Kenny R N Wilson

Software Architecture and Design

THIS DOCUMENT COVERS

- Introduction

Introduction

Architecture represents the significant design decisions that shape a system, where significant is measure by cost of change

– Grady Booch.

A good architecture comes from understanding it more as a journey than a destination, more as an ongoing process of enquiry than as a frozen artifact

– Robert C Martin

Software architecture is the process of taking functional and non-functional requirements and using them to create a high-level conceptual architecture. The high-level conceptual architecture will define a set of coarse-grained components. The coarse-grained components are then decomposed into finer grained components. The architecture specifies how components are arranged and defines the interfaces, responsibilities, and interactions between these components. We can think of the architecture as the shape of the system.

> **ARCHITECTURE VERSUS DESIGN**
>
> We might assert that software design is about organising classes withing modules and architecture is about organising modules. In practice the organisation of software elements from low level to high level is one discipline. For this reason in this document I will cover topics traditionally considered design.

Managing Dependencies

The goal of software architecture is to create a shape that minimises the effort required to develop, deploy, operate, and maintain the system. Ideally the effort involved in implementing a new feature should depend only on the complexity of that feature and not on the shape of the system. We want our components to have well-defined boundaries, well specified responsibilities, and well controlled dependencies.

> **MODULES, COMPONENTS AND SERVICES**
>
> It is hard to get a hard definition of the differences between a module and a component. I have read that a module logically groups together functions and classes without mandating any physical separation. Components on the other hand are the physical manifestation of modules. If the components are libraries, such as jars in Java or dll's in .NET, then they can be linked together in a single address space and communicate via function call. Components can also be services running in their own address space and communicating using networks calls (SOAP, Rest etc.).
>
> In this document from now on I will only refer to components irrespective of whether they are logically separate packages in a single executable, separate DLLs or deployable services.

The architecture must meet the needs of all specified use cases while at the same time satisfying non-functional requirements such as performance.

Policy and Detail

All components of a system can be roughly categorised as either policy or detail. Policy is the high-level business rules and detail is the supporting detail such as IO, databases and UI needed to support the policy.

We should architect the system such that the details are irrelevant to the policy and furthermore such that the detailed can be delayed as long as possible. The benefit of such a delay is that the longer we wait the more we know and the more likely we are to get details such as the database correct. We can even try plugging in different document and relational databases to see which works best.

Stable Dependencies and Stable Abstractions

Within a system some volatile components will be expected to change frequently. Consequently, we want to ensure such components are easy to change. Even a component that is well designed for change will itself become hard to change If we give it a load of

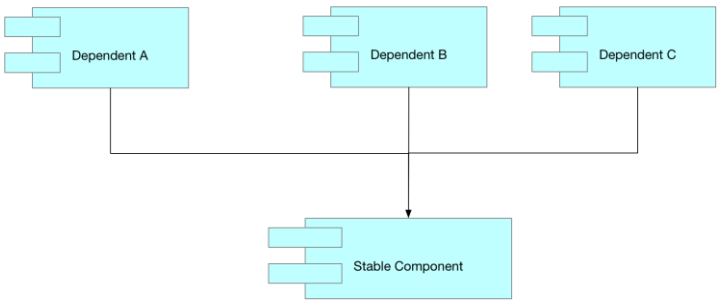
incoming dependencies. In general, adding incoming dependencies (dependents) to a module makes it harder to change as any changes need to be coordinated with all such dependents.

If a module has many outgoing dependencies it is likely that it will have to change frequently in response to changes in those dependencies. We should ensure that such modules are easy to change. Conversely a module with no outgoing dependencies is said to be independent. There is no module that can force it to change.

Categorising Stability

Stable Component

Any changes to Stable Component is stable because any changes to it must be co-ordinated with all its dependents.

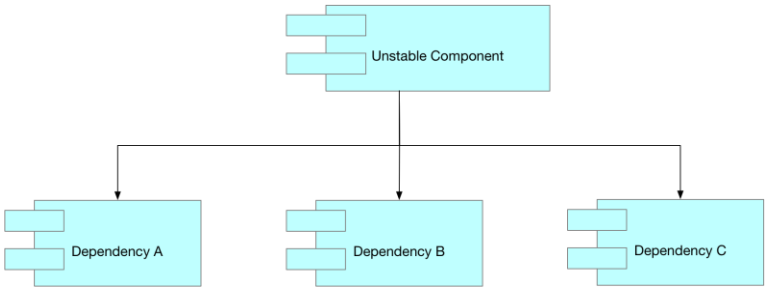


Irresponsible Component

An irresponsible component has no dependents (no incoming dependency arrows).

Unstable Component

The Unstable Component is unstable because any changes to any of its dependencies will force it to change. Any such components should be designed to be easy to change.



Independent Component

An independent component has no dependencies and so nothing can force it to change.

Arranging Dependencies

We want to arrange our components so the dependencies flow from the least stable components to the most stable. We would expect the components holding the high-level policy to be the most stable components in the system as they depend on nothing and most things depend on them.

Because the high-level policy modules are very stable this can lead to a system that is hard to extend. To mitigate this, we make very stable components abstract. Typically, the more stable

a component the higher the percentage of its types will be interfaces and abstract classes. Abstractness flows in the direction of stability. This is DIP at the component level.

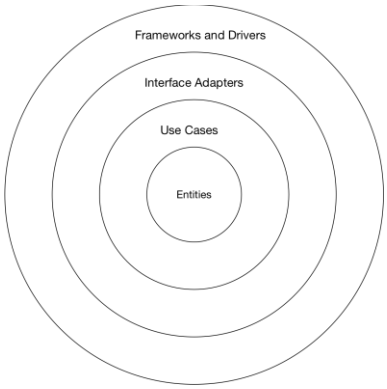
We can use plugins to create scalable and extensible architectures. The policy contains the core business logic. Plugins provide all the other non-business logic such as UI logic and persistence. The dependencies point from the plug-in components to to policy modules. This can be extended to more levels. Lower-level modules always depend on higher level modules. A module level is defined by how far it is from the IO.

Architectures

Type	Description
Clean Architecture	
Domain Model	Layered architecture that uses presentation layer, application layer, domain layer and infrastructure layer.
Layered	Use Tiers instead of layers
CQRS	A Layered architecture with two columns; one for commands and one for queries. Each column can have its own architecture
Event Sourcing	Evolved from CQRS with focus on events rather than data
Monolithic	Enough said
Microservices	https://martinfowler.com/articles/microservices.html
Hexagonal	
Service Oriented Architecture	

Clean Architecture

Robert C Martin defines the clean architecture in his book of the same name. Each level is a concentric circle and all dependences inwards towards higher level modules



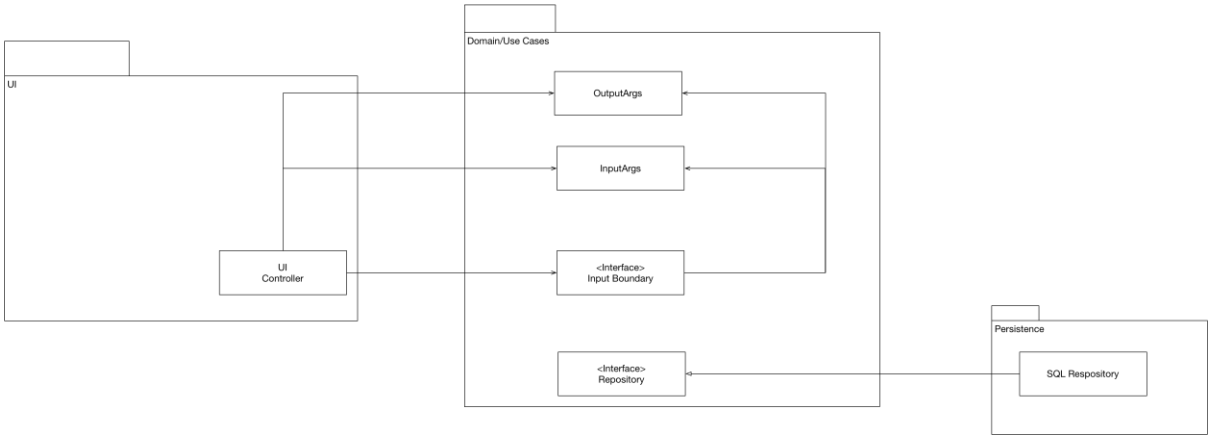
Within the business rules we have two different kinds of types. An entity has both data and behaviour and carries out critical business functionality. Entities are the highest-level modules in the business layer. Use cases use the entities and are one level lower. The entities are applicable to multiple applications whereas use cases only apply to one application. Use cases should accept simple request objects and produce simple response objects. These request/response objects should not depend on things like http requests. They should also not use the entities themselves.

The data that crosses the boundaries should be simple data structures, function arguments or data transfer objects. We should never pass things like entities or database rows.

	Description
Entities	Enterprise level business concepts. Has data and behaviour
Use Cases	Application specific logic that orchestrates the entities.
Interface Adaptors	Convert data from the format of use cases and entities to the format needed by user interfaces, databases etc.

The main function is the lowest and dirties part of the system. It depends on everything and sets up all dependency injection. Main is of course itself a plug-in. If we consider a simple

scenario where we merge the Entities and use cases and add persistence and UI we see something like this



Domain Model

The domain model is a layered architecture where the layers are presentation, application, domain, and infrastructure. The domain layer consists of entities and value objects and their state and behaviour. Entities have identity whereas value objects do not. In order to make a more manageable design entities can be grouped together into aggregates.

Domain Layer

A domain layer focusses on behaviour rather than data. It consists of entities, their behaviours, and the relationships between them. Each Entity represents important entity in the domain. In addition to Entities, we have Value Objects and Services. Services are classes utilise multiple entities and are known as Entity Services. The objects in the domain layer are grouped into modules where each module is a .NET namespace.

- One bounded context maps to one domain model
- One bounded context can consist of multiple modules
- One domain model can consist of multiple modules
- Each module is a .NET namespace.

Entity

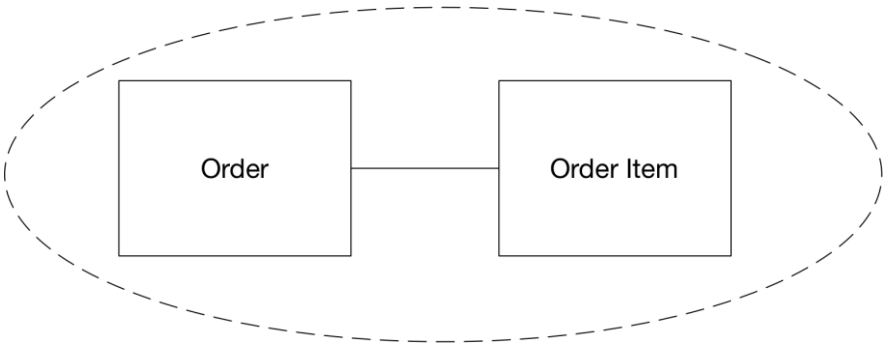
An Entity has identity. Two objects with the same attributes are not necessarily the same. Objects that need and Id are entities in the domain layer.

Value Object

In contrast to an entity, a value object is just data grouped together.

Aggregates

Where entities are often being used together, we can treat them as being logically grouped together into an aggregate. An aggregate has a root. Consider an order and its order items. The order items have identity, but most clients won't need direct access to them. The order becomes the root of an aggregate. Entities should only below in one aggregate. Value objects can be in multiple different aggregates



An aggregates root entity can reference its own child entities or the root entities of other aggregates. A child entity can only access root entities of other aggregates. The root of the aggregate is visible to other objects in the model. Aggregate child entities have identity and life but cannot be directly referenced outside the aggregate. If a root provides reference to child items the receiver should treat them as transient.

Services

Implement logic that utilise multiple entities and aggregates to carry out a business action. The actions of domain services are defined in the requirements. A repository is a domain service

Repositories

There should be one repository per aggregate root.

Domain model classes should not have direct knowledge of how to persist and load objects from the persistence store.

POCO means Plain Old CLR Object. Persistence is not a domain model responsibility. The domain model will define a repository interface and the infrastructure layer will implement it. The infrastructure level implementation can be injected in using Inversion of Control.

Layered Architecture

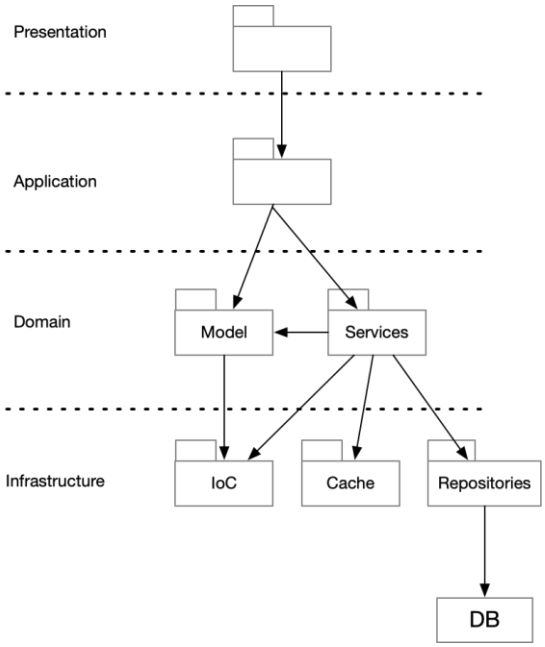
In a layered architecture layers determine the division of components into logical groups.

> **LAYERS VERSUS TIERS**
>
> Tiers describe the physical location of components on networks and servers. Layers say nothing about where components are deployed

The architect should consider whether layers exist purely to provide logical separation or to also provide physical separation. Reasons to use physical separation are shared business logic, scalability, or security. The components within each layer can be further decomposed into sub layers.

A modern layered architecture can look as follows

Modern Layered Architecture



Presentation layer

Anything that populates the presentation layer is known as the view model. The input model consists of actions originating in the UI that cause back end actions.

Application layer

Helps to separate presentation and domain. Takes care of use cases by executing and coordinating work done in lower layers of the stack.

Domain layer

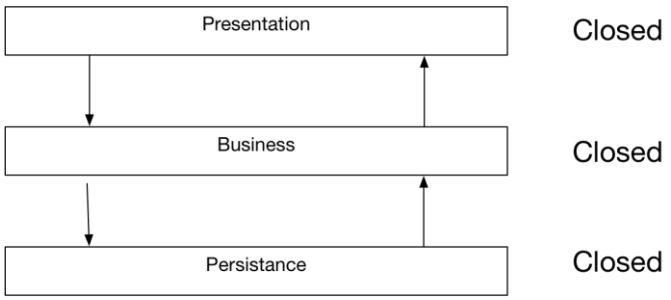
All the business logic. An Entity should provide data and behaviour. In addition to entities the domain model provides domain services. Domain services operate on multiple entities to achieve some task.

Infrastructure layer

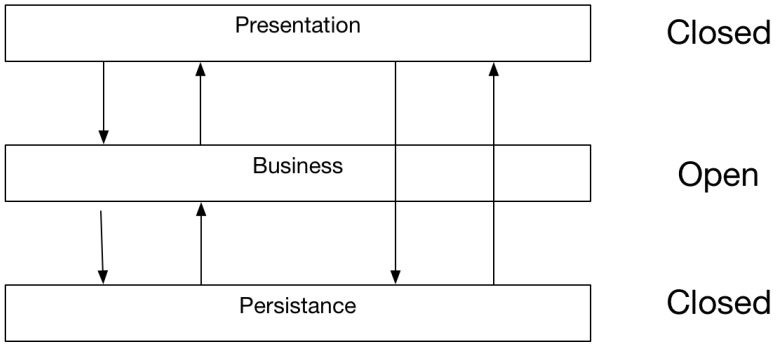
Data access, IoC containers, logging, caching

Interface Between Layers

Layers can be closed or open. If a layer is closed it cannot be bypassed. The layer above it must go through it.



If a layer is marked as open, then a layer above it can bypass that layer.



Layers Of Isolation

If all the layers are closed and we have well defined contracts between the layers we reduce coupling. In additions we can replace any layer without impacting the other layers. If we change the interface to any layer, we only impact the layer immediately above. The trade-off is that crossing layers is expensive, especially when deployed on tiers. Skipping layers can increase performance. The benefits of increased scalability, maintainability, extensibility, and flexibility often outweighs the performance costs.

If we mark a layer or multiple layers as open, we increase coupling which can lead to a more brittle architecture.

Sinkhole Anti-pattern

Agile Design

Agile design is a process, not an event. It's the continuous application of principles, patterns, and practices to improve the structure and readability of the software. It is the dedication to keep the design of the system as simple, clean, and expressive as possible at all times. – Robert C Martin

Metrics

Coupling

If one package is tightly coupled to other packages it makes reuse harder. It also leads to unstable and brittle code bases where changing one part breaks other parts.

Cohesion

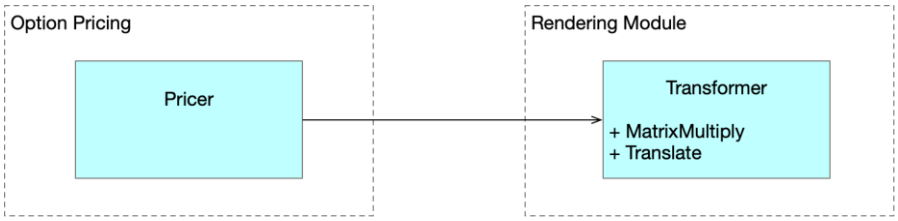
Cohesion measures how related the various parts of a module or class are to each other.

> Attempting to divide a cohesive module would only lead to increased coupling
>
> - Larry Constantine

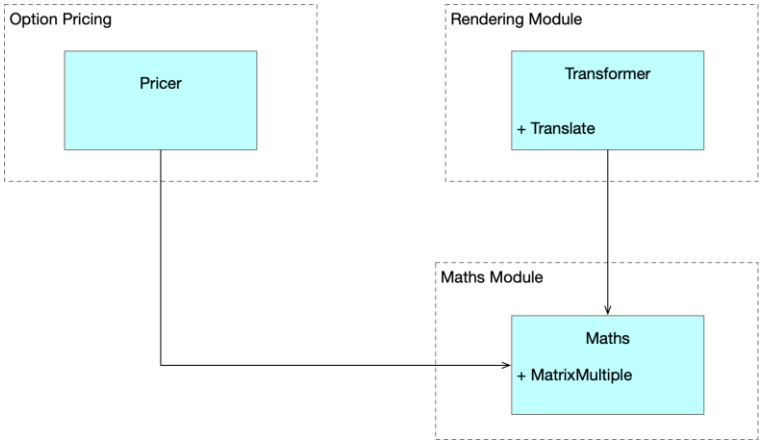
S.O.L.I.D.

Single Responsibility Principle

A single class should have only one responsibility. If a class has multiple responsibilities, then changes to one responsibility can break the logic of the other responsibilities thus increasing brittleness. In cases where a module exposes a class that provides multiple responsibilities, clients of that module may be forced to recompile and redeploy when a responsibility on which they have no logical dependency changes.



We can fix this as follows



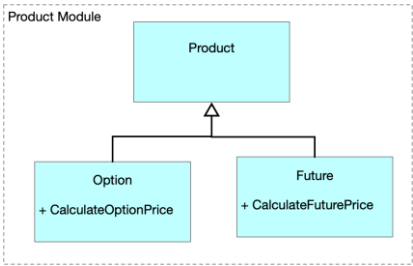
Classic violation

Mixing business rules and persistence is a classic example of breaking the Single Responsibility Principle. Patterns such as Façade, Proxy and DAO patterns can be used in such situations

OPEN CLOSED PRINCIPLE

"The Open/Closed principle is at the heart of object-oriented design. Conformance to this principle is what yields the greatest benefits claimed for object-oriented technology: flexibility, reusability, and maintainability. – Robert C Martin

Types should be open for extension and closed for modification. Consider the following simple type hierarchy and a piece of client code which breaks the Open/Closed Principle

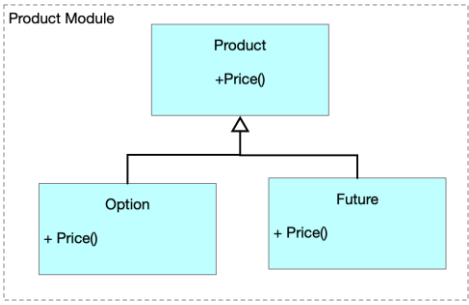



```
double PriceProducts(IEnumerable<Product> products)
{
    double total = 0.0;

    foreach (var product in products)
    {
                if (product is Option)
                total += ((Option)product).PriceOption();
          if (product is VarSwap)
                total += ((VarSwap)product).PriceVarswap();
    }

    return total;
}
```

The use of abstractions and polymorphism enable us to modify our hierarchy and `PriceProducts` method to obey the open closed principle.




```
double PriceProducts(IEnumerable<Product> products)
{
    double total = 0.0;

    foreach (var product in products)
          total += product.Price();
```

```
    return total;
```

The primary enablers of the open closed principle are abstraction and polymorphism.

Limitations

It is impossible to design software that is open to extension in all contexts.

LISKOV SUBSTITUTION PRINCIPLE

Replacing objects of a base-class with object of a subclass should not impact the correctness of the program. This is not just a question of type. An overridden method must have pre-conditions no more restrictive than the base class method and post-conditions no less restrictive than the base class method. This leads to an important point. The validity is determined by the clients of the design. Since are coded against the base-class that is all they know. A subclass cannot expect that clients obey preconditions more restrictive than those specified in the base class method. Similarly, a subclass cannot expect the clients to accept results that are less restricted than the results of the base class method.

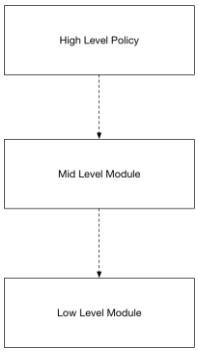
The LISKOV Substitution principle is a key facilitator of the Open/Closed principle because it enables code that is expressed in terms of a base class to be extended without modification.

INTERFACE SEGREGATION PRINCIPLE

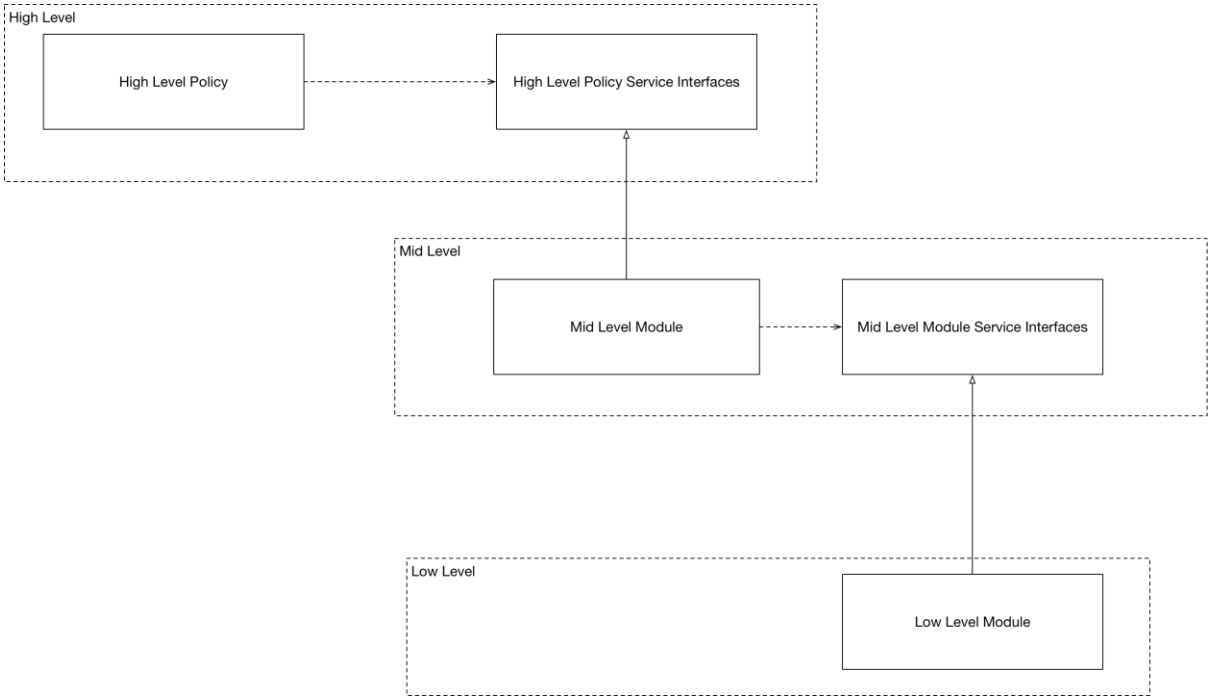
Interfaces should form a cohesive whole. If one class of client uses some methods of an interface and a different class of client uses other methods it would be better to split the interface into two smaller interfaces. The basic idea is that we do not make clients depend on interface methods that they never use,

DEPENDENCY INVERSION PRINCIPLE

In historic structured design methodologies high level modules depend on lower-level modules and policy depends on detail. The DI Principle says this is wrong as it leads to a situation where it is difficult to use the high-level policy in different contexts of use. Consider the following diagram. Because dependencies are transitive the high-level policy module is dependent on both the medium level module and the low-level module. Notice dependencies always flow in the direction of arrows.



The DI Principle states that the high-level policy that specifies business rules should be independent of low-level implementation details. We can restructure our illustrative example as follows. Notice the directions of the arrows. Higher level modules never have dependencies on lower level modules.


High Level
High Level Policy
High Level Policy Service Interfaces
Mid Level
Mid Level Module
Mid Level Module Service Interfaces
Low Level
Low Level Module

Code Smells

Rigid

Rigid code is hard to change often because a small change in one part of the codebase leads to large number of cascading changes in other seemingly unrelated parts.

Fragile

Fragile code is easy to break. A single change in one part of the codebase breaks other seemingly unrelated parts.

Immobile

If software has functionality that would be useful to use elsewhere but we don't do so because it is so hard to separate out the logic we say it is immobile.

Viscous

Viscosity makes it hard to change the software in a way that preserves the design leading developers to make hacks instead. Viscosity can be caused by the environment or by the code itself.

Needlessly Complex

Needlessly Repetitive

Can be fixed by factoring out appropriate abstractions.

Opaque

Code is written in a convoluted and hard to understand form.

Questions -Agile Design

What is SOLID?

	Column Header
Single Responsibility	A class should have one responsibility
Open Closed Principle	Open for extension, closed for modification
Liskov Substitution Principle	Replacing objects with sub-type instances should not break the code
Interface Segregation	Multiple specific interfaces are better than one large one
Dependency Inversion	One should depend on abstractions, not concrete types

Why is SRP important?

If a class has multiple responsibilities, then changes to one responsibility can break the logic of the other responsibilities thus increasing brittleness.

Creating an Architecture

To produce an architectural solution, the software architect considers

INPUTS TO THE ARCHITECTURAL PROCESS

- Use cases – how the application will be used
- Functional and non-functional requirements
- Technology requirements
- How the application will be deployed and managed

The architect must decide on the most appropriate application type, architectural style(s), technologies, and deployment methods. Each draft should be tested against scenarios known constraints, and quality attributes.

	Description
1. Goal	A goal might be to create a prototype or a complex systems architecture
2. Key Scenarios and Use Cases	These will be used to evaluate candidate architecture
3. Application Type	Are we building a thick client, web application etc.
4. Deployment Constraints.	Corporate policy might restrict the protocols or network topology on which the application will be deployed. Attributes such as security and reliability will also impact the deployment
5. Architectural Styles	Come up with architectural styles such as message bus, SOA etc.
6. Decide on technologies	The choice of technology will be influenced by the application type, deployment constraints and architectural styles.
7. Whiteboard the architecture	If you can't it is sure you don't understand it
8. Quality attributes	Consider Usability, performance, reliability, security
9. Cross-cutting concerns	Authentication and authorization, caching, configuration, Logging

Immutability

Immutability is a design property that has two main benefits. Firstly, it makes code easier to reason about. If I pass an immutable object as the parameter to a method, I do not have to look inside that method to see what it does to my object. I know it cannot do anything. Secondly, immutable objects do not require synchronization to work correctly in the presence of multiple threads of execution. Before we look at the new Immutable types that .NET supports, we should look at some different kinds of immutability.

TABLE 1 KINDS OF MUTABILITY AND IMMUTABILITY

Mutable	A type whose internal state can be modified in place.
Immutable	A type who internal state cannot be modified in place. Pseudo mutating methods create new objects that share much of the same memory as the original object but with some new memory that describes the change
Freezable	A type that can be mutated until such time as it is frozen after which it cannot be changed
Read only	A type through which references cannot mutate the underlying data. Typically, the underlying type can be mutated via other references

Another benefit of immutable collections is that we get a record of state changes. When we change a mutable collection, the original state is lost.

Immutable collections are also useful when we have snapshot semantics. If we want some threads to see all or none of a batch of modifications. If we were to use a concurrent collection or basic collection, we would need to use a lock and copy the whole collection.