Object Oriented Software Design

Introduction to Object Oriented Programming

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

September 23, 2010

Outline

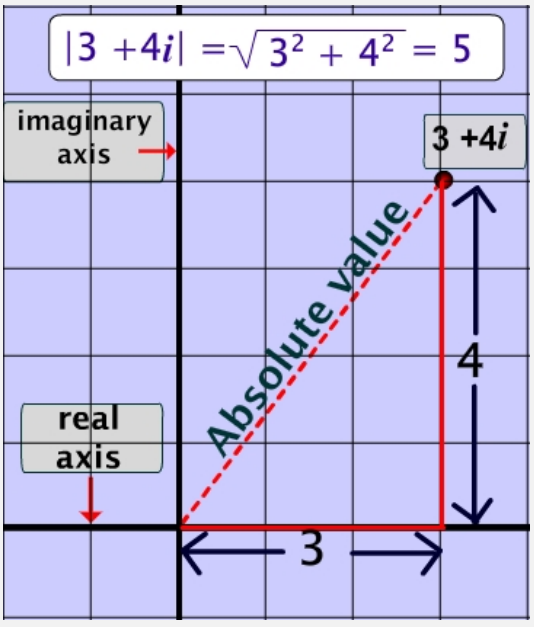
Abstract data types

- An important concept in programming is the *Abstract Data Type* (ADT)
- An abstract data type is a user-defined type, that can be used similarly to built-in data types
- An ADT defines
  - What kind of values the data type can assume (*domain*)
  - What operations we can perform on the data type
- How the data and the operations are implemented is *hidden* to the user, and it is part of the implementation

Example of ADT

- Most programmign language do not include primitive types for representing complex numbers

- In mathematical analysis, a *complex number* consists of a pair of real numbers
- The first one is the *real part* of the number
- The second part is the *imaginary part*
- The solutions of a polynomial equation can be real or complex numbers



ADT of Complex

- Complex numbers have a **domain**: the set of all pairs of real-numbers
  - In this case, the domain is continuous and infinite, but can be discrete and finite as well
- The **set of operations** that can be done on complex numbers are similar to the set of operations you can perform on real numbers
  - Addition, subtraction, scalar multiplication, etc.
- Other operations
  - Extract the real or the imaginary part, compute the modulo (lenght of the vector), compute the angle with the x-axis
- Therefore, we can represent a complex number as an element of an Abstract Data Type: **Domain** + **Operations**

Why abstract?

Why we say that complex is an ADT? Why abstract? It looks very concrete!

- "**Abstract**" here is used to indicate that we do not specify the implementation details. We are not saying how the number is represented internally, or how the operations are performed. The type is still abstract:
- It becomes **concrete** when we implement it in a programming language
- We can implement an ADT in any language: however, OO programming languages have special support for defining ADTs.

ADT in programming languages

Trying to implement ADT in low-level languages like C can be cumbersome

- Usually, this is done using structures for representing the domain, and functions to represent operations
- However, there are a lot of details to take care of:
  - name clashing (global namespace for functions)
  - manual type checking
  - syntax
- It can be done (and **it is done**, for example in OS kernel programming!), but it requires a lot of extra effort and extra care

Example of implementation of Complex in C

Here is an example of implementation of the complex ADT in C.
For every operation, you must define a function that takes a pointer to the structure

complex.c

```
struct complex {
    double r;  // real part
    double i;  // imaginary part
};

void
cmplx_init(struct complex *c, double r_, double i_);

void
cmplx_addto(struct complex *c1, struct complex *c2);

void
cmplx_subfrom(struct complex *c1, struct complex *c2);

void
cmplx_mult(struct complex *c, double m);
```

ADT and OOP

- Creating abstract data types (**classes**) is a fundamental concept in object-oriented programming.
- Abstract data types work almost exactly like built-in types
  - You can create variables of a type (called **objects** or instances in object-oriented parlance)
  - and manipulate those variables sending them *messages* or *requests*

Complex in Java

This is the interface of the Complex class in Java.

examples/Complex.java

```
class Complex {
    private double real = 0;
    private double img = 0;

    public Complex() {...}
    public Complex(double r, double i) {...}

    public addto(Complex b) {...}
    public subfrom(Comple b) {...}
    public mult(double m) {...}
};
```

Complex in C++

This is a (more complete) interface of the Complex class in C++

complex.h

```
class Complex {
    double real_;
    double imaginary_;
public:
    Complex();
    Complex(double a, double b);
    Complex(const Complex &a);
    ~Complex();

    double real() const;
    double imaginary() const;
    double module() const;

    Complex &operator=(const Complex &a);
    Complex &operator+=(const Complex &a);
    Complex &operator-=(const Complex &a);

    const Complex& operator++();            // prefix
    const Complex  operator++(int);         // postix
    const Complex& operator--();            // prefix
    const Complex  operator--(int);         // postix
};
```


Definition

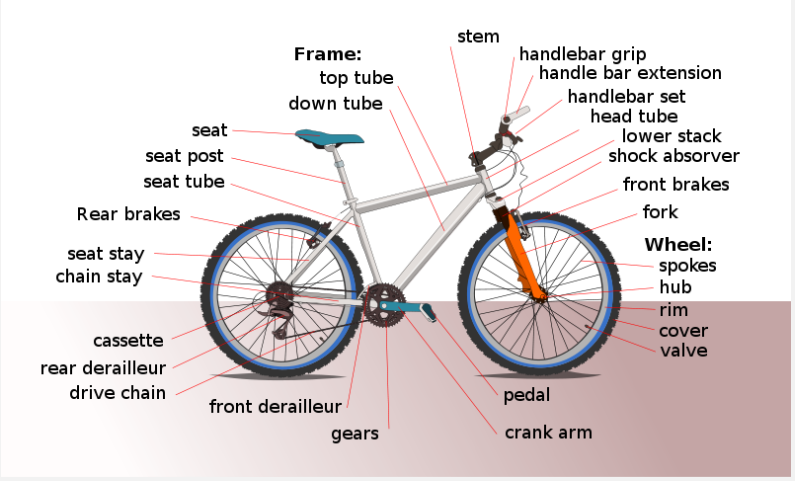
Now that we have introduced the concept, we can give a more formal defintion of ADT

- **Definition (Abstract Data Type)**: An abstract data type (ADT) is characterized by the following properties:
  1. It exports a type
  2. It exports a set of operations. This set is called interface
  3. Operations of the interface are the one and only access mechanism to the type's data structure
  4. Axioms and preconditions define the application domain of the type
- Notice that point 3 cannot be really ensured in any programming language that allows arbitrary casting from one type to another: if you know (or guess) the internals of an object, you can always access them by casting to the appropriate type
- More specifically, the C language allows to access the internals of a structure, while C++ allows arbitrary casting

ADT in the real world

ADTs can be used to represent not only numerical data, but also more "real" data types.

- Let's model a bicycle



Bycicle model

- Real-world objects share two characteristics: They all have **state** (the *domain*) and **behavior** (the *operations*)
  - Bicycles have state (current gear, current pedal cadence, current speed)
  - and behavior (changing gear, changing pedal cadence, applying brakes)
- Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming

Modeling the real-world

- Take a minute right now to observe the real-world objects that are in your immediate area.
  - What possible states can this object be in?
  - What possible behavior can this object perform?
- real-world objects vary in complexity:
  - your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off),
  - your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune).
- some objects, will also contain other objects (the bicycle is composed of wheels, brakes, gears, etc.)

Abstraction

How far we must go into the description of the objects?

- In the description of the bicycle we must identify what are the useful details that are interesting for our program and *hide* all non-interesting details
  - For example, the color of the bicycle may or may not be important
  - It is important if we are modeling a catalog for a bicycle store: the customer wants to know about the color, so we must be able to know which color we have in our store
  - It is not so important if we are simulating the bicycle dynamics in physical simulation, because the color does not impact on the simulation parameters
- Therefore, it is important to select the right level of detail

Level of abstractions

- Different kinds of objects often have a certain amount in common with each other.

Level of abstractions

- Different kinds of objects often have a certain amount in common with each other.
  - Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear).
  - Yet each also defines additional features that make them different:
    - tandem bicycles have two seats and two sets of handlebars;
    - road bikes have drop handlebars;
    - some mountain bikes have an additional chain ring, giving them a lower gear ratio.
- All this different details are treated at *different level of abstractions*
  - At some point in the program, we only need to know that an object is a bike, but it is not important which kind of bike
  - In other points, we deliberately use the type information (Road, Tandem or Mountain) to call specific operations on each type

Objects in the problem space

- Low level language are closer to the machine
  - their basic elements are closely related to machine instructions
  - the programmer has the difficult task to represent (model) the problem with low level constructs (large abstraction effort)
- High level languages (Prolog, Lisp, etc.) are close to the problem space
  - Their basic constructs allow to reason directly in terms of problem elements
  - However, these language are often too specific for a set of problems (e.g. Prolog), and inflexible
- Object Oriented languages provide tools for the programmer to represent elements in the problem space
  - This representation is general enough that the programmer is not constrained to any particular type of problem

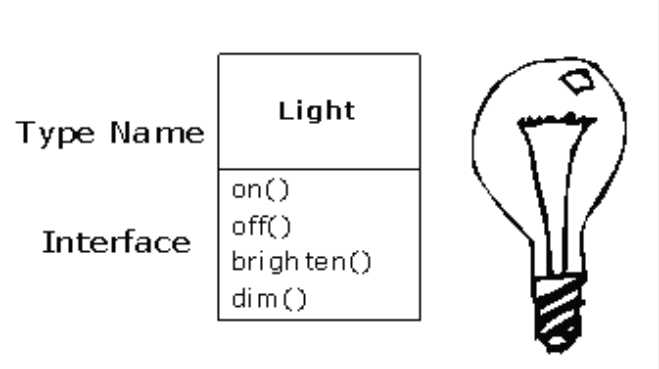
Objects

- OOP allows you to describe the problem in terms of the problem, via decomposition process
- we refer to the elements in the problem space and their representations in the solution space as "objects"
- the program is allowed to adapt itself to the *lingo* of the problem by adding new types of objects, so when you read the code describing the solution, you're reading words that also express the problem
- "An object has state, behavior and identity" (Booch)
  - an object can have internal data (which gives it state),
  - methods (to produce behavior), and
  - each object can be uniquely distinguished from every other object – to put this in a concrete sense, each object has a unique address in memory

Object interface

- The idea that all objects, while being unique, are also part of a class of objects that have characteristics and behaviors in common was used directly in the first object-oriented language, Simula-67, with its fundamental keyword class that introduces a new type into a program
- although what we really do in object-oriented programming is create new data types, virtually all object-oriented programming languages use the "class" keyword. When you see the word "type" think "class" and vice versa
- one of the challenges of object-oriented programming is to create a one-to-one mapping between the elements in the problem space and objects in the solution space

Example: light



- the name of the class is `Light`, the name of the object is `lt`, and the interface permits to turn it on/off, make it brighter or make it dimmer.

An object provides services

- You should think of an object as a service provider
- the goal of the programmer is to produce (or find in existing libraries) a set of objects that provide the right services that you need to solve the problem
  - To do this, you need to decompose the problem space into a set of objects
  - it does not matter if you do not know yet how to implement them
  - what it is important is to a) identify what are the "important" objects that are present in your problem and b) identify which services these objects can provide
  - Then, for every object, you should think if it is possible to decompose it into a set of simpler objects
  - You should stop when you find that the objects are *small enough* that can be easily implements and are self-consistent and self-contained

High cohesion

- Thinking of an object as a service provider has an additional benefit: it helps to improve the cohesiveness of the object.
- High cohesion is a fundamental quality of software design
  - It means that the various aspects of a software component (such as an object, although this could also apply to a method or a library of objects) "fit together" well
- One problem people have when designing objects is cramming too much functionality into one object
  - Treating objects as service providers is a great simplifying tool, and it's very useful not only during the design process, but also when someone else is trying to understand your code or reuse an object
  - if they can see the value of the object based on what service it provides, it makes it much easier to fit it into the design.

Hiding the implementation

- Even when you are writing a program all by yourself, it is useful to break the playing field into class creators and client programmers
  - the class creators implements the internals of a class, so that it can provide services
  - the client programmers use the class to realize some other behavior (e.g. another class)
  - almost all programmers are at the same time class creators and client programmers
- The class creators must not expose the implementation details to the client programmers
  - The goal is to export only the details that are strictly useful to provide the services

Hiding the implementation

- The concept of implementation hiding cannot be overemphasized
- Why it is so important?
  - The first reason for access control is to keep client programmers' hands off portions they shouldn't touch
  - This is actually a service to users because they can easily see what's important to them and what they can ignore
  - The second reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer
  - For example, you might implement a particular class in a simple fashion to ease development, and then later discover that you need to rewrite it in order to make it run faster
  - If the interface and implementation are clearly separated and protected, you can accomplish this easily

Java support for implementation hiding

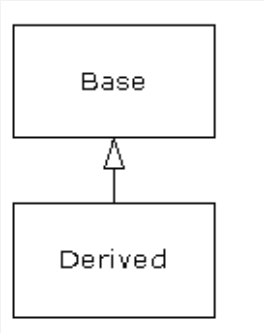
- Java uses three keywords for access control
  - **public** means the following element is available to everyone
  - **private** means that no one can access that element except you, the creator of the type, inside methods of that type
  - **protected** is similar to private, except that derived class (that will see later) can access it

Reusing implementation

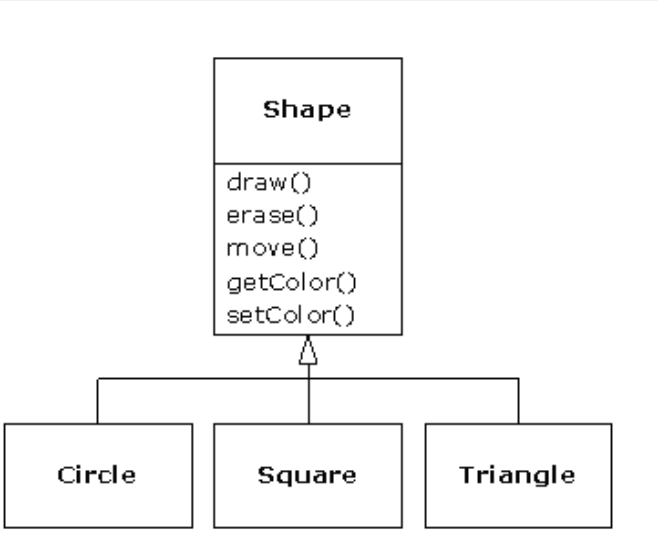
- Once a class has been created and tested, it should (ideally) represent a useful unit of code, so a candidate for usage in different programs
- The simplest way to reuse a class is to just use an object of that class directly
  - you can also place an object of that class inside a new class
  - This is called composition
- Composition is often referred to as a "has-a" relationship, as in "a car has an engine"
  - The member objects of your new class are typically private, making them inaccessible to the client programmers who are using the class.
  - This allows you to change those members without disturbing existing client code.
  - You can also change the member objects at run time, to dynamically change the behavior of your program (in this case, it is often called aggregation)

Reusing the interface

- Two types can have characteristics and behaviors in common, but one type may contain more characteristics than another and may also handle more messages (or handle them differently)
- Inheritance expresses this similarity between types by using the concept of base types and derived types

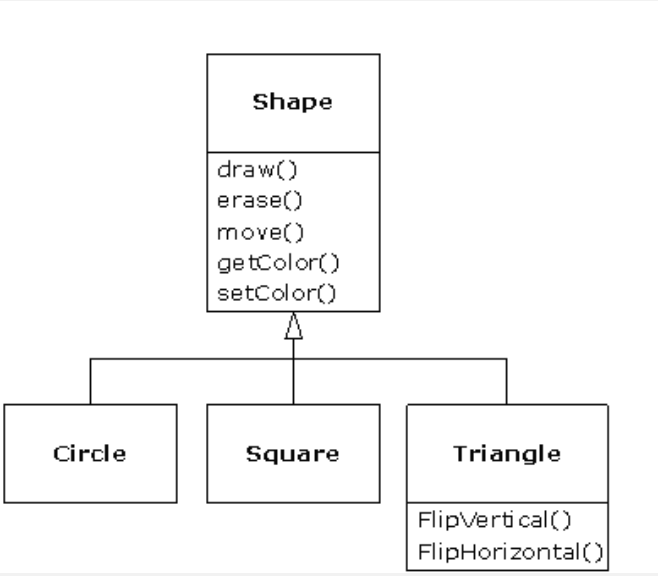


Shape example



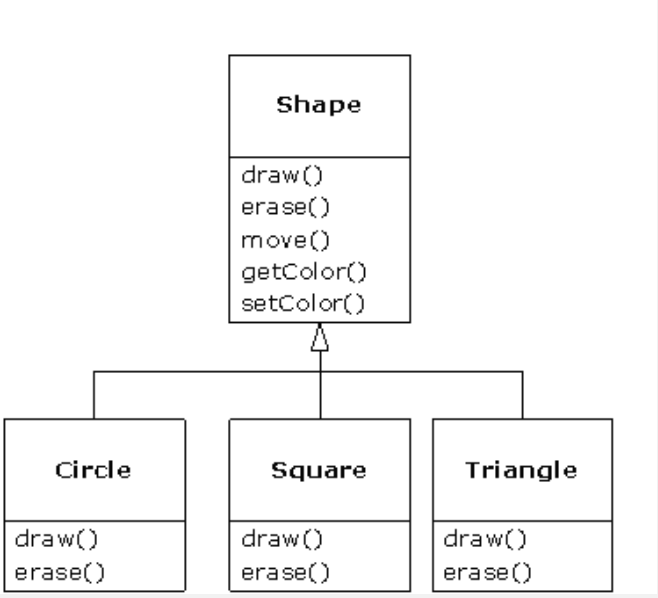
- all derived objects can be treated as the base object (they are all shapes)
- all the messages you can send to objects of the base class you can also send to objects of the derived class
- of course, they all behave slightly differently

Inheritance



- You have two ways to differentiate your new derived class from the original base class
- **1:** add new methods to the derived class
  - These new methods are not part of the base class interface

Method overriding



- **2:** to change the behavior of an existing base-class method. This is referred to as method overriding
- You reuse the same interface, but with a different implementation

Polymorphism

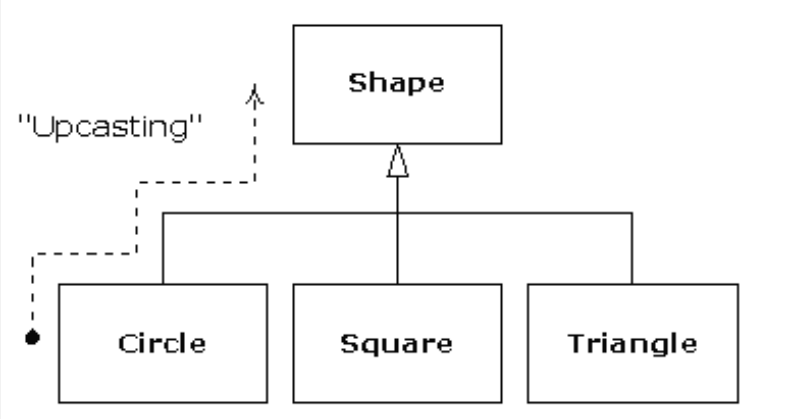
- When dealing with type hierarchies, you often want to treat an object not as the specific type that it is, but instead as its base type
- This allows you to write code that doesn't depend on specific types
- Example:
  - In the shape example, you may have a collection (list) of shapes, and you want to draw them all
  - However, every different type of object is drawn in a different way
  - You do not want to deal with this difference: you just want to tell the object "Please, draw yourself", and the object will take care of how it is drawn, depending on its type
  - In this way, if you add a new type of shape (e.g. a pentagon) you do not need to change the general code for drawing everything

Java example

```
void doStuff(Shape s) {
  s.erase();
  // ...
  s.draw();
}
...
Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
doStuff(c);
doStuff(t);
doStuff(l);
```

- What is going on here?
- (*discussion*)

Upcasting



- We call this process of treating a derived type as though it were its base type upcasting
- when the Java compiler is compiling the code for `doStuff()`, it cannot know exactly what types it is dealing with
- This means that the correct call is only linked at run-time (late binding)