Software System Architecture and Design

The Functional Viewpoint
View Relationships

- **Context View** defines scope, context, and interfaces for **Software Design**.
- **Software Design** defines runtime environment for **Development View** and defines implementation constraints for **Operational View**.
- **Functional View** is emphasized within the diagram.

The Functional Viewpoint
The Functional view of a system defines the architectural elements that deliver the functions of the system being described.

This view documents the system’s functional structure including:
- the key functional elements,
- their responsibilities,
- the interfaces they expose, and
- the interactions between them.

Taken together, this demonstrates how the system will perform the functions required of it.

The Functional view is the cornerstone of most ADs and is often the first part of the description that stakeholders try to read.
- The Functional view usually drives the definition of many of the other architectural views.
Concerns

1. Functional Capabilities – *functional decomposition*
2. External Interfaces
3. Internal Structure – *structural decomposition*
4. Functional Design Philosophy:
   - Separation of concerns and High Cohesion
   - Loose Coupling
   - Low Interdependency
   - Extensibility and Functional flexibility
1. Concerns – Functional Capabilities

- Functional capabilities define what the system is *required to do* and, explicitly or implicitly, what it is not required to do (either because this functionality is outside the scope of consideration or because it is provided elsewhere).

- On some projects, you will have an agreed-upon set of requirements at the start of architecture definition, and you can focus in the Functional view on showing how your architectural elements work together to provide this functionality.
  - However, in many projects this isn’t the case, and the architect needs to ensure that there is a clear definition of what the system will (and won’t) be required to do.

- This concern cares about *functional decomposition*
  - Architect needs to look for *reference models* in the domain under consideration

---

**The Functional Viewpoint**

5
2. Concerns – External Interfaces

- External interfaces are the data, event, and control flows between your system and others.
  - Data can flow inward (usually resulting in an internal change of system state) and/or outward (usually as a result of internal changes of system state).
  - Events can be consumed by your system (notifying your system that something has occurred) or may be emitted by your system (acting as notifications for other systems).
  - A control flow may be inbound (a request by an external system to yours to perform a task) or outbound (a request by your system to another to perform a task).
- Interface definitions need to consider both the interface syntax (the structure of the data or request) and semantics (its meaning or effect).
3. Concerns – Internal Structure

- In most cases, you can design a system in a number of different ways to meet its requirements.
  - It can be built as a single monolithic entity or a collection of loosely coupled components; it can be constructed from a number of standard packages, linked together using commodity middleware, or written from scratch; etc.

- Your challenge is to choose among these many options in order to create an architecture that meets the functional requirements and exhibits the required quality properties.

- The internal structure of the system is defined by its internal elements, what they do (i.e., how they map onto the requirements), and how they interact with each other.
  - This internal organization can have a big impact on the system’s quality properties, such as its availability, ability to scale, and security
    - for example, a complex system that crosses organizational boundaries is generally harder to secure than a simple one running on a couple of collocated machines

- This concern cares about the structural decomposition
  - Architect needs to look for reference architectures in the domain under consideration
4. Concerns – Functional Design Philosophy

- Many of your stakeholders will be interested only in what the system does and the interfaces it presents to users and to other systems.
  - However, some stakeholders will be interested in how well the architecture adheres to established principles of sound design.
- Technical stakeholders, in particular the development and test teams, want a sound architecture, because a well-designed system is easier to build, test, operate, and enhance.
- Other stakeholders—particularly acquirers—implicitly want a well-designed system because it is faster, cheaper, and easier to get such a system into production.
Design characteristics

- The design philosophy will be underpinned by a number of design characteristics such as:
  - Cohesion
  - Coupling
  - Interdependency
  - Extensibility and Functional flexibility
  - Coherence, Consistency, Generality, Separation of concerns, Simplicity
Cohesion is a measure of how strongly-related or focused the responsibilities of a single class/component/module are.

As applied to object-oriented programming, if the methods that serve the given class tend to be similar in many aspects, then the class is said to have high cohesion.

Cohesion is decreased if:
- The functionalities embedded in a class, accessed through its methods, have little in common.
- Methods carry out many varied activities, often using coarsely-grained or unrelated sets of data.

Disadvantages of low cohesion (or “weak cohesion”) are:
- Increased difficulty in understanding modules.
- Increased difficulty in maintaining a system, because logical changes in the domain affect multiple modules, and because changes in one module require changes in related modules.
- Increased difficulty in reusing a module because most applications won’t need the random set of operations provided by a module.

In a highly-cohesive system, code readability and the likelihood of reuse is increased, while complexity is kept manageable.
Design characteristics – Coupling

http://en.wikipedia.org/wiki/Coupling_(computer_programming)

- Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong")
- Low coupling refers to a relationship in which one module interacts with another module through a simple and stable interface and does not need to be concerned with the other module's internal implementation
- Coupling increases between two classes $A$ and $B$ if:
  - $A$ has an attribute that refers to (is of type) $B$.
  - $A$ calls on services of an object $B$.
  - $A$ has a method that references $B$ (via return type or parameter).
  - $A$ is a subclass of (or implements) class $B$. 
Design characteristics – Coupling

(http://en.wikipedia.org/wiki/Coupling_(computer_programming))

- Disadvantages of high coupling:
  - A change in one module usually forces a ripple effect of changes in other modules.
  - Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
  - A particular module might be harder to reuse and/or test because dependent modules must be included.

- Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.
How to achieve loose coupling

Techniques to achieve low coupling are:

- Introduce **stable** interfaces between communicating components – ripple effect of changes stops at the interface boundary
- Code that creates class/component instances is moved away from classes/components
  - Design patterns – Abstract Factory, Builder
  - Dependency Injection (DI) techniques – EJB, CDI, SpringFramework
  - **Stable interfaces are still mandatory!**
- Remove the dependency between the two components
  - Event-driven techniques (e.g. CDI, SpringFramework)
Cohesion and Requirements

• Applying *Separation of Concerns* and *High Cohesion* principles to requirements specification we can distinguish these kinds of requirements¹:
  - **Localizable** functional requirements (implemented in SINGLE module)
    • Example: Internal bank money transfer
  - **Crosscutting functional** requirements (tend to be scattered throughout the whole code-base)
    - *Not-impacting* cross-cut functionality (Observer Pattern):
      • Example: if business transaction exceeds some money limit, inform Tax Inspection
    - *Impacting* cross-cut functionality (Decorator Pattern)
      • Example: if money are incoming from other bank, charge commissions
  - **Non-functional** requirements ([Wikipedia](https://en.wikipedia.org/wiki/Non-functional_requirements)) (Interceptor/Proxy Pattern)
    • They always are crosscutting¹
      • Examples: transactions, security checks, audit logging, etc.

¹ *Modularization of Crosscutting Concerns in Requirements Engineering*, 2008 ([PDF](https://example.com))
Cohesion and Requirements: Example
Crosscutting Requirements and DRY principle

- Loose Coupling is very important for implementation of crosscutting functional requirements.
- Modern technological platforms should provide means how to implement crosscutting functional requirements:
  - Without code duplication (DRY principle)
  - With loose coupling (or no coupling at all)
Consider Concurrency Patterns and Asynchronous Communication

- If use case implementation algorithm takes long time, consider splitting it to concurrently running parts
- Concurrency patterns help to choose best algorithm decomposition scheme
  - Producer-Consumer, Work pool, Map-Reduce, Fork-Join, etc.
- Often communication between parts will have to be asynchronous
Design characteristics – Interdependency

• What proportion of processing steps involves interactions between elements as opposed to within an element?

• Communicating between certain types of elements can be an order of magnitude more expensive (in terms of processing time and elapsed time), and significantly less reliable, than performing an operation within a functional element.
  • Especially if communication occurs via the network
Design characteristics – Extensibility and Functional flexibility

- Extensibility – Will the architecture be easy to extend to allow the system to perform new functions in the future?
- Functional flexibility – How amenable is the system to supporting changes to the functions already provided?
  - Systems that are designed to be easy to change are usually harder to build and typically don’t perform as well as systems that are less adaptable.
- Modifiability is independent quality property and has dedicated perspective
Functional Structure Model

- The functional structure model typically contains the following elements:
  - **Functional elements**: A functional element is a well-defined runtime (as opposed to design-time) part of the system that has particular responsibilities and exposes well-defined interfaces that allow it to be connected to other elements. At its simplest level, an element is a software code module, but in other contexts it could be an application package, a data store, or even a complete system.
  - **Interfaces**: An interface is defined by the inputs, outputs, and semantics of each operation offered and the nature of the interaction needed to invoke the operation.
  - **Connectors**: Connectors are the pieces of your architecture that link the elements together to allow them to interact.
  - **External entities**: External entities are other systems, software programs, hardware devices, or any other entity with which your system interacts. They are obtained from your system’s Context diagram, and each appears in the functional model at the far end of an interface, external to your system.
Functional Structure Model – Example using UML notation

- **Customer Care Interface**: 
  - Concurrent = '80'
  - Manage Customer

- **Customer Information System**: 
  - QueryCustomer

- **Web Shop**: 
  - OrderManagement
  - QueryCatalog
  - {type = 'HTML_UI', protocol = 'HTTP', concurrent = '1000'}
  - Customer Weblnterface

- **Order Processor**: 
  - OrderChanges

- **Product Catalog**: 
  - {type = 'RPC', protocol = 'LU-6.2'}
  - InventoryCheck
  - Manage Catalog

- **Catalog Management Interface**: 
  - Employee Weblnterface

- **Order Fulfillment**: 
  - {type = 'HTML_UI', protocol = 'HTTP', concurrent = '15'}

Order message propagated via PUR1 EAI message endpoint to order fulfillment
What is not obvious from this diagram:

- The responsibilities of the components aren’t clear,
- The details of their interfaces aren’t clear, and
- The details of how the components interact aren’t clear.

This impresses on us the need to complete the textual descriptions that underpin the diagram and the need to understand the system via a number of models rather than just one.
The use of a sketch is often required to effectively communicate essential aspects of the view to nontechnical stakeholders.

The problem with this approach is that it can lead to a poorly defined view and confusion among stakeholders.

You can get around this by using a sketch to augment a more formal view notation (such as UML) and using different notations for different stakeholder groups.
Analyze the Interactions

• Given the impact that excessive inter-element interactions can have, it is useful to analyze the chosen structure from the point of view of the number of inter-element interactions taken during common processing scenarios.

• Refining the functional structure to reduce inter-element interactions to a (feasible) minimum usually results in a well-structured system with cohesive, loosely coupled elements.
  • It is typically an important step toward an efficient and reliable system.

• You’ll have to make tradeoffs
Functional Structure Model – Common Pitfalls

- Infrastructure Modeled as Functional Elements
  - In general, you should not model underlying infrastructure as functional elements. The Deployment view defines the infrastructure in more detail.

- “God Elements”
  - Software designers often see object-oriented designs that have a single huge object in the center, with lots of small objects attached to it.
    - This situation is often dubbed the “God object” problem.
  - The underlying problem in such cases is usually an inappropriate partitioning of responsibilities among design elements—the large object (often called “Manager”) is really the *entire program*, and the small objects are often just *data structures* that this object uses.
Example of “God Element”
Homework

- Book "Software Systems Architecture"
  - Chapter 17 The Functional Viewpoint
- Article:
  - Modularization of Crosscutting Concerns in Requirements Engineering, 2008 (PDF)