Foreword

This document is an extension of more generic need/want–to-have keywords/key phrases into a more concrete architectural context. Some of the keywords/key phrases are:

- divide and conquer
- minimal dependencies
- re usability

Architectural overview

<u>Problem description</u>

The main architectural concern is the tight, often implicit, coupling in software systems. It is extremely hard to handle mutations of said software system.
The complexity of interactions in such software, given this tight coupling, increases at an alarming rate. This is because every possible new combination introduces new fault scenarios.

<u>Proposed solution</u>

The key architectural principle to solving such a problem is the application of a decoupling process. Decoupling should be the main theme both along the functional axis at the same functional level combined with a strict decoupling between the different functional levels themselves.

When attempting to apply a decoupling process to an entangled system you are bound to run into problems. When devising a way to tackle the problem I found the onion analogy to be a good representation.

The onion salesman

You are a poor onion salesman and you have an onion with a rotten core and several layers around it's core that are funky at best. The easiest way to salvage as much of the onion as possible is to start peeling off layers starting at the outer layer. You peel the layer off, clean it up and then stick it back on again. The onion looks better but it still has a rotten core. You continue peeling off layers and cleaning them up as you go. At a certain point you reach the core of the onion which is so rotten that the amount of cleaning that would be required is no longer worth attempting to salvage more of the onion. This is the turning point when a critical mass of new/cleaned infrastructure has formed that the time has come to get rid of the rotten remains of the original onion and place the onion peels you cleaned around a new core. Keep this poor onion salesman in mind when you read this document.

In order to arrive at the desired result at an implementation level it is important to follow a particular development process. Such a process must have safeguards to protect against context contamination and discourage sequential reasoning.

Context contamination

Context contamination is the process where awareness of the environment is included in the makeup of the context itself. Context contamination usually results from an insufficient “separation of concerns”.

The following is a simplified example:
Suppose you are back in college and you are asked to create the famous ‘BigInt’ class. This entails the creation of a class that can handle integer numbers of any size. Now suppose you want to display the value using OpenGL. Displaying the value as a number in your visualization might be something you have to do a lot so to some people it might make perfect sense to add this kind of capability to the ‘BigInt’ class. Next suppose that a colleague comes along and he/she wants to use your ‘BigInt’ class for their application. Their application however doesn’t have a clue of visualization capabilities let alone OpenGL. It might have nothing to do with visualization. This makes the ‘BigInt’ functionality unusable for your colleague.

In the given example the ‘BigInt’ functionality is contaminated with a higher-level context, one that deals with visualization. The application of the decoupling process should allow the colleague in the example to the use ‘BigInt’ functionality after the process is complete.

Sequential reasoning

Sequential reasoning is when you think of some task you are attempting to achieve and in doing so design all the different required components based on the sequential steps needed to achieve the task. This almost always leads to context contamination.

Decoupling within this context does not mean decoupling at a use-case level. Use-cases can determine the need-to-have functionality but they cannot be used to steer the decoupling effort. A use-case is almost by definition created using sequential reasoning which is a process that can be viewed as being orthogonal to the functional decomposition needed to realize decoupling in this architectural vision.

In order to avoid become a victim of this common pitfall it is very important to make an explicit switch in your development process from sequential reasoning to a purely functional decomposition. In my view this can best be achieved in the phase where use-cases have been converted into high-level requirements using a top-down view. After this step it is important to throw all those requirements on one big heap and forget about the uses-cases that where used to define those requirements. The details of the uses-cases become irrelevant at this point. You can then begin to create your functional requirements by examining the type of functionality that is affected using a bottom-up view.

This type of functional requirement is not allowed to have any application awareness. It simply defines a capability of an individual component at a particular functional level. These are the requirements the designs should be based on.

The decoupling process

The functional decomposition phase should map the desired functions on specific functional levels in the overall design. If a function has knowledge of items on multiple functional levels then it should be placed at the level of the highest level dependency. If so desired the function can itself be decomposed. It can delegate elements of its functionality to a lower functional level where if needed new functionality should be added.

As a rule of thumb when designing these systems keep the following in mind:

1. Each atomic block of logic is only allowed to have *one* core functionality. Note that delegation, coordination and manager roles are also considered to be core functionalities.
2. Apply 'Occam's razor' when making the functional decomposition and when creating the designs at each functional level. The principle states that the explanation of any phenomenon should make as few assumptions as possible, eliminating, or "shaving off," those that make no difference in the observable predictions of the explanatory hypothesis or theory. In short check the 'need to know' in a very critical fashion.

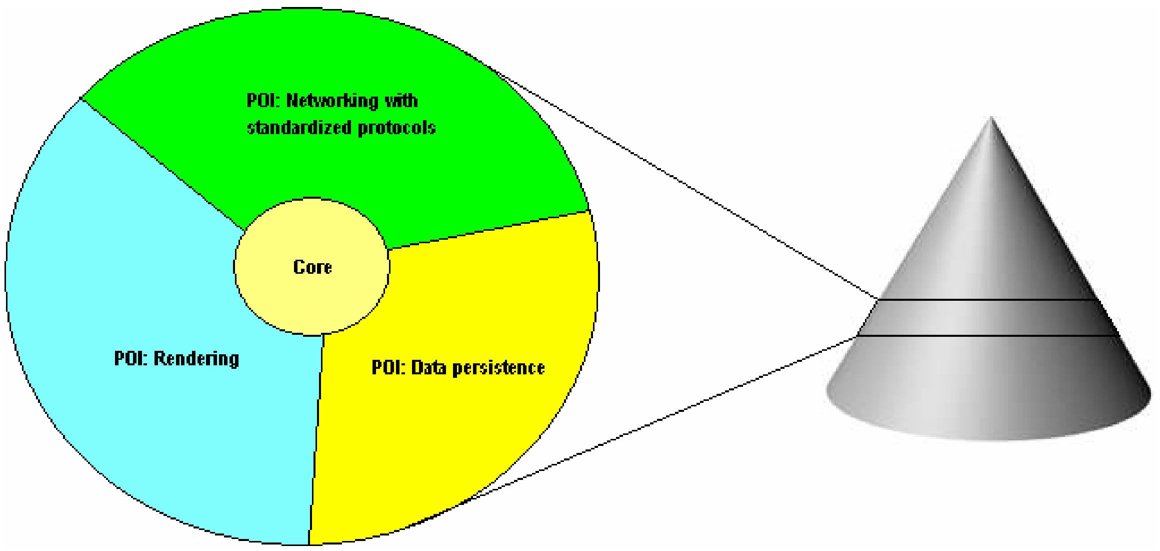


Fig. 1: Example schematic horizontal cross-section of an application infrastructure

You can think of each functional level seen from the application design standpoint as a slice/level in a Pyramid or Cone, whichever you prefer. At such a level you can differentiate between separate items as defined out of a purely functional decomposition.

Figure 1 offers a example of what you might see at the mid-tier of an application. At each level, except the most abstract, you can define Points Of Interest or POI’s for short. What the context and content of a POI is depends on the functional level at which it was defined and on whether or not it is even needed by the application. For example a server application does not need the rendering POI and a visualization application need not have networking.
At the center you have the core functionality which is very basic, just implementations of design patterns. They can be used on almost all levels because they are such abstract and pure building blocks without any context. Typically the decrease in surface area covered by the core is smaller then that of the POI’s. This can also be observed in Fig 2.

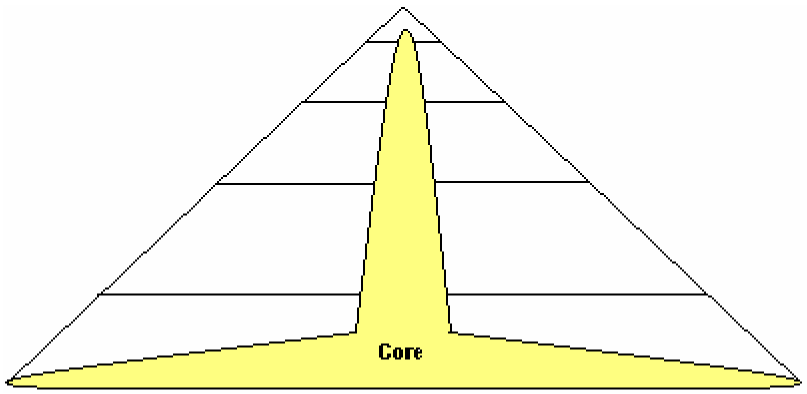


Fig. 2: Example schematic vertical cross-section of an application infrastructure

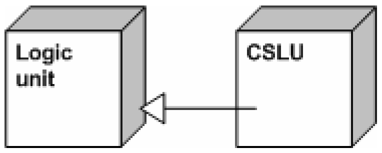
Note that application POI's can usually be mapped directly onto software infrastructure components. Such a component tends to be a separate binary library at the implementation level.

Design Terms and definitions

Logic unit:
A logic unit is a generic building block representing certain logic. The logic it represents should be independent of the context in which it is used. Each logic unit has a single core function.
An example of a logic unit is an abstract factory (see ‘Design Patterns’ by Gamma).

CSLU:
A CSLU (Context Sensitive Logic Unit) within this scope is a specialization of a logic unit that derives at least part of it’s reason to exist from the context in which it is placed. Aside form the context knowledge all rules that apply to a pure logic unit also apply to a CSLU.
An example of a CSLU is a mesh factory. A mesh has a certain meaning within the context of rendering. The fact that the factory knows it is creating a mesh turns the factory into a CSLU.

Relationship between logic units and CSLU’s:
The separation between logic units and it’s specialization CSLU is made to help purify and define toolkits of truly generic building blocks without contamination from context sensitive logic units.



Application layering

The architecture of an application can be defined as a number of layers each with it's own meaning and boundaries. Such a division is not always necessary but that doesn't mean it is not a good idea.
CSLU's are defined at different abstraction levels. Application layering is introduced to explicitly group CSLU's belonging to the same or similar level of abstraction.
Defining layers can help limit contamination within the hierachy that results from a decomposition at the lowest logical level using logic units and CSLU's.
Experience shows that contamination between CSLU's can occur over time despite the best intentions of the people involved. Application layering (among other things) allows you to introduce boundaries that help prevent the contamination from spreading to far. They act as a sort of firewall and quality portal.

Application sub-system

An application sub-system can be defined as a product capable of beeing integrated into a larger system to create an overall application system as seen by the end-user
Note that you can define multple levels within the context of application sub-systems as well. For example a platform like sub-system can serve as an application level building block for other sub-systems.
This is typically the layer that is most suited for outsourcing to a 3rd party supplier since a sub-system will typically offer a lot of functionality within it's context with relatively minor connections to the outside world. As such they are perfect candidates to be turned into plugins, allowing dynamic extension of your applications.
An example of a low level sub-system is for example a block called 'connectivity' in which an entire array of clients are defined for a wide variety of communication protocols. This might include a URL based system with support for a HTTP client, a FTP client a POP3 client etc.
An example of a high-level sub-system would be tool plugin that is seamlessly integrated into the application adding a set of functions to the application as a whole.

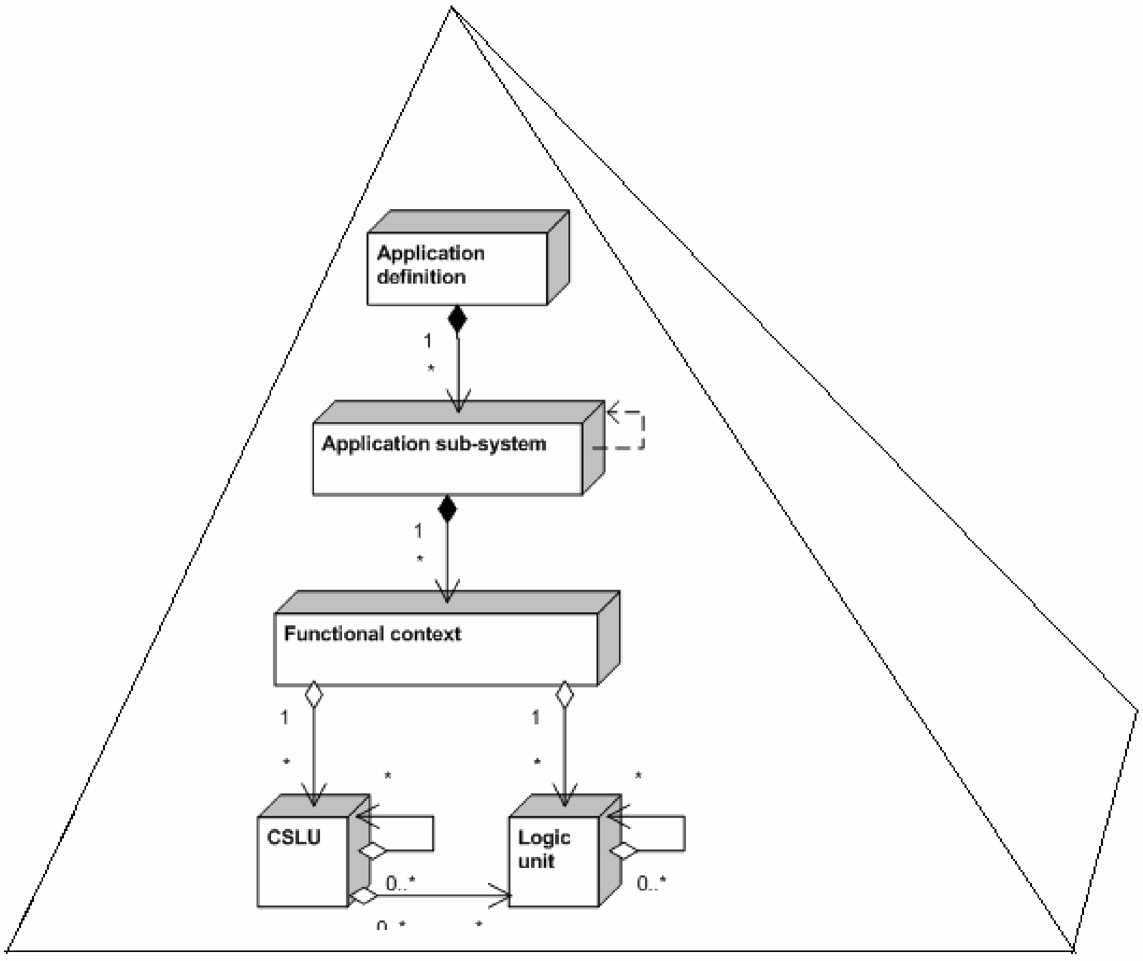
Functional context

A functional context is basically a *conceptual* sub-set of an application sub-system. It may only exist at the conceptual architectural level without a direct counterpart in a concrete design. Instead the functional context may be directly translated into a CSLU if you can avoid contaminating the CSLU with application specific context.
In a high-level application sub-system a functional conext is typically related to a set of use-cases at the design level. In low-level sub-systems a functional context is typically related to a set of CSLU's/Logic units that can be seen a family.
A more concrete example could be the addition of BitTorrent support in the 'connectivity' sub-system mentioned in the 'Application sub-system' segment. You might need the following CSLU's: BTClient, BTServer, BTPeer and a BTArbiter. This family of CSLU's combined can be referred to as a functional context which adds the BitTorrent functional context to the 'connectivity' sub-system.

Architectural requirements / rules

The following are a number of example requirements / rules and should by no means be considered complete or without errors at this point. It does however give you an example of how you can refine your architectural vision further.

3. Ability to add functionality to a functional context without disrupting said context
4. Ability to add functionality to an application as a separate functional context without disrupting the application.
5. Applications are defined as a set of sub-systems where the top-most application layer acts purely as a glue layer between the relevant sub-systems.
6. Sub-system awareness is limited to the highest level application logic, there is no direct awareness of specific sub-systems between sub-systems.
7. An application sub-system is defined as a common set of application level functions belonging to the same application level context.
8. Application level functionality should be designed in terms of logic units and CSLU's. No sequential function logic is allowed, meaning that the logic units cannot be constructed in a way that forces it's user into a specific usage pattern. This makes the logic units basicly part of a sequential chain of function logic, hence the term. If function specific logic is required then it should be limited to a glue layer between the different logical components. This limitations is a requirement for the reuse of logic units as building blocks.
9. Logic units should either consist out of an atomic logical context (basically a logic unit that is self-contained logic wise and cannot be divided further) or as the top of a pyramid/frustum comprised of logic units.
10. A logic unit that is part of a logic pyramid/frustum can only have dependencies on the same or lower level or the pyramid/frustum. Dependencies to a higher level are never allowed, such a dependency automatically places the logic unit at this higher level.
11. Pure logic units cannot use CSLU's, doing so turns the logic unit into a CSLU itself.



Application definition
1
*
Application sub-system
1
*
Functional context
1
*
1
*
*
*
CSLU
Logic unit
0..*
0..*
0..*
*