Class Interface Design

So far we've focused mainly on the internal details of a class, i.e., **class implementation**.

We noted there that it is the **interface** to the class that is most important to the rest of the system.

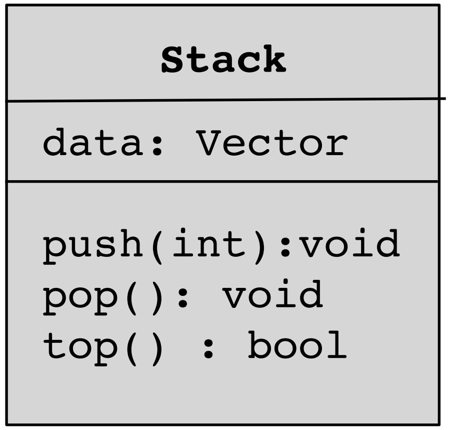
In this section we look at the issue of how to design the interface to a class.

Two Views of a Class

From what was stated previously about class interface and implementation, we see that there are two distinct views of a class:

The **client** of the class uses the class to help them build their own application.

The **designer** of the class designs both the interface of the class and implements its internal data structure and operations.



Client uses the class interface only

Designer implements the entire class

Client and Designer

Both client and designer are, of course, programmers.

In *programming-in-the-small*, the class designer and the class client are often the same person.

In *programming-in-the-large*, they are more commonly different people, and this is the more important case to consider.

Client vs. Designer

What is the class client concerned with and what is the class designer concerned with?

Client	Designer

The Client View

The client is concerned with issues like:

> the class should be easy to understand;
>
> it should help them solve the problem they're working on;
>
> it should provide an efficient implementation of its operations.

The Designer View

The designer on the other hand is concerned with issues like:

- providing an easy-to-understand interface for all clients;
- choosing a suitable data structure for the class;
- providing an efficient implementation of the operations of the class.

Try to design your classes as if you were expecting someone else to use them. Never expect that clients will look the private data/operations of the classes you design.

Use Consistent Method Names

Use a consistent set of names for the methods of a class.
An example to avoid:

```
class List
   def add_element(element)
      ...
   end

   def remove(element)
      ...
   end
end
```

Seems trivial, but this is very irritating for any clients of the
the class.

Make classes cohesive

A class should describe a single abstraction. Such a class is called *cohesive*. Real-world classes are naturally cohesive… that's **why** we recognise them as classes.

In software, it is all too easy to design classes in such a way that they lack this vital property.

A class whose interface lacks cohesion is difficult to use. It is hard for a client to build a memorable mental model of such a class.

What cohesion isn’t

If a operation of a class sticks out like a "sore thumb," the class isn't cohesive. For example:

```
class Mailbox
   def addMessage(message)
      ...
   end
   def removeMessage(msg_no)
      ...
   end
   def getCommand
      ...
   end
   def printMessage(msg_no)
      ...
   end
end
```

The **getCommand** operation clearly belongs somewhere else.

Lack of cohesion

If a class can be partitioned into smaller classes that don't interact, it's not cohesive. An unlikely (why not use a likely one?!) example:

```
class XXX
   def initialize(radius)
      ...
   end
   def pop
      ...
   end
   def push(num)
      ...
   end
   def move(dx, dy)
      ...
   end
   def radius
      ...
   end
   def top
      ...
   end
end
```

It may seem bizarre that a designer could merge the two abstractions Circle and Stack. However when you don't know a domain well, this is an easy mistake to make.

Primitive Operations

What problem would you have using this `Stack` class?

```
class Stack
   def initialize
      ...
   end

   # remove and return top element
   def pop
      ...
   end

   # push element on to top of stack
   def push(element)
      ...
   end
end
```

Decomposing into Primitive Operations

The pop method should be decomposed into its constituent parts:

```
# remove top element
def pop
   ...
end

# return top element
def top
   ...
end
```

This set of primitive operations allows the client to do what they want with the stack.

Attribute Accessors

In Ruby, providing read/write access to instance variables is a cinch:

```
class Foo {
   attr_accessor :x, :y
   ...
}
```

In other languages, it takes a bit more work.

However, providing read/write access to an instance variable is the most primitive operation you can provide:
- don't do it automatically
- only provide this level of access if it is really required

Completeness

Formally put, the data structure of a class coupled with the invariants over that structure define the legal set of values that objects of that class can assume.

A class with a **complete** interface allows an object of that class to reach any of its legal states.

If an interface is incomplete, certain legal states will not be reachable. Example on next slide.

An Incomplete Class Interface

Informally, the class must provide "enough" operations to support the desired functionality. Here's a contrived example:

```
class Person
   def initialize ...
      ...
      @age = 0    // invariant: 0 <= @age <= 150
   end
   ...
   def age_plus_2
      @age += 2
   end
end
```

The interface to this class is not **complete** -- `@age` can never be set to an odd value.

Convenience

A class may provide primitive operations and be complete but not be very convenient for clients to use, e.g., this version of the Date class:

```
class Date
   attr_accessor :day, :month, :year

   def initialize day, month, year
      @day = day
      @month = month
      @year = year
   end
end
```

A client can do anything they want with this class, but even common operations like comparing, printing etc. are not supported.

Clutter or Convenience?

An interface design **lacks convenience** when commonly-occurring operations, which are themselves compositions of primitive functions, are not provided in the interface to the class.

On the other hand, avoid adding too many "convenience" operations. A "convenience" operation that a client doesn't need is noise, cluttering up the vital public interface of the class.

There’s no hard-and-fast rule here – it’s a matter of judgement.

Interfacing Guidelines (1)

Narrow Interfaces
As a general rule, the fewer operations a class has the better. If a class you develop has a lot of public operations, assess its cohesiveness and consider splitting it up if necessary.

Keep reuse in mind
Consider how another programmer might want to use the class you're developing. Remember that reuse is facilitated by **design simplicity**, not by a complicated interface.

Interfacing Guidelines (2)

Minimise interfaces to other classes
Again, it aids reuse if a class is aware of as few other classes as possible.

A class that writes to the screen is almost impossible to reuse. A class that doesn't do screen I/O at all can be more easily reused in e.g. a web or mobile application.

It's all about trade-offs!
Design involves the resolution of conflicting forces. There's seldom one exactly correct solution. Don't vacillate for too long on any issue.

Summary

In this section of the course we looked at issues involved in developing the **interface** to a class.

This topic is vital when you are developing new classes.

In this module, when you are asked to write a new class, it is usually quite clear what the interface should be.