.

Patterns

Patterns are another attempt to document and reuse abstractions.

Patterns are description of proven and useful relationships between objects; which can help guide the solution of new problems.

Example pattern, Proxy:

Will have many more patterns in a later chapter.

A Short History of Abstraction Mechanisms

Another way to better understand OOP is to put it in context with the history of abstraction in computer science.

- Assembly languages
- Procedures
- Modules
- ADT
- The Service View
- Objects
- The future....

Assembly Languages

Assembly languages and linkers were perhaps the first tools used to abstract features of the bare machine.

- Addresses could be represented symbolically, not as a number.
- Symbolic names for operations.
- Linking of names and locations performed automatically

Procedures and Functions

Libraries of procedures and functions (such as mathematical or input/output libraries) provided the first hints of *information hiding*.

They permit the programmer to think about operations in high level terms, concentrating on *what* is being done, not *how* it is being performed.

But they are not an entirely effective mechanism of information hiding.

Information Hiding -- The Problem of Stacks

```
int datastack[100];
int datatop = 0;

void init()      // initialize the stack
{ datatop = 0; }

void push(int val)   // push a value on to the stack
{ if (datatop < 100)
                datastack [datatop++] = val; }

int top()   // get the top of the stack
{ if (datatop > 0)
                return datastack [datatop - 1];
        return 0; }

int pop()   // pop element from the stack
{ if (datatop > 0)
                return datastack [--datatop];
        return 0; }
```

Where can you hide the implementation?

Modules

Modules basically provide collections of procedures and data with import and export statements

Solves the problem of encapsulation -- but what if your programming task requires two or more stacks?

Parnas's Principles

David Parnas described two principles for the proper use of modules:

- One must provide the intended user of a module with all the information needed to use the module correctly, and with *nothing more*.
- One must provide the implementor of a module with all the information needed to complete the module, and *nothing more*.

Abstract Data Types

An *Abstract Data Type* is a programmer-defined data type that can be manipulated in a manner similar to system-provided data types

- Must have the ability to instantiate many different copies of the data type.
- Data type can be manipulated using provided operations, without knowledge of internal representation.

But ADTs were important not because they were data structures, but because they provided an easily characterized service to the rest of an application.

Three Eons of History

Looking at this history, we can separate it into three periods of time

functionality of an application
data types used in an application
services provided by objects in the application

Objects - ADT's with Message Passing

Characterists of Objects

- Encapsulation -- similar to modules
- Instantiation -- similar to ADT's
- Messages -- dynamic binding of procedure names to behavior
- Classes -- a way of organization that permits sharing and reuse
- Polymorphism -- A new form of software reuse using dynamic binding

---

What Does the Future Hold

What will be the next evolutionary step in software?

Prediction is hard, particularly about the future.

However, one you have accepted the idea of an application formed from interacting agents, there is no reason why those components must exist on the same computer (distributed computing) or be written in the same language (components).

So some of the trends we see today in software are natural results of the OOP mind set.
