Dotnet Architectural Principles

You should architect and design software solutions with maintainability in mind. These principles will guide you towards building applications out of discrete components that are not tightly coupled to other parts of your application, but rather communicate through explicit interfaces or messaging systems.

# Common design principles

## Separation of concerns

A guiding principle when developing is separation of concerns. This principle asserts that software should be separated based on the kinds of work it performs. For instance, consider an application that includes logic for identifying noteworthy items to display to the user, and which formats such items in a particular way to make them more noticeable. The behaviour responsible for choosing which items to format should be kept separate from the behaviour responsible for formatting the items, since these behaviours are separate concerns that are only coincidentally related to one another.

Architecturally, applications can be logically built to follow this principle by separating core business behaviour from infrastructure and user interface logic. Ideally, business rules and logic should reside in a separate project, which should not depend on other projects in the application. This separation helps ensure that the business model is easy to test and can evolve without being tightly coupled to low level implementation details (it also helps if infrastructure concerns depend on abstractions defined in the business layer). Separation of concerns is a key consideration behind the use of layers in application architectures.

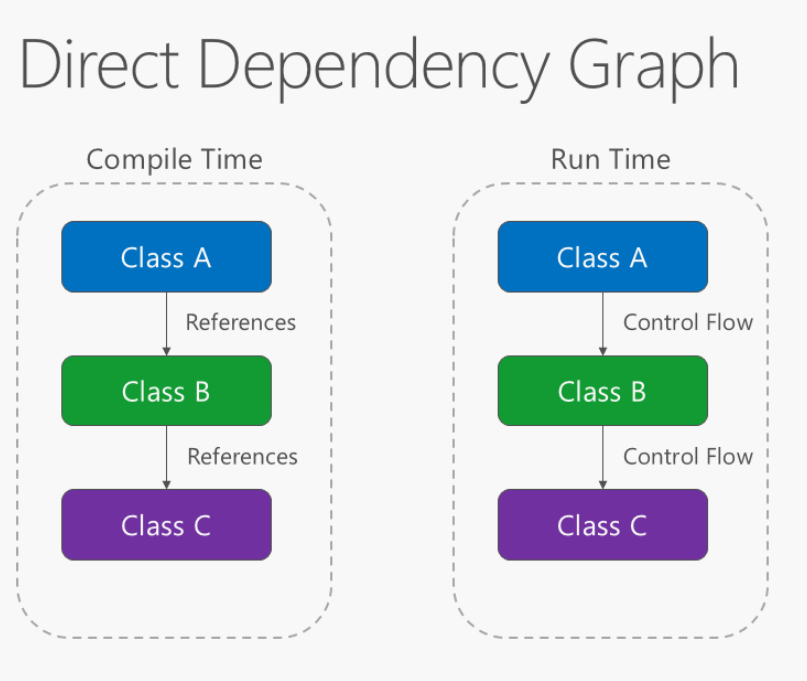
Encapsulation

Different parts of an application should use encapsulation to insulate them from other parts of the application. Application components and layers should be able to adjust their internal implementation without breaking their collaborators if external contracts are not violated. Proper use of encapsulation helps achieve loose coupling and modularity in application designs, since objects and packages can be replaced with alternative implementations if the same interface is maintained.

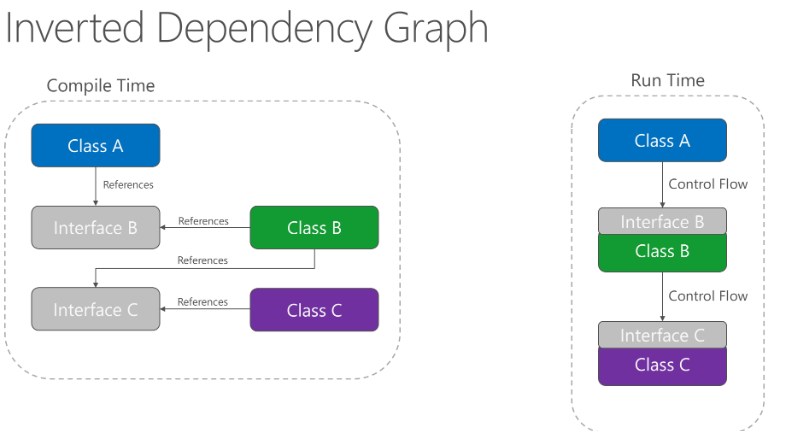
In classes, encapsulation is achieved by limiting outside access to the classes internal state. If an outside actor wants to manipulate the state of the object, it should do so through a well-defined function (or property setter), rather than having direct access to the private state of the object. Likewise, application components and applications themselves should expose well-defined interfaces for their collaborators to use, rather than allowing their state to be modified directly. This approach frees the applications internal design to evolve over time without worrying that doing so will break collaborators, so if the public contracts are maintained.

Mutable global state is antithetical to encapsulation. A value fetched from mutable global state in one function cannot be relied upon to have the same value in another function (or even further in the same function). Understanding concerns with mutable global state is one of the reasons programming languages like C# have support for different scoping rules, which are used everywhere from statements to methods to classes. It’s worth noting that data driven architectures which rely on a central database for integration within and between applications are, themselves, choosing to depend on the mutable global state represented by the database. A key consideration in domain driven design and clean architecture is how to encapsulate access to data, and how to ensure application state is not made invalid by direct access to its persistence format.

## Dependency Inversion

The direction of dependency within the application should be in the direction of abstraction, not implementation details. Most applications are written such that compile-time dependency flows in the direction of runtime execution, producing a direct dependency graph. That is, if class A calls a method of class B and class B calls a method of class C, then at compile time class A will depend on class B, and class B will depend on class C, as shown:  
  


Applying the dependency inversion principle allows A to call methods on an abstraction that B implements, making it possible for A to call B at run time, but for B to depend on an interface controlled by A at compile time (thus *inverting* the typical compile-time dependency). At run time, the flow of program execution remains unchanged, but the introduction of interfaces means that different implementations of these interfaces can easily be plugged in. Inverted dependency graph:



Dependency inversion is a key part of building loosely coupled applications, since implementation details can be written to depend on and implement higher-level abstractions, rather than the other way around. The resulting applications are more testable, modular and maintainable as a result. The practice of dependency injection is made possible by following the dependency inversion principle.

## Explicit dependencies

Methods and classes should explicitly require any collaborating objects they need in order to function correctly. It is called the ‘*Explicit Dependencies Principle*’. Class constructors provide an opportunity for classes to identify the things they need in order to be in a valid state and to function properly. If you define classes that can be constructed and called, but that will only function properly if certain global or infrastructure components are in place, these classes are being dishonest with their clients. The constructor contract is telling the client that it only needs the things specified (possibly nothing if the class is just using a parameter less constructor), but then at runtime it turns out the object really did need something else.

By following the explicit dependencies principle, your classes and methods are being honest with their clients about what they need in order to function. Following the principle makes your code more self-documenting and your coding contracts more user friendly, since users will come to trust that if they provide what’s required in the form of method or constructor parameters, the objects they’re working with will behave correctly at run time.

## Single responsibility

The single responsibility principle applies to object-oriented design but can also be considered as an architectural principle similar to separation of concerns. It states that objects should have only one responsibility and that they should have only one reason to change. Specifically, the only situation in which the object should change is if the manner in which it performs its one responsibility must be updated. Following this principle helps to produce more loosely coupled and modular systems, since many kinds of new behaviour can be implemented as new classes, rather than by adding additional responsibility to existing classes. Adding new classes is always safer than changing existing classes, since no code yet depends on the new classes.

In a monolithic application, we can apply the single responsibility principle at a high level to the layers in the application. Presentation responsibility should remain in the UI project, while data access responsibility should be kept within an infrastructure project. Business logic should be kept in the application core project, where it can be easily tested and can evolve independently from other responsibilities.

When this principle is applied to application architecture and taken to its logical endpoint, you get microservices. A given microservice should have a single responsibility. If you need to extend the behaviour of a system, it’s usually better to do it by adding additional microservices, rather than by adding responsibility to an existing one.

## Don’t repeat yourself (DRY)

The application should avoid specifying behaviour related to a particular concept in multiple places as this practice is a frequent source of errors. At some point a change in requirements will require changing this behaviour. It’s likely that at least one instance of the behaviour will fail to be updated, and the system will behave inconsistently.

Rather than duplicating logic, encapsulate it in a programming construct. Make this construct the single authority over this behaviour and have any other part of the application that requires this behaviour use the new construct.

**Note:** avoid binding together behaviour that is only coincidentally repetitive. For example, just because two constants have the same value, that doesn’t mean you should have only one constant, if conceptually they’re referring to different things. Duplication is always preferable to coupling to the wrong abstraction.

## Bounded contexts

Bounded contexts are a central pattern in domain driven design. They provide a way of tackling complexity in large applications or organisations by breaking it up into separate conceptual modules. Each conceptual module then represents a context that is separated from other contexts (hence, bounded), and can evolve independently. Each bounded context should ideally be free to choose its own names for concepts within it and should have exclusive access to its own persistence store.

At minimum, individual web applications should strive to be their own bounded context, with their own persistence store for their business model, rather than sharing a database with other applications. Communication between bounded contexts occurs through programmatic interfaces, rather than through a shared database, which allows for business logic and events to take place in response to changes that take place. Bounded contexts map closely to microservices, which also are ideally implemented as their own bounded contexts.

## Architectural Agility

This principle highlights the importance of designing software architectures that can adapt to changing requirements and conditions. Architects should consider using techniques such as modular design, microservices or other architectural patterns that promote flexibility and adaptability.

The principle of architectural agility, while not as widely recognized as some other principles, is an important concept in modern software architecture. It emphasizes the need to design software architectures that can adapt to changing requirements, conditions or technologies. The term “Architectural Agility” is more of a philosophy that encompasses several practices and pattens that enable a system to evolve more easily. Although the term itself might not be as widely referenced, the underlying idea is echoed in various architectural styles, methodologies and literature. The concepts of architectural agility is closely related to agile software development methodologies, which prioritize flexibility and adaptability in software projects.

Other areas of software development that support architectural agility include:

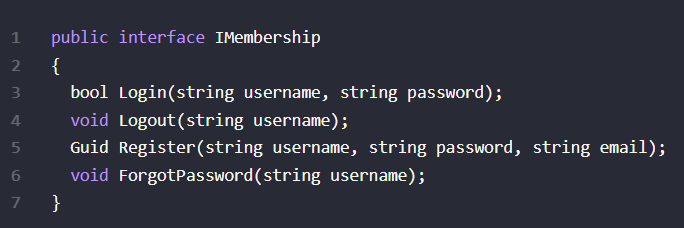
* Agile Software development, whose methodologies focus on iterative development and adapting to change, often allowing for ongoing architectural agility.
* Microservices Architecture, which is designed to break systems into small, loosely coupled services that can be developed, deployed and scaled independently. Each microservice can evolve without requiring the entire system to be impacted.
* Continuous delivery and DevOps practices that emphasize close collaboration between development and operations teams, and promote rapid, frequent reliable delivery of software.

## Interface Segregation Principle

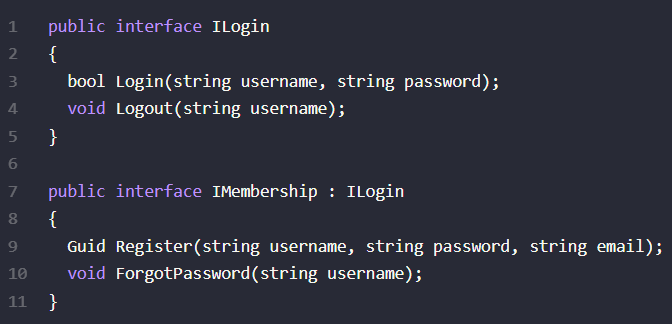
The interface segregation principle (ISP) states that clients should not be forced to depend on methods that they do not use. Interfaces should belong to clients, not to libraries or hierarchies. Application developers should favour thin, focused interfaces to “fat” interfaces that offer more functionality than a particular class or method needs.



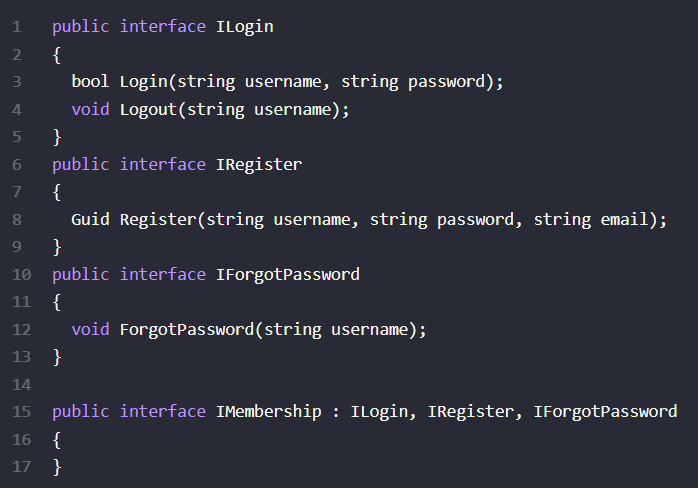
In many languages, such as C# interfaces can inherit from multiple other interfaces. Thus if you need a larger interface in some parts of the application, but not in others, you may be able to compose it from two or more other interfaces. This is also a good approach to keep in mind if you find yourself refactoring a legacy codebase, which already has large interfaces that you can’t break. Consider an interface like this one:



It’s easy to imagine such an interface growing completely out of control and having more functionality than any one class would ever require. To keep, say a login form from having more methods on it than it needs, you could create a login-specific interface and have the existing interface extend from it:



You could extend this further, as required, perhaps ending up with an original “fat” interface that only exists for legacy reasons, and is totally composed of other interfaces:



Ideally your thin interface should be cohesive, meaning they have groups of operations that logically belong together. This will prevent you from ending up with one interface per method most of the time in real world systems (as opposed to the above trivial example).

Another benefit of smaller interfaces is that they are easier to implement fully. They also provide greater flexibility in how you implement functionality, since parts of a larger interface can be implemented in different ways. Consider the repository pattern, which usually includes methods for reading as well as writing. A common performance pattern for database reads it so add a caching layer, but this generally only makes sense for read operations. Likewise, scalability can often be improved by queuing commands (like write operations) rather than executing them immediately, but you wouldn’t queue queries. Thus having an IRepository interface composed of an IReadOnlyRepository and an IWriteRepository would allow base implementations that go against a data store and separate implementations that employ caching and queuing as well.

## SOLID