7COM1025
Programming for Software Engineers
Lecture 16

Dr. Renato C. de Amorim

Architecture design

Describes the structure of the entire system. The collection of top-level objects in the API and their relationship to each other.

At a high level, the process of creating an architecture for an API resolves to four basic steps
a) Analyse the functional requirements that affect the architecture.
b) Identify and account for the constraints on the architecture.
c) Invent the primary objects in the system and their relationships.
d) Communicate and document the architecture.

The first step is feed by the earlier requirement gathering stage.

Architecture constraints.
Organisational factors. Eg. Budget schedule, team size and expertise, etc.
Environmental factors. Eg. Hardware, platform, software constraints (use of other APIs), File format constraints, database dependencies, development tools, etc.
Operational factors. Eg. Performance, memory utilisation, Reliability, Extensibility, etc.

Always design for change. Change is inevitable.

Architecture design

Once you have analysed the requirements and constraints of the system, you can start building a high-level object model.

There are many techniques to classify the major abstractions in a system.

Example: Natural language.
In general nouns tend to represent objects, verbs functions, and adjectives (and possessive nouns) represent attributes.

Going back to the address book example. Address book and person are both nouns (objects). Adding a person (function). A person has a name (possessive noun so its an attribute).

Architecture design

Architectural patterns describe solutions to common problems at architectural level.

Common examples include:

- Structural patterns: Layers (partitions the concerns of the application into stacked layers, eg. OSI network model),
- Interactive systems: Model – View – Controller (MVC)
- Distributed Systems: Client/Server, Peer to Peer, etc.
- Adaptable systems: Micro-kernel (separates minimal functional core from extended functionality – helps to adapt systems to changing requirements).

Class design

With a high-level architecture in place, you can start refining the design to describe specific C++ classes and their relationship to other classes.

It involves identifying actual classes, how these classes relate to each other, and their major functions and attributes.

When considering the creating of a class, there are many factors to be considered, including:
Use of inheritance: is it appropriate to add the class to an existing hierarchy?
Use of composition: is it more appropriate to hold a related object as a data member rather than inherit from it directly?
Use of abstract interfaces: is the class meant to be an abstract base class, with subclasses overriding various pure virtual member functions?
Use of standard design patterns:Can you employ a known design pattern to the class design?
Initialisation and destruction model: Will the clients use new and delete or will you use a factory method?

Class design

The biggest decision when designing your classes is determining when and how to use inheritance.

Recommendations:

Design for inheritance or prohibit it. If you allow it think about what methods should be declared as virtual, and document their behaviour. If not, declare a non-virtual destructor.

Avoid deep inheritance trees. Such hierarchies increase complexity. The limit is subjective but any more than three levels is already getting too complex.

Use pure virtual member functions to force subclasses to provide an implementation. Pure virtual functions must be defined in subclasses.

Don't add new pure virtual functions to an existing interface. If you so you will break the code of your clients who already use the your API.

Class design

Give expressive and consistent names for your classes.

Simple class names should be powerful, descriptive and self-explanatory. Eg. Customer, Bookmark, Document, etc..

Good names drive good designs. The class name should instantly convey its purpose. If a class is difficult to name your design may need improvement.

Sometimes compound names convey greater specificity and precision. Eg. TextStyle, SelectionManager, LevelEditor. Anymore than 2 or 3 words tends to indicate confusing (or complex) design.

Function design

This is the lowest granularity of API design: how you represent individual function Calls.

Alternatives for free functions:
Static (“private” to its own file) vs non-static functions.
Pass arguments by value, reference or pointer.
Use of optional arguments with default values.
Return result by value, reference or pointer.
Return result as const or non-const.

For member functions, consider all the above plus:
Virtual vs non-virtual member functions.
Pure virtual vs non-pure virtual member function.
const versus non-const member function.
Public, protected or private member function.

Function design

Function names tend to form the verbs of your system, describing actions to be performed or values to be returned.

Recommendations:
- Use Set and Get when appropriate.
- Function that answer yes or no to queries should have the appropriate prefix: Is, Are or Has (and should return bool!). Eg: IsEnabled() IsEmpty().
- Functions that perform an action should be named with a strong verb: Print(), Save(). If its a free function add the name of the object it acts upon. Eg. FormatString(), MakeVector3D(), etc.
- Use positive concepts (not negative), eg. IsConnected() - not IsUnconnected()
- Avoid abbreviations.
- Shouldn't start with an underscore _. In C++ global symbols start with _ are reserved for internal compiler use.
- Function that form natural pairs should use the correct complementary terminology: Eg. OpenWindow() and CloseWindow() - not DismissWindow(). Precise opposite terms makes it clearer.
- Use meaningful variable names in the parameter list – not cryptic names.