7COM1025
Programming for Software Engineers
Lecture 4

Dr. Renato C. de Amorim

QUALITIES – MODEL THE PROBLEM DOMAIN

An API should be easy for users to use and understand. It must correlate closely with their pre-existing knowledge and experience.

Provide a good abstraction
- It should be formulated in terms of high level concepts (instead of exposing low-level implementation issues).
- A non-technical reader should be able to see that the groups of operations provided by the API make sense and belong together as a unit.

eg. API of a simple address program.
- It should be a container for the addresses of multiple people
- It is logical that the API should provide an AddressBook object containing a collection of Person objects.
- Adding people and removing them change the state of AddressBook, so they are logically part of it

QUALITIES – MODEL THE PROBLEM DOMAIN

MODEL THE KEY OBJECTS
You should model the key objects (Object-Oriented Design).

Lets say you got these new requirements:
1) Each person may have multiple addresses.
2) Each person may have multiple telephone numbers.
3) Telephone numbers can be validated and formatted.
4) An Address book may contain multiple people with the same name.
5) An existing address book entry can be modified.

Some people would solve (1) By adding fields to the Person object like HomeAddress1, WorkAddress2, etc. (this is not an elegant solution)
Instead you could introduce the object Address and allow a Person object to have multiple of these.

The same applies to (2). Also, Telephone should have methods such as .IsValid() and .GetFormaTTedNumber(). These methods should naturally belong to Telephone, not Person

Clearly you won't be able to uniquely identify a Person using the name, so you should introduce an ID.

QUALITIES – HIDE IMPLEMENTATION DETAILS

It allows one to change the implementation details without affecting any existing client.

There are two ways to hide implementation details.

Physical Hiding: The private source code is simply not available. Declarations and definitions are stored in two different files (.h and .cpp)

Logical Hiding: Uses language features to limit access (encapsulation). Methods can be defined as:
- Public: Accessible to everyone.
- Protected: Accessible to members of this class and derived classes.
- private: Accessible to members of this class only.

Hide implementation methods. You should hide all methods that do not *need* to be public.

Hide Implementation classes. You should hide any class that is purely implementation detail

QUALITIES – HIDE IMPLEMENTATION DETAILS

Hide member variables. Instead provide getter and setter methods.
This allows you to:
- Validate values.
- Lazy evaluation. If calculating the value of a variable has a cost, you can defer this cost until the value is actually needed.
- Caching. Calculate the value once and then save it.
- Extra computation. If necessary you can perform additional operations.
- Notifications. Other modules may wish to know when a variable has changed in your class.
- Debugging. You may want to add debugging or logging statements to track changes in variables.
- Synchronisation: To make it thread safe by using mutually exclusive (“mutex”) locking.
- Finer access control. You can even make certain variables read-only (or write-only).
- Maintaining invariant relationships. Allowing values to depend on each other.

QUALITIES – MINIMALLY COMPLETE

A good API should be as small as possible (but no smaller!).

It should be complete. Provide the clients with all needed functionality.

Don't over-promise Every public element of your API is a promise, a promise that you will support that functionality for the lifetime of the API.

You may break this promise, but you may frustrate your clients.

Adding new functionality is (relatively) easy, but removing is difficult.

You should try to keep your API as simple as you can: minimise the number of classes you expose, and the number of public members in those classes.

Add virtual functions judiciously

- A virtual function is a method that is declared in the superclass but defined in the subclass.
- Be careful not to expose more than you would like through inheritance.

QUALITIES – EASY TO USE

A well-designed API should make simple tasks easy and obvious. A client should be able to understand how to use a method just by looking at its signature.

- But this is not an excuse not to provide documentation!

It should be discoverable. A user should be able to work out how to use the API on his own.

Discoverability does not necessarily leads to easy of use. For example, and API may be easy for 1st time users to learn, but cumbersome for an expert user to use on a regular basis.

Difficult to misuse. Some of the most common ways to misuse an API include:

- Passing the wrong arguments.
- Passing illegal values.

The above can happen when you have multiple arguments of the same type and the user forgets the correct order of the arguments.

Consistency. A good API should apply a consistent design approach. This way its conventions will be easy to remember (and therefore easy to adopt!). This applies to:

- Naming conventions, parameter order, the use of standard patterns, memory model semantics, use of exceptions, error handling, etc.

QUALITIES – EASY TO USE

Orthogonality. Methods should be independent and not have side effects.

- Calling a method that sets a particular property should not change other publicly accessible properties.
- As a result: making a change to the implementation of one part of the API should have no effect on other parts of the API.

Orthogonality produces APIs with behaviours that are more predictable and comprehensible.

1) Reduce redundancy. There should be a single source for each piece of knowledge.

2) Increase independence. Ensure there is no overlapping of meaning in the exposed concepts

<u>Robust resource allocation</u>

Frequently bugs arise from misuse of pointers or references:

- Null dereferencing: Trying to use → or * operators on a NULL pointer.
- Double freeing: releasing the same memory twice.
- Accessing invalid memory: trying to use → or * on a pointer that has not been allocated yet or that has been freed.
- Memory leaks: not freeing memory.

You can avoid some of the issues above by using shared pointers, weak pointers and scoped pointers.

QUALITIES – EASY TO USE

Platform independence

- A well-designed API should always avoid platform specific lines.
- There are only a few cases where the API should be different for different platforms.

Loosely coupled

-Coupling: A measure of the degree to which each component depends on other components in the system.

  - Given two components A and B, how much code in B must be changed if A changes.

- Cohesion: A measure of how strongly related the various functions of a *single* software component are.

You should aim for loose coupling and high cohesion

The design of your API may affect the degree of coupling of your client applications

QUALITIES – EASY TO USE

Coupling by name only

If class A only needs to know the name of class B (it doesn't need to know its size and it doesn't call any methods in B)

- Then class A does not need to depend on the full declaration of B.

In these cases all you need is to include a forward declaration (rather than including the whole interface).

```
class MyObject;

class MyObJectHolder
{
Public:
	MyObjectHolder();
	Void SetObject(MyObject *obj);
	MyObject *GetObject() const;
private:
	MyObject *mObj;
}
```

QUALITIES – EASY TO USE

Intentional Redundancy

Normally good software engineering aims to remove redundancy. However, reuse of code implies coupling.

Sometimes it may be worth to add a small degree of duplication.

Manager class

A manager class owns and coordinates several lower-level classes. This can be used to break the dependency of one or more classes upon a collection of low-level classes.

- A database manager class is a good example.

QUALITIES – STABLE, DOCUMENTED, AND TESTED

Any API should be Stable
- It doesn't mean the API never changes.
- It does mean the API should be Versioned and should not change incompatibly from one version to the next.

An API should be designed to be extensible.

An API should be well-documented so that users have clear information about:
- Capabilities
- Behaviour
- Best practices
- Error conditions

EXERCISE

In small groups:
Choose one of the qualities (or subqualities) discussed today and write a C++ code example that does NOT follow it. Afterwards re-write the same piece of code following the recommendations.