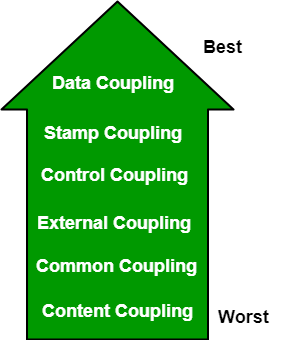
**Design Concepts:**

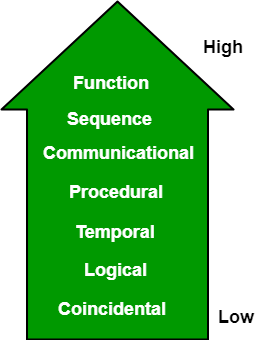
**Coupling:** Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling. 



**Types of Coupling:**

* **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Module communications don’t contain tramp data. Example-customer billing system.
* **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice was made by the insightful designer, not a lazy programmer.
* **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
* **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
* **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability.
* **Content Coupling:** In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

**Cohesion:** Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.



**Types of Cohesion:**

* **Functional Cohesion:** Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
* **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
* **Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.
* **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
* **Temporal Cohesion:** The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.
* **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
* **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

Design concepts:

1. **Architecture- design a structure of something**  
   Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.
2. **Refinement- removes impurities**  
   Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software. Refinement is very necessary to find out any error if present and then to reduce it.
3. **Pattern- a repeated form**   
   The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.
4. **Information Hiding- hide the information**   
   Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can’t be accessed by any other modules.
5. **Refactoring- reconstruct something**  
   Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without affecting the behavior or its functions. Fowler has defined refactoring as “the process of changing a software system in a way that it won’t affect the behavior of the design and improves the internal structure”.

**Refactoring** or **Code Refactoring** is defined as systematic process of improving existing computer code, without adding new functionality or changing external behaviour of the code. It is intended to change the implementation, definition, structure of code without changing functionality of software. It improves extensibility, maintainability, and readability of software without changing what it actually does.

**Why should we refactor our code when it works fine?**  
The goal of refactoring is not to add new functionality or remove an existing one. The main goal of refactoring is to make code easier to maintain in future and to fight technical debt. We do refactor because we understand that getting design right in first time is hard and also you get the following benefits from refactoring:

* Code size is often reduced
* Confusing code is restructured into simpler code

Both of the above benefits greatly improve maintainability which is required because requirements always keep changing.

**When do we refactor?**

* Before you add new features, make sure your design and current code is “good” this will help the new code be easier to write.
* When you need to fix a bug
* When you do a peer review
* During a code review

**How to identify code to refactor?**  
Martin Fowler proposed using “code smells” to identify when and where to refactor. Code smells are bad things done in code, just like bad patterns in the code. Refactoring and Code smells are a few techniques that help us identify problems in design and implementation. It also helps us in applying known solutions to these problems.

**Refactoring Techniques :**  
There are more than 70 refactoring techniques that exist. But we will discuss only a few, more common ones.

1. **Extract Method –**  
   When we have a code that can be grouped together.

**Example:**

def student():

getgrades()

# details

name = input()

class = input()

This could be refactored as:

def student():

getgrades()

getdetails()

def getdetails():

name = input()

class = input()

1. **Replace Temp with Query –**  
   When we are using a temporary variable to hold the result of an expression.

**Example :**

SI = P \* R \* T / 100

if(SI > 100):

return SI

else:

return SI \* 1.5

This could be refactored as:

def SI():

return P \* R \* T / 100

if(SI() > 100):

return SI()

else:

return SI()\*1.5