**Scale Agile Framework**

The Scaled Agile Framework (SAFe) has four levels of the organization

* Team Level
* Program Level
* Large Solution Level
* Portfolio Level

The Scaled Agile Framework (SAFe) is a framework for scaling Agile methodology to large enterprises. It is designed to help organizations address the challenges of implementing Agile in complex, multi-team environments. SAFe provides a structured approach to scaling Agile that incorporates Lean principles, systems thinking, and DevOps practices. Here are some key concepts of SAFe:

1. Agile Release Train (ART): An Agile Release Train is a long-lived team of Agile teams that is responsible for delivering a portion of the larger solution. It typically includes 5-12 teams and operates on a fixed cadence of planning, execution, and delivery.
2. PI Planning: PI Planning is a two-day event that occurs every 8-12 weeks in which teams plan the upcoming Program Increment (PI). During PI Planning, teams review the program vision and objectives, define features, and identify dependencies and risks.
3. Program Backlog: The Program Backlog is a prioritized list of features that are planned for the upcoming Program Increment. It is used to guide the work of the Agile Release Train and is reviewed and updated at each PI Planning event.
4. DevOps: DevOps is an approach to software development that emphasizes collaboration and communication between development and operations teams. In SAFe, DevOps practices are integrated into the framework to enable continuous delivery of value.
5. Lean-Agile Leadership: Lean-Agile Leadership is a key aspect of SAFe. It involves empowering teams to make decisions, creating a culture of continuous improvement, and providing leadership and guidance to support Agile teams.

SAFe provides a comprehensive framework for scaling Agile that can be customized to meet the needs of specific organizations. By incorporating Lean principles, systems thinking, and DevOps practices, SAFe enables large enterprises to deliver high-quality software faster and more efficiently than traditional approaches.

**SOLID Principle**

1. **Single Responsibility Principle**

“a class should have one, and only one, reason to change.” Following this principle means that each class only does one thing and every class or module only has responsibility for one part of the software’s functionality. More simply, each class should solve only one problem.

1. **Open close principle**

Following this principle is essential for writing code that is easy to maintain and revise. Your class complies with this principle if it is:

* Open for extension, meaning that the class’s behavior can be extended; and
* Closed for modification, meaning that the source code is set and cannot be changed.

1. **Liskov Substitution Principle**

Broadly, this principle simply requires that every derived class should be substitutable for its parent class.

සෑම sub-class object එකක්ම එහි parent class object එක වෙනුවට භාවිතා කල හැකි විය යුතුයි.

(every subclass/derived class should be substitutable for their base/parent class.)

1. **Interface Segregation Principle**

The general idea of interface segregation principle is that it’s better to have a lot of smaller interfaces than a few bigger ones. Martin explains this principle by advising, “Make fine grained interfaces that are client-specific. Clients should not be forced to implement interfaces they do not use.”

*ඔබගේ abstract class/interface එක භාවිතා කරන කෙනෙකු හට එහි ඇති functions implement කිරීමට බල කිරීමක් නොකල යුතුයි. එමෙන්ම ඔහු භාවිතා නොකරන methods මත functionalities රඳා පැවතීම සිදු නොවිය යුතුයි.*

1. **Dependency Inversion Principle**

Simply put, dependency inversion principle means that developers should “depend on abstractions, not on concretions.” Martin further explains this principle by asserting that, “high level modules should not depend upon low level modules. Both should depend on abstractions.” Further, “abstractions should not depend on details. Details should depend upon abstractions.”

Entities abstraction මත පමණක් යැපිය යුතු අතර කිසිම විටක concretions මත රඳා පැවතිය නොයුතුයි.

ඒ අනුව, low level modules මත high level modules රඳා නොපැවතිය යුතු වන අතර එය abstraction මත පමණක්ම රඳා පැවතිය යුතුයි

**Software Architecture**

Event driven architecture.

Client server architecture

Peer to Peer architecture

Space based Architecture

Component base architecture.

Service oriented architecture

Micro kernal architecture

Monolothic Architecture

Micro Service Architecture