# Algorithm Design and Data Structures – Introduction to building programming solutions

Algorithm design and data structures are an integral part of software development.

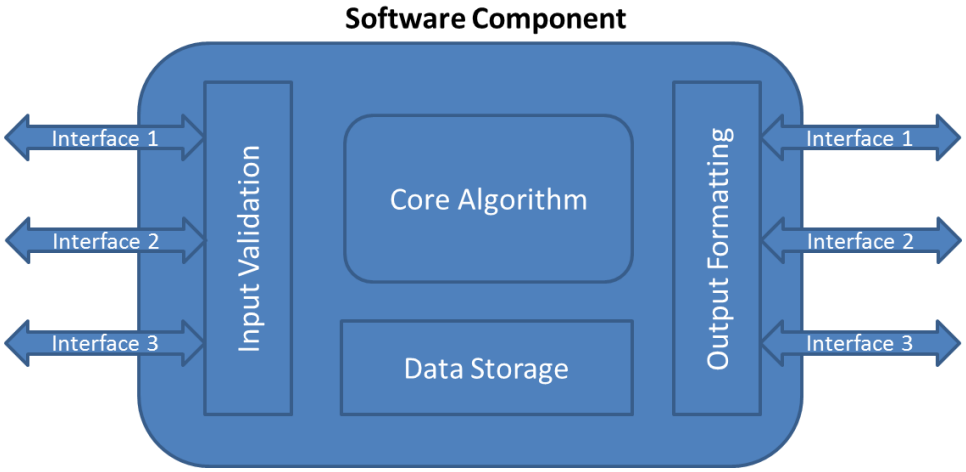
There is a plethora of material available on algorithm design, analysis and data structures. Lot of scholarly articles on optimization of the algorithms are available. We will explore these shortly.

In this note let us explore these in the context of design and development of components, modules and systems.

## Design & Development of a Component

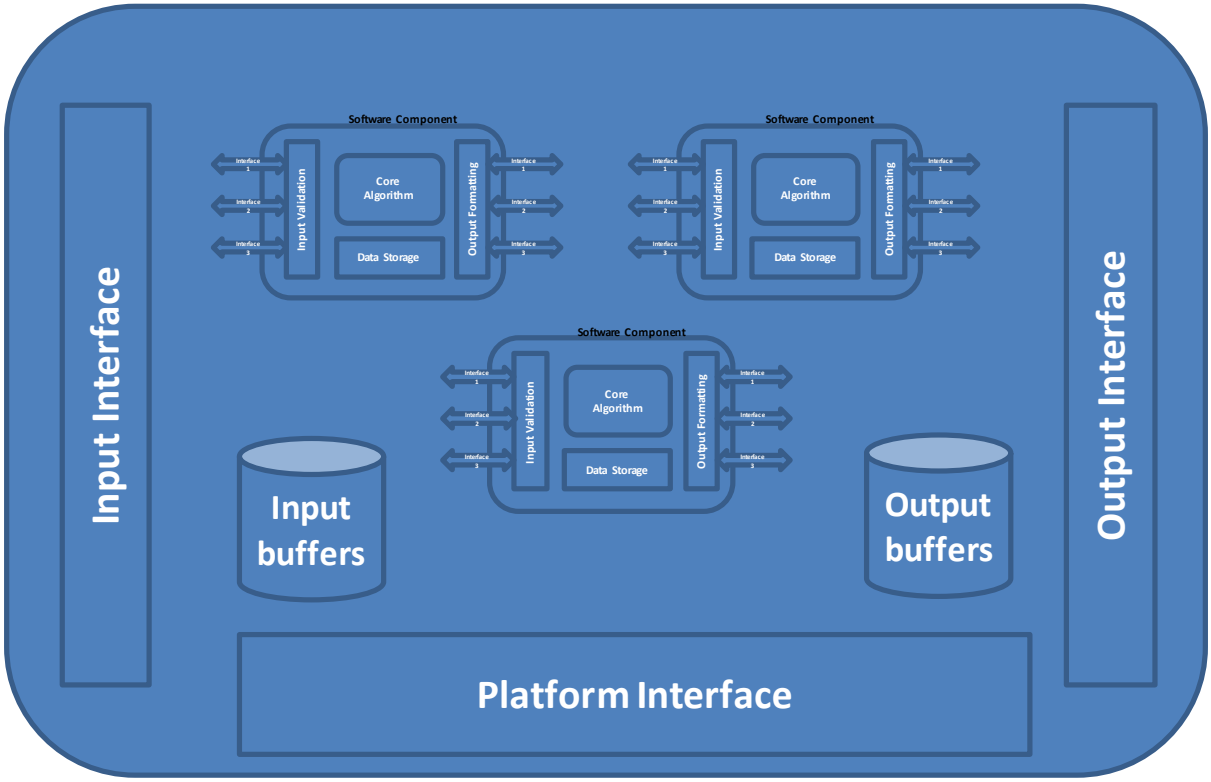
To start with let us consider design and development of a component.

1. **Core Algorithm:** What is the basic problem that this component is addressing? This would be the core algorithm that would need to be designed
2. **Inputs & Interfaces:** For any computing problem or process would need defined inputs and the interfaces through which it is available for the core algorithm
    a. Configurable parameters – data types, other requirements, i/o interfaces
    b. Data that would be used for computing
    c. Interfaces – keyed in inputs, files, shared memory, message pipes, queues etc
3. **Validation:** What are the input validations required? Remember – Garbage In-Garbage Out
4. **Data Storage:** How is the input data and output data stored? – data structures
5. **Outputs & Interfaces:** same as Inputs & Interfaces.
6. **Other requirements:** What are the non-functional requirements? Performance – load handling, latency, portability, security, reliability etc.



## Design & Development of a Module

Extrapolating the same for a module




## Typical goals in a module design

- Decompose system/module into components and interfaces
  - i.e., identify the software architecture
- Determine relationships between components
  - e.g., identify component dependencies and determine intercomponent communication mechanisms
- Specify component interfaces
  - Interfaces should be well-defined - Facilitates component testing and team communication
- Describe component functionality - informally or formally
- Identify opportunities for systematic reuse - Both top-down and bottom-up

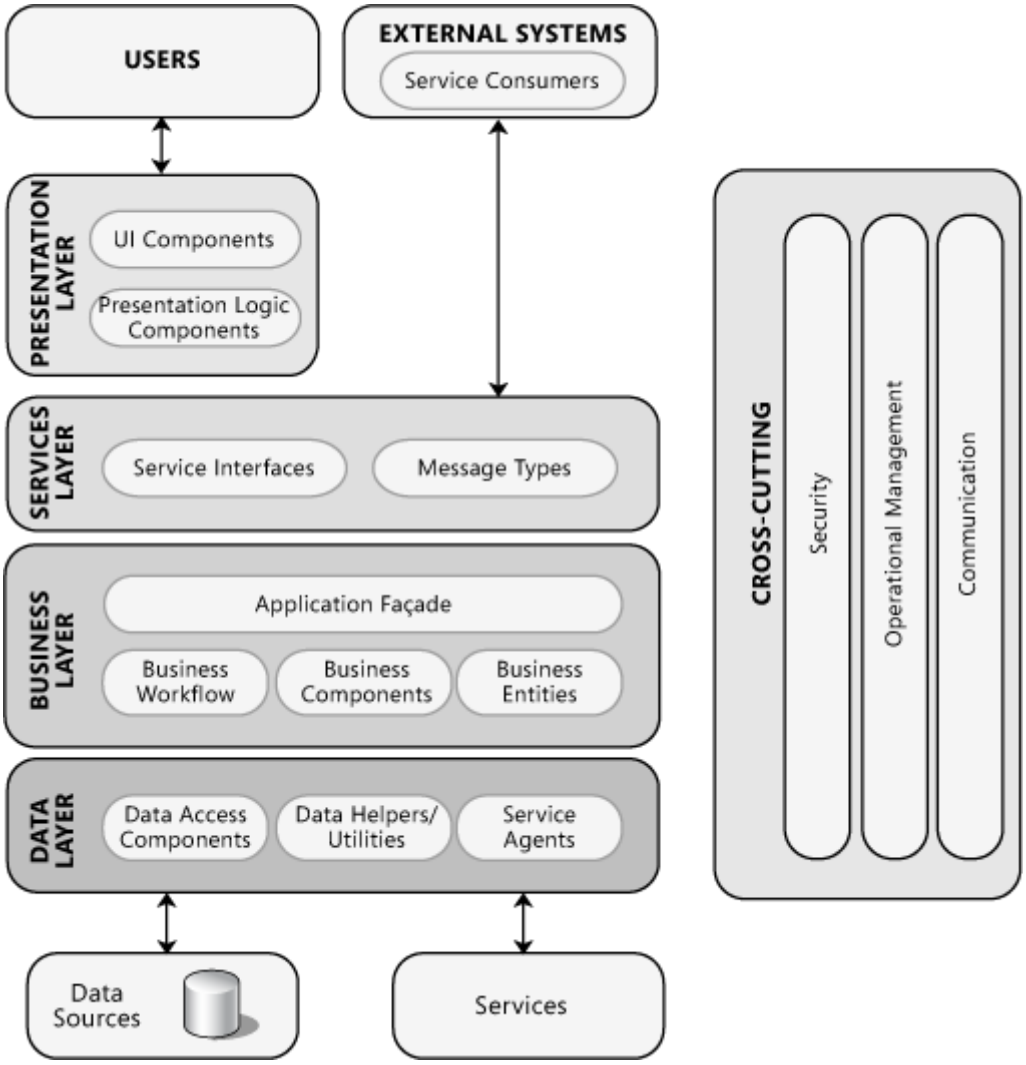
# Algorithm Design

## Principles of Design

- **Separation of concerns**. Divide your application into distinct features with as little overlap in functionality as possible. The important factor is minimization of interaction points to achieve high cohesion and low coupling. However, separating functionality at the wrong boundaries can result in high coupling and complexity between features even though the contained functionality within a feature does not significantly overlap.
- **Single Responsibility principle**. Each component or module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionality.
- **Principle of Least Knowledge** (also known as the Law of Demeter or LoD). A component or object should not know about internal details of other components or objects.
- **Don't repeat yourself (DRY)**. You should only need to specify intent in one place. For example, in terms of application design, specific functionality should be implemented in only one component; the functionality should not be duplicated in any other component.
- **Minimize upfront design.** Only design what is necessary. In some cases, you may require upfront comprehensive design and testing if the cost of development or a failure in the design is very high. In other cases, especially for agile development, you can avoid big design upfront (BDUF). If your application requirements are unclear, or if there is a possibility of the design evolving over time, avoid making a large design effort prematurely. This principle is sometimes known as YAGNI ("You ain't gonna need it").

# Algorithm Design

## System design - Layered Design



# Algorithm Design

| Category | Architectural Design | Description |
|---|---|---|
| Communication | Message bus | Prescribes use of a software system that can receive and send messages using one or more communication channels. |
| | Service–Oriented Architecture (SOA) | Defines the applications that expose and consume functionality as a service using contracts and messages. |
| Deployment | Client/server | Separate the system into two applications, where the client makes requests to the server. |
| | 3-tier or N-tier | Separates the functionality into separate segments with each segment being a tier located on a physically separate computer. |
| Domain | Domain Driven Design | Focused on modeling a business domain and defining business objects based on entities within the business domain. |
| Structure | Component Based | Breakdown the application design into reusable functional or logical components that expose well-defined communication interfaces. |
| | Layered | Divide the concerns of the application into stacked groups (layers). |
| | Object oriented | Based on the division of responsibilities of an application or system into objects, each containing the data and the behavior relevant to the object. |

# Algorithm Design

## Core Algorithm Design

Typical Stages in a Core Algorithm design would be

1. Problem definition
2. Development of a model
3. Specification of Algorithm
4. Designing an Algorithm
5. Checking the correctness of Algorithm
6. Analysis of Algorithm
7. Implementation of Algorithm
8. Program testing
9. Optimization based on test results.

The most common algorithm design paradigm or types are:

1. Simple recursive algorithms
2. Divide and conquer algorithms
3. Dynamic programming algorithms
4. Greedy algorithms
5. Backtracking algorithms
6. Branch and bound algorithms
7. Brute force algorithms
8. Randomized algorithms