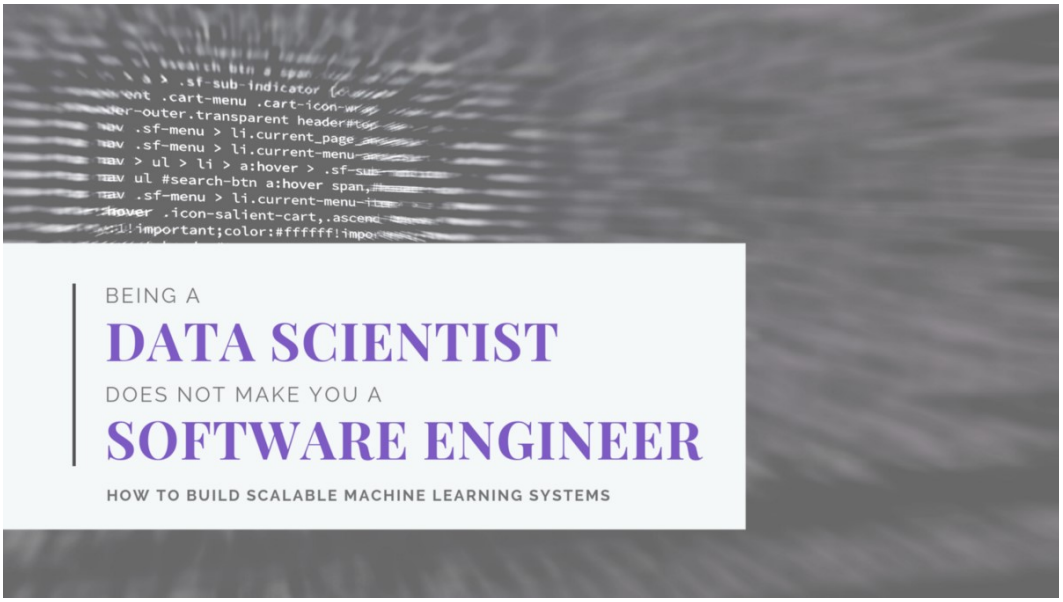


Being a Data Scientist does not make you a Software Engineer!

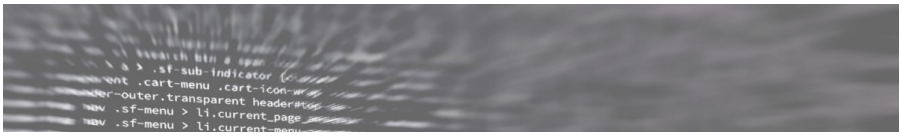
towardsdatascience.com

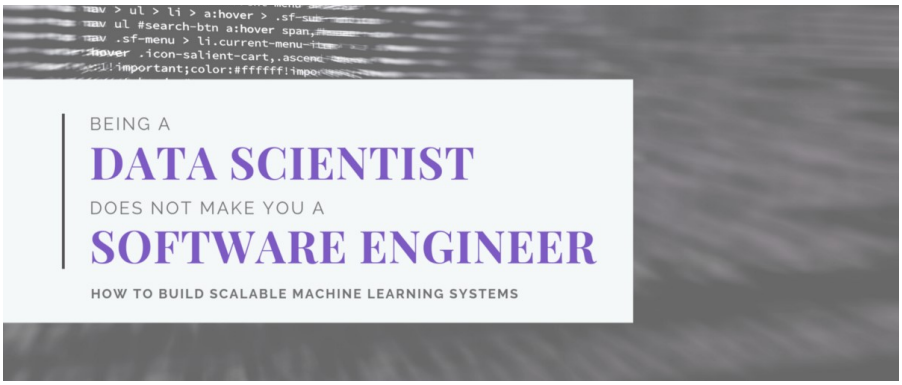
Data Science in the Real World

How to build scalable Machine Learning systems—Part 1/2

Semi Koen

Mar 2★





Disclaimer

Hopefully I caught your attention with the controversial title. Great! Now bear with me as I am going to show you how you can build a scalable architecture to surround your witty Data Science solution!

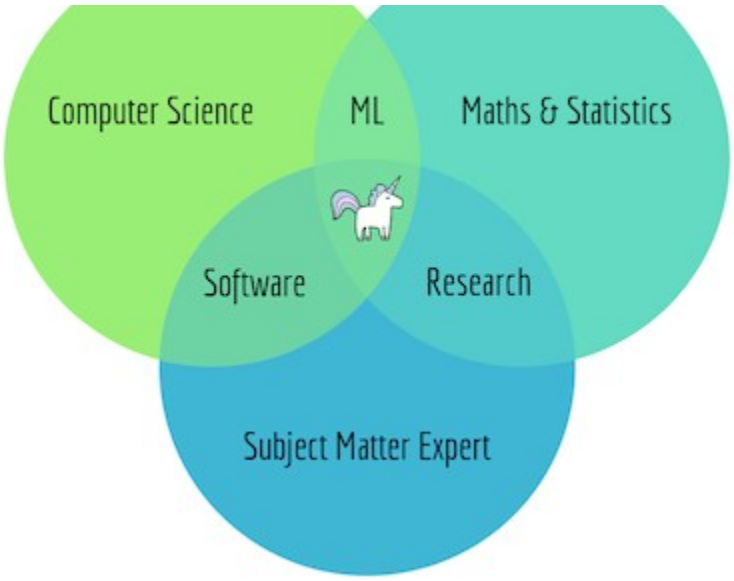
I am starting a **series of 2 articles** that will cover the basics of software engineering with regards to architecture and design and how to apply these on each step of the Machine Learning Pipeline:

Part 1: *Problem Statement | Architectural Styles | Design Patterns | SOLID*

Part 2: *Architecting a Machine Learning Pipeline*

Introduction

As we have seen before in the famous Venn diagram of Steven Geringer, Data Science is the intersection of 3 disciplines: Computer Science, Mathematics/Statistics and a particular Domain knowledge.



Data Science Venn Diagram [Copyright Steven Geringer]

Having basic (or even advanced) programming skills is key to put your end to end experiment together, however it does not mean that you have created an application that is production ready. Unless you have come into Data Science and Machine Learning (ML) from an **IT background** and have tangible experience into building enterprise, distributed, solid systems, your Jupyter notebook does not qualify as a great piece of software and sadly does not make you a Software Engineer!

What you have built is a great **prototype** of a predictive product, but you still have to push it through the engineering roadmap. What you need is a team of professional Software Engineers by your side to take your (disposable) proof of concept and turn it into a **performant**, **reliable**, **loosely coupled** and **scalable** system!

Everything is designed; few things are designed well!

In this series we will see some ideas of how this can be achieved... We will start with the basics in Part 1, and gradually design the holistic architecture in Part 2. The suggested architecture will be

technology agnostic. The ML pipeline will be broken down into layers with clear demarcation of responsibilities, and at each layer, we can choose from a number of technology stacks.

But let's start by defining how a successful solution looks like!

Problem Statement

The main objectives are to build a system that:

- *Reduces **latency;***
- *Is integrated but **loosely coupled** with the other parts of the system, e.g. data stores, reporting, graphical user interface;*
- *Can **scale** both horizontally and vertically;*
- *Is **message driven** i.e. the system communicates via asynchronous, non-blocking message passing;*
- *Provides efficient computation with regards to **workload management;***
- *Is **fault-tolerant** and self healing i.e. breakdown management;*
- *Supports **batch** and **real-time** processing.*

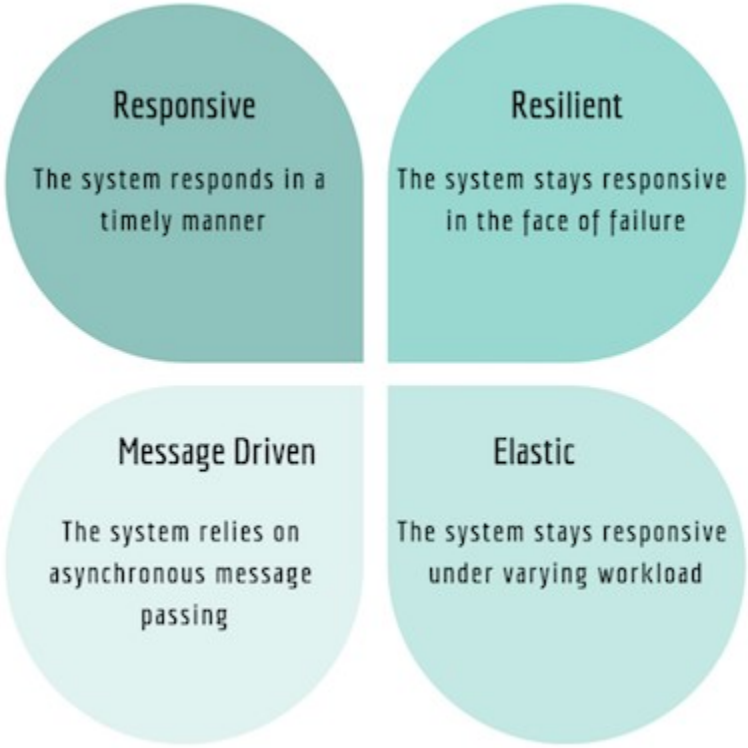
Architectural Styles

We will first introduce what a reactive system is and will proceed to a quick tour of the most prevalent architectural patterns.

Reactive Systems

The reactive systems design paradigm is a coherent approach to building better systems, which are designed according to the tenets of the Reactive Manifesto. Each reactive principle maps to an important system dimension of scalability:

- *Responsive* → Time
- *Elastic* → Load
- *Resilient* → Error
- *Message Driven* → Communication.



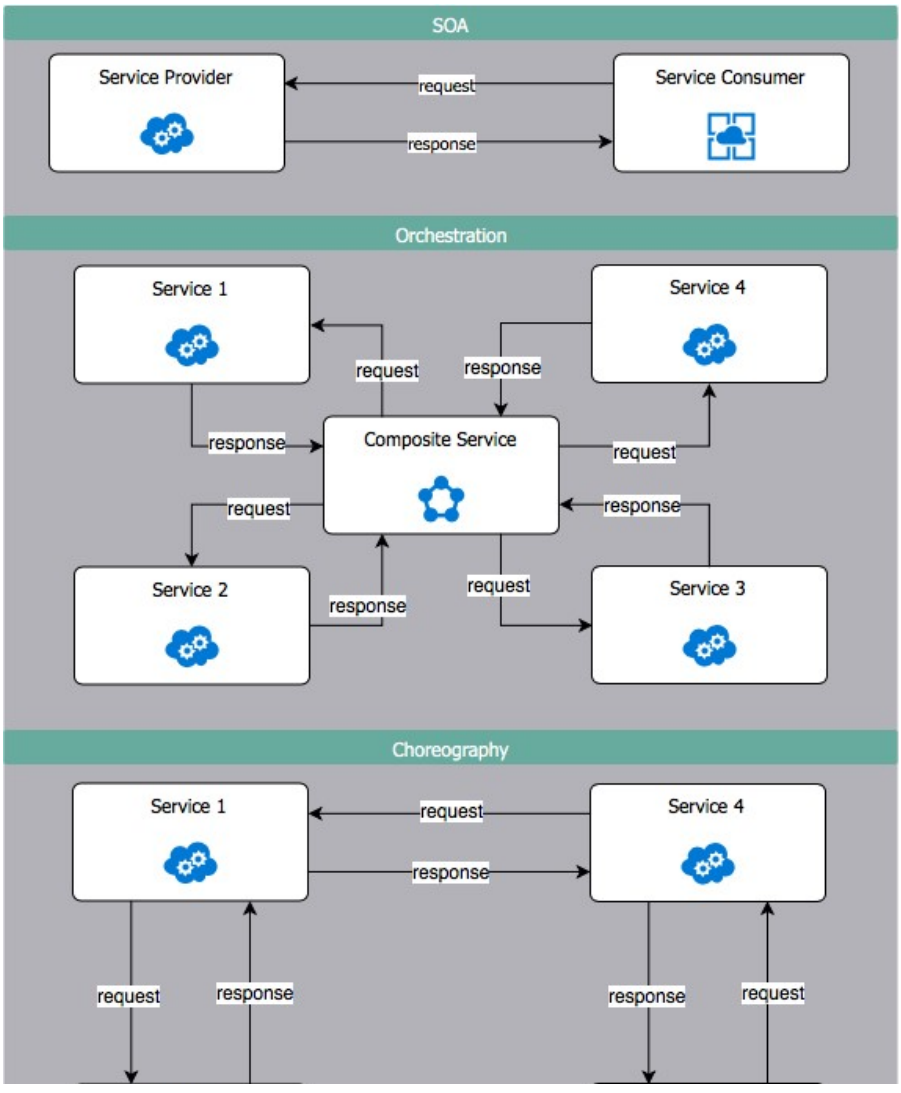
Features of Reactive Systems

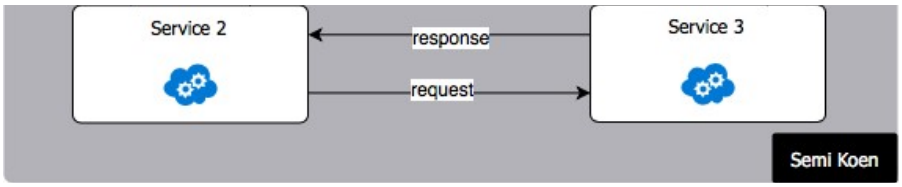
Service Oriented Architecture (SOA)

SOA centres around the concept of decomposing business problems into services. The services share information via the network and they also share code (i.e. common components) to maintain

consistency and reduce development effort.
The service **provider** publishes a contract that specifies the nature of the service and how to use it. The service **consumer** can locate the service metadata in the registry and develop the required client components to bind to it and use it.

An **orchestrator** is a composite service which is responsible for invoking and combining other services. Alternatively, **choreography** employs a decentralised approach for service composition, i.e. services interact with the exchange of messages/events.



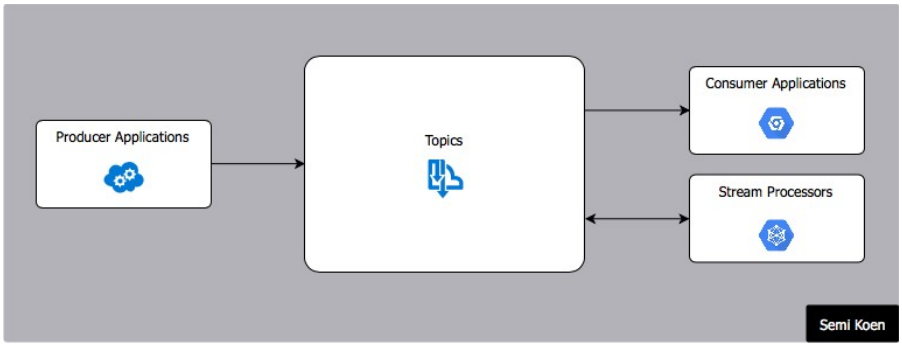


SOA

Streaming Architecture

A streaming architecture comprises of the following components:

- **Producers**: Applications that generate and send messages
- **Consumers**: Applications that subscribe to and consume messages
- **Topics**: Streams of records belonging to a particular category and stored as a sequence of ordered and immutable records partitioned and replicated across a distributed cluster
- **Stream Processors**: Applications that process messages in a certain manner (e.g. data transformations, ML models, etc).

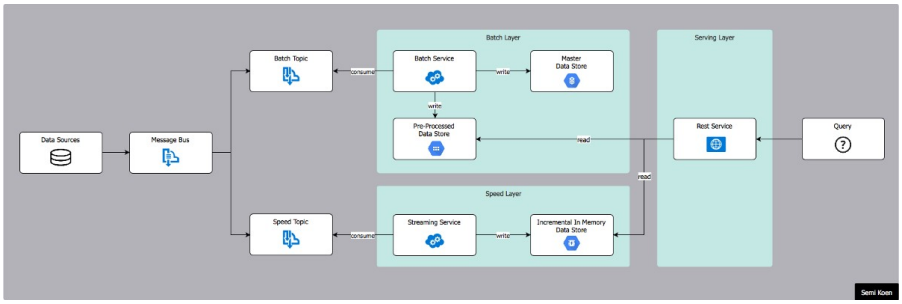


Streaming Architecture

Lambda Architecture

The Lambda (λ) Architecture is designed to handle both **real-time** and historically aggregated **batched data** in an integrated fashion. It separates the duties of real-time and batch processing while query layers present a unified view of all of the data.

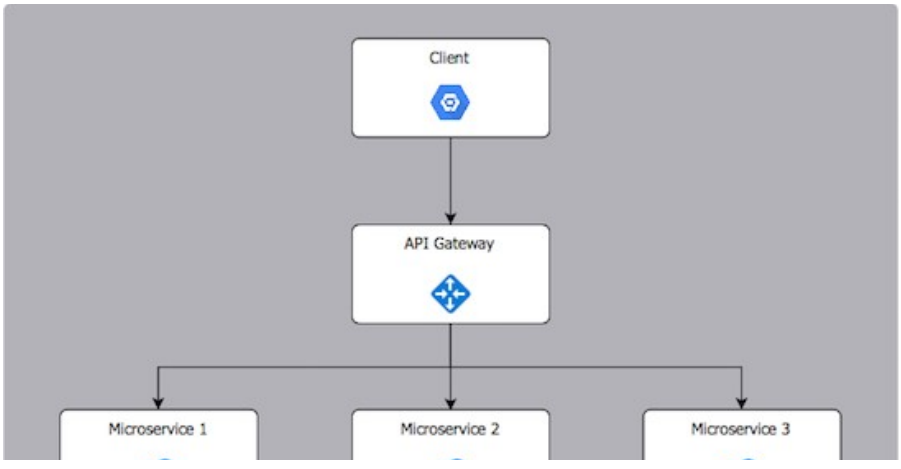
The concept is simple: When data is generated, it is processed before stored, so analysis can include data generated in the last second, the last minute, or the last hour by only processing the incoming data—not all the data.

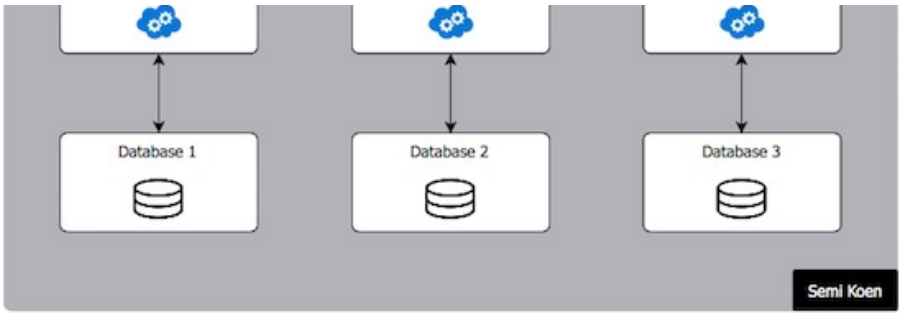


Lambda Architecture

Microservice Architecture

Microservices, is an architectural style that structures an application as a collection of small, autonomous, loosely coupled and collaborating services, modelled around a business domain. The services communicate using either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP. They can be developed and **deployed independently** of one another. Each service has its own database in order to be decoupled from other services.



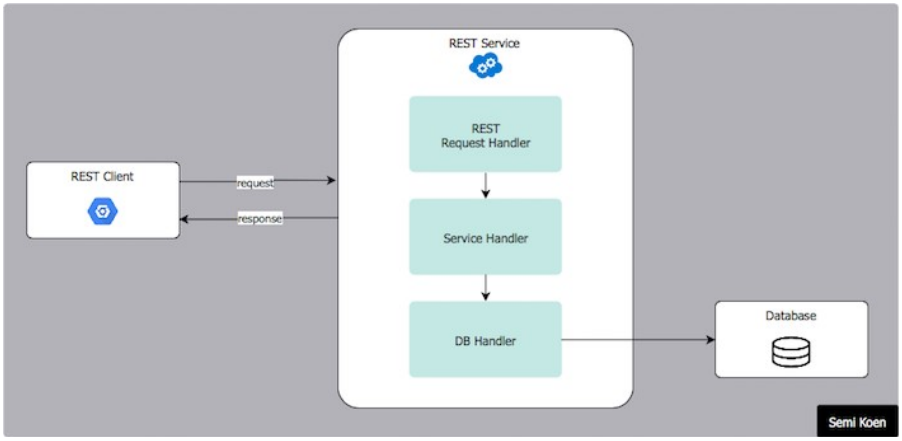


Microservices Architecture

Representational State Transfer (REST) Architecture

REST is an architectural style for developing **web services** and it builds upon existing features of the internet's HTTP. It allows transferring, accessing and manipulating textual data representations, in a stateless manner i.e. applications can communicate agnostically.

A RESTful API service is exposed through a Uniform Resource Locator (URL), which provides the capability of data being created, requested, updated, or deleted (CRUD). It is best used to manage systems by decoupling the information that is produced and consumed from the technologies that produce and consume it!



REST Architecture

Design Patterns

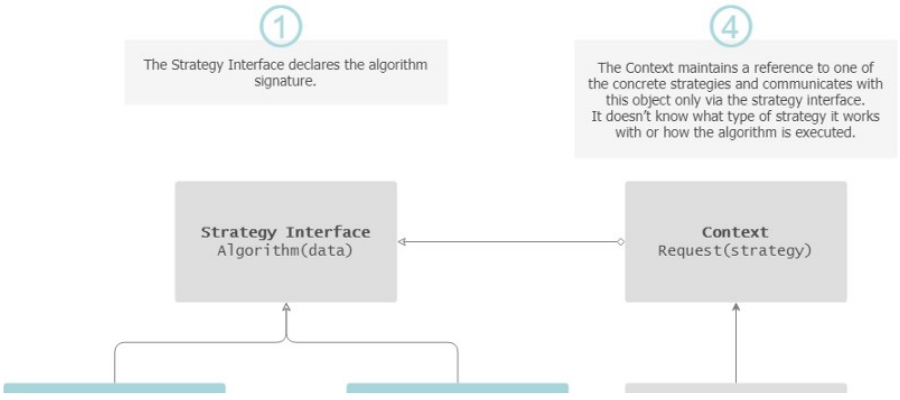
We will only scratch the surface on this topic and will only discuss those patterns that I may be referring to in the 2nd Part of the series. —[Hard to know just yet, but these are the patterns I use on a daily basis]

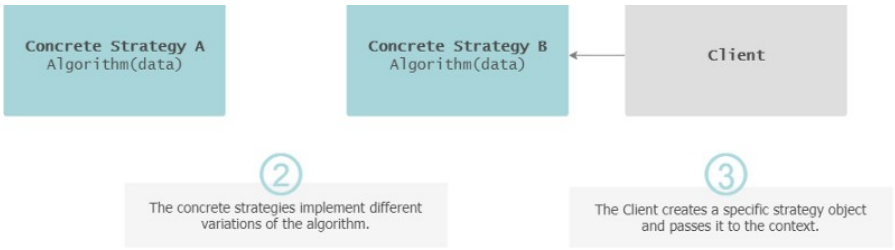
A software design pattern is an optimised, repeatable solution to a commonly occurring problem in software engineering. It is a template for solving a problem that can be used in many different situations.

Strategy

The Strategy pattern defines a family of algorithms, put each one in a separate class and make them **interchangeable**. Encapsulating the behaviour in separate classes, eliminates any conditional statements and the correct algorithm (i.e. strategy) is chosen at run-time.

— Indication for usage: There are different implementations of a business rule or different variants of an algorithm are needed.





Strategy Pattern

Template Method

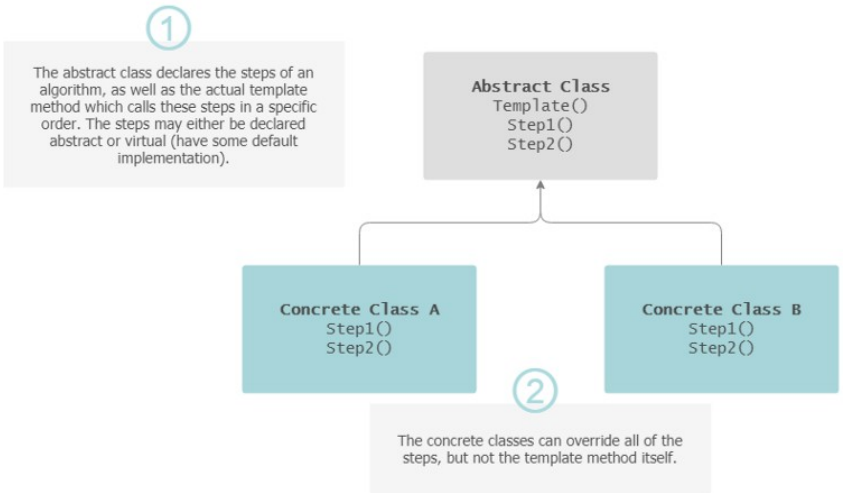
The Template Method intends to abstract out a common process from different procedures. It defines the **skeleton** of an algorithm, deferring some steps to sub-classes. The sub-classes can override some behaviour but cannot change the skeleton.

— Indication for usage: There is a consistent set of steps to follow but individual steps may have different implementations.

⭐ **Difference to Strategy Pattern**:

• Template: Algorithm is selected at **compile-time** by **sub-classing**.

• Strategy: Algorithm is selected at **run-time** by **containment**.

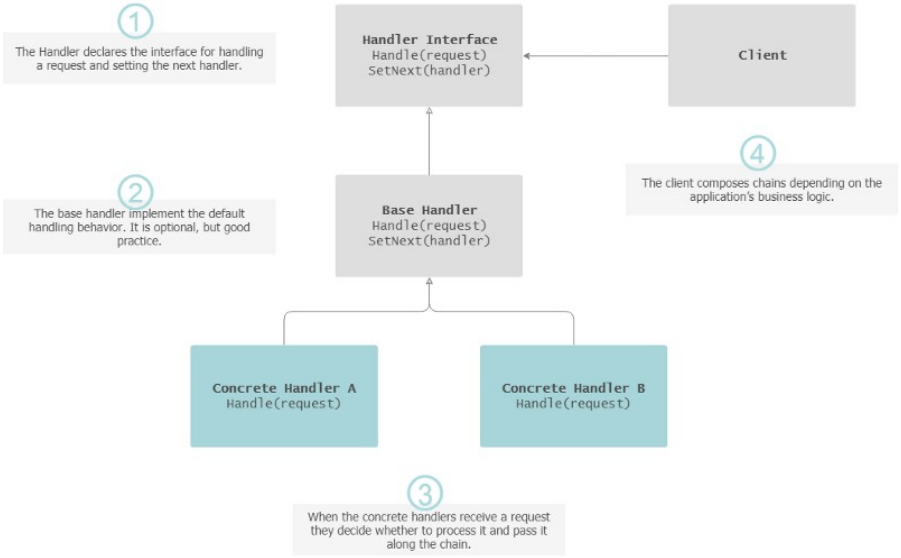


Template Method

Chain of responsibility

The Chain of Responsibility pattern suggests avoiding coupling the client (sender of requests) with the receiver, by enabling one or more **handlers** to cater for the requests. These handlers are linked into a chain i.e. each handler has a reference to the next handler in the chain.

— Indication for usage: More than one objects may handle a request, and the handler (nor the sequence) isn't known a priori.



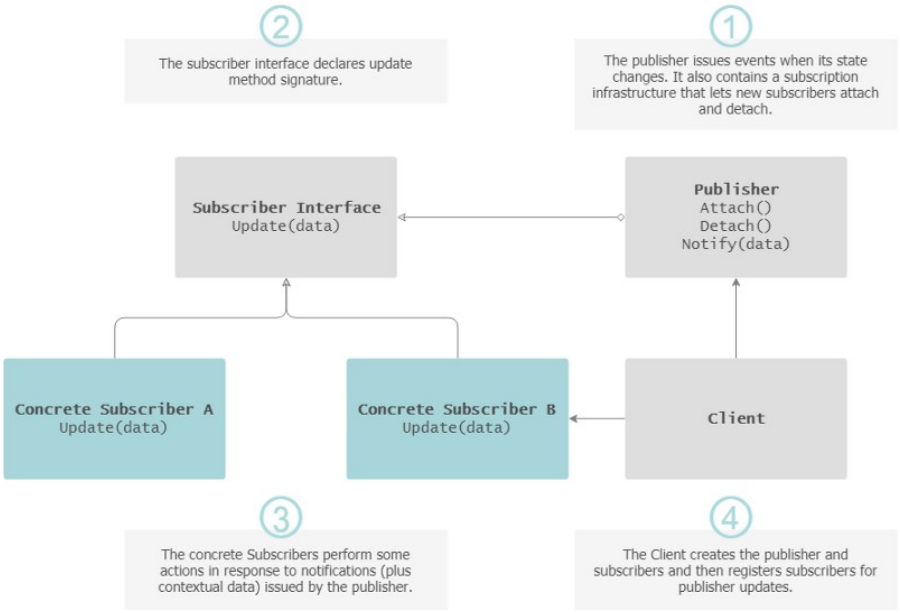
Chain of Responsibility

Observer

The Observer pattern (aka Publish/Subscribe or PubSub for short) enables easy **broadcast** of communication by defining a one-to-many dependency between objects, so that when one object undergoes a change in state, all its dependents are notified and

updated automatically. It is the observers responsibility to register the event they are ‘observing’.

— Indication for usage: When a change to one object requires changing others, and you don’t know how many objects need to be changed.

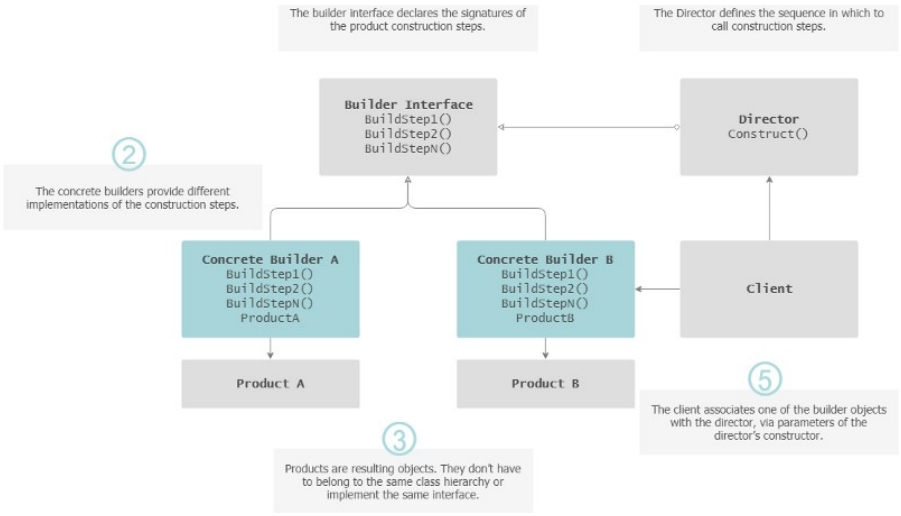


Observer Pattern

Builder

The Builder pattern is intended to construct a complex object in a **step-by-step** fashion and also separate the construction from its representation. In essence, it allows to produce different types and representations of an object using the same code.

— Indication for usage: Several kinds of complex objects can be built with the same overall build process, albeit the variation in the individual construction steps.

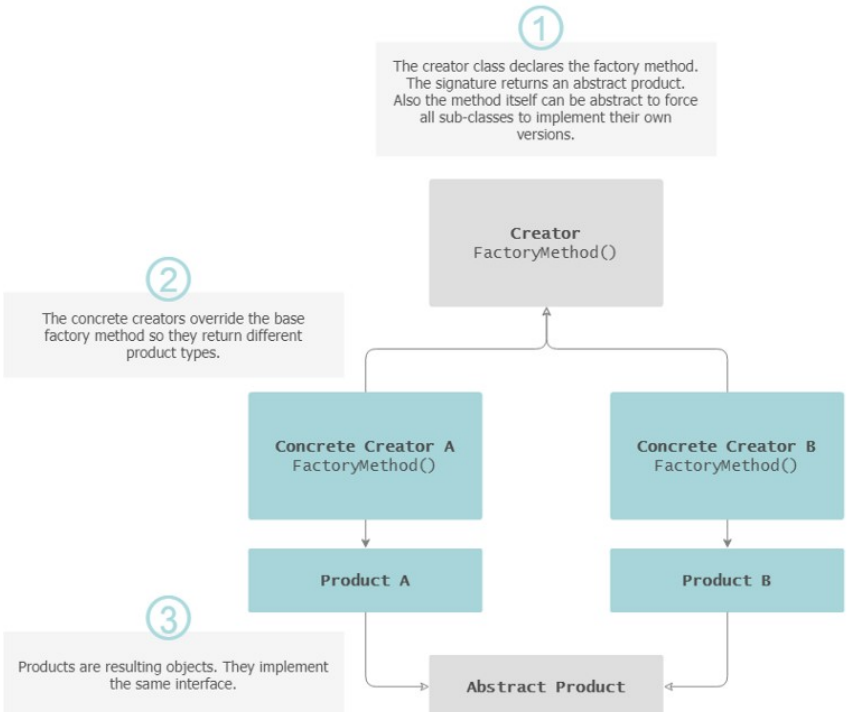


Builder Pattern

Factory Method

The Factory Method defines an interface for **creating objects**, but the instantiation is done by sub-classes.

— Indication for usage: The exact types and dependencies of the objects are not known beforehand.



Factory Method

Abstract Factory

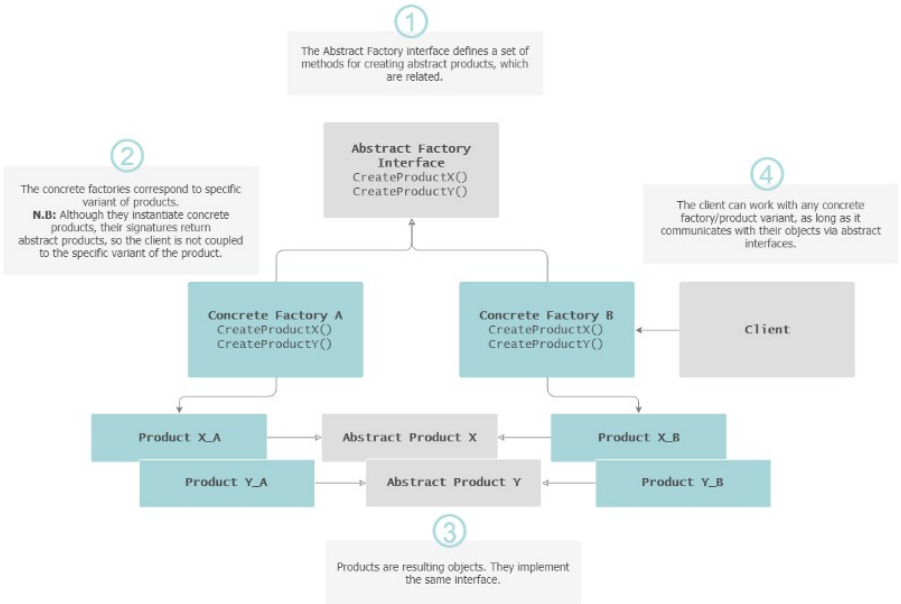
The Abstract Factory captures how to create **families of related objects** without specifying their concrete classes.

— Indication for usage: Different cases exist that require different implementations of sets of rules, that either unknown beforehand or extensibility is a concern.

⭐ **Difference to Abstract Method**:

• Abstract Factory: Creates other factories, and these factories in turn create objects derived from base classes.

• Factory Method: Creates objects that derive from a particular base class.



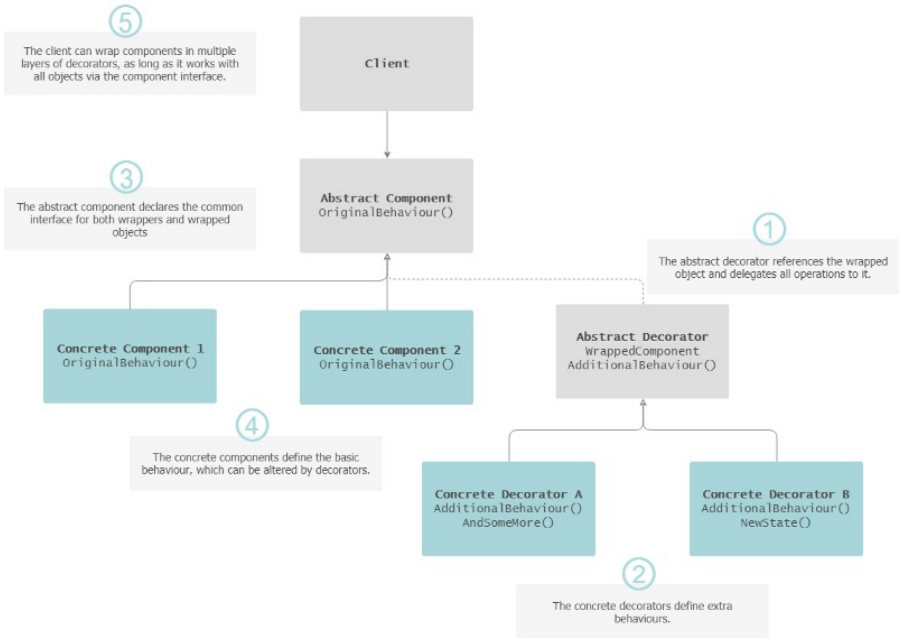
Abstract Factory

Decorator

The Decorator pattern attaches new responsibilities to an object dynamically, by placing it inside a special wrapper class that contains

these behaviours, so there is no impact to the signature of the original methods (composition over inheritance).

— Indication for usage: Assigning extra behaviours to objects at run-time without breaking the code that uses these objects.

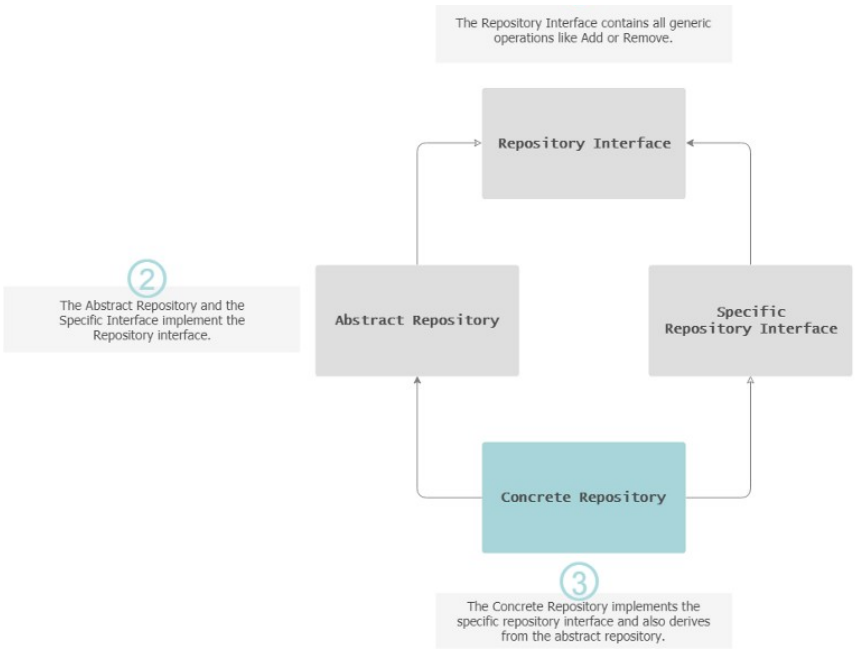


Decorator Pattern

Repository

The Repository pattern addresses code centralisation for data retrieval and persistence and provides an **abstraction for data access** operations i.e. acts like an in-memory collection of domain objects to allow for CRUD methods to be performed, and removes any database concerns.

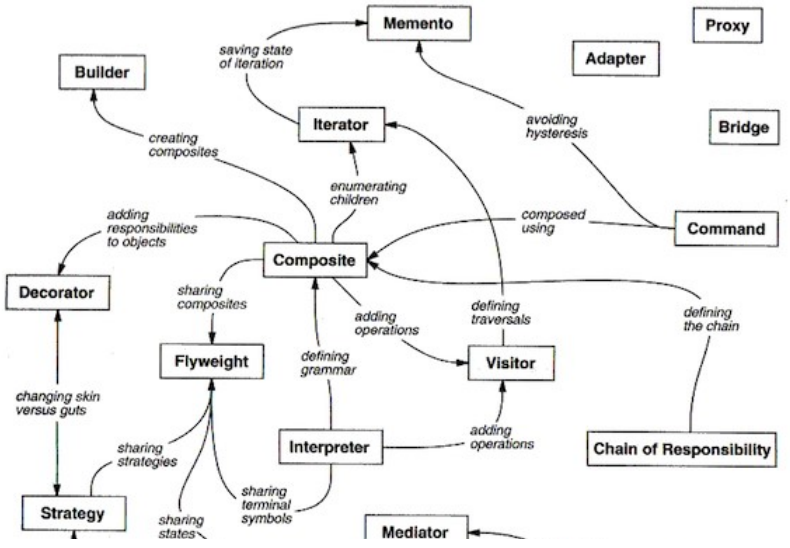
— Indication for usage: Decoupling the business logic with data access code.



Repository Pattern

Little bonus

Want to learn more about patterns? Start with the de-facto book of the 'Gang of Four', namely: '[Design patterns: elements of reusable object-oriented software]'. The following diagram with the patterns' relationships is noteworthy—*pretty spiffy, eh*?



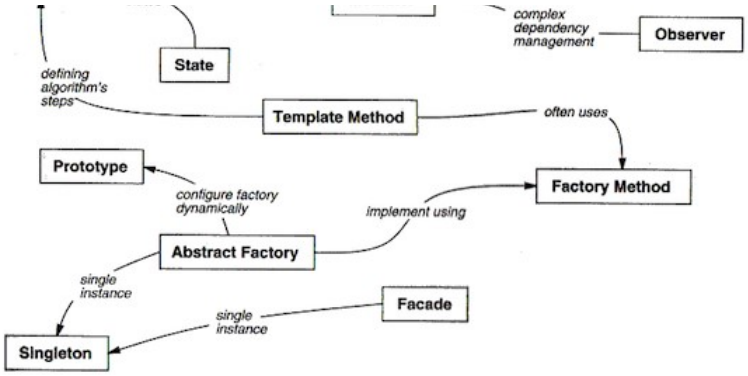


Figure 1.1: Design pattern relationships

Courtesy: [Design Patterns: Elements of Reusable Object-Oriented Software]

SOLID

We will only toy with the SOLID principles here, as they are essential for every software developer to know.

As [Uncle Bob] says: *"[They are not laws. They are not perfect truths. The are statements on the order of: An apple a day keeps the doctor away]"*.

What this means is that they are not some kind of 'magic' that lead to the Promised Land of milk, honey and great software, but nevertheless they are crucial contributors to robust and long lasting software.

In a nutshell, these principles revolve around two major concepts, which are the building blocks for successful enterprise applications: **coupling** is the degree to which one class knows about and interacts with another class and **cohesion** indicates the degree to which a class has a single purpose. In other words:

- *Coupling is all about how classes interact with each other, and*
- *Cohesion focuses on how a single class is designed.*

Single Responsibility Principle

A class should have one, and only one, reason to change.

This is self explanatory, but easier said than done—it is always tempting to add new behaviours into existing classes, but that's a recipe for disaster: each behaviour could be a reason to change in the future, so less behaviours result in less opportunities to introduce bugs during changes.

Open-Closed Principle

You should be able to extend a class' behaviour, without modifying it.

The classes you use should be open for extension but closed for modification. One way to achieve this is via inheritance i.e. create a sub-class so the original class is closed for modification, but custom code is added to the sub-class to introduce a new behaviour.

Liskov Substitution Principle

Derived classes must be substitutable for their base classes.

When extending the behaviour of a class A into a sub-class B you must ensure that you can still exchange A with B without breaking anything. This can be a bit catchy especially when combining this principle with the Open-Closed one.

Interface Segregation Principle

Make fine grained interfaces that are client specific.

Interfaces and classes must be as specialised as possible, so calling clients do not depend on methods they don't use. This goes hand in hand with the Single Responsibility principle.

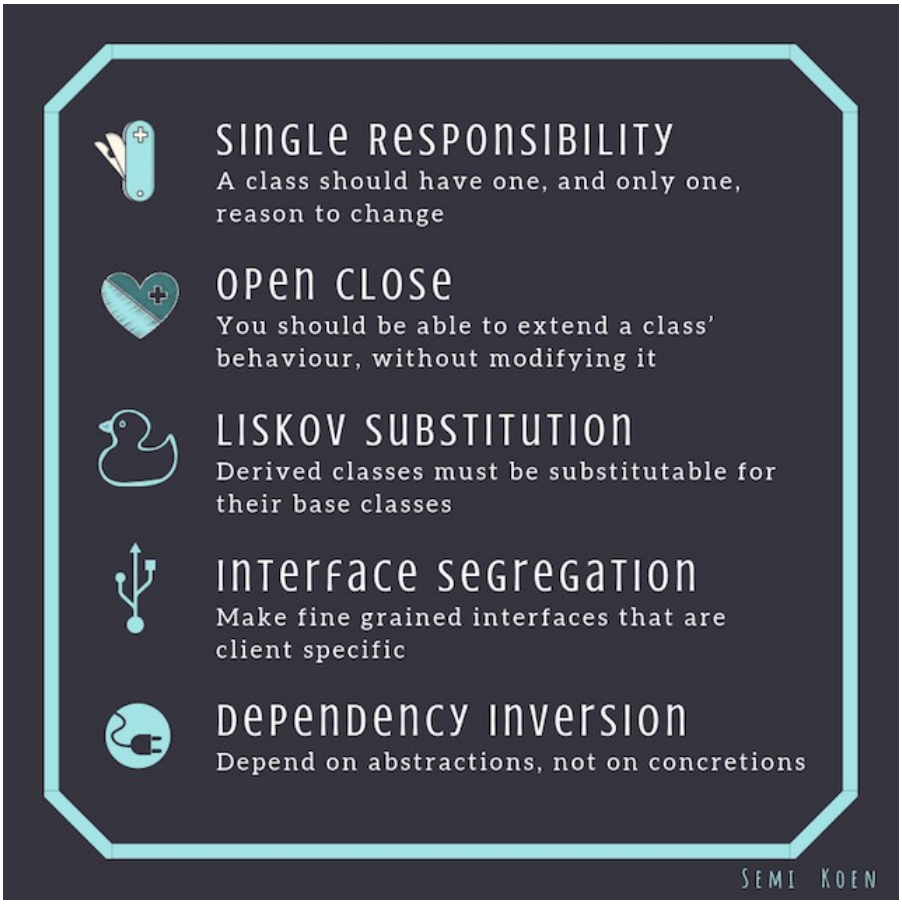
Dependency Inversion Principle

Depend on abstractions, not on concretions.

High level classes should not depend on low level ones. They should both depend on abstractions. Likewise, abstractions should not depend on details. Details should depend on abstractions.

Little Bonus

I have created this quick reference diagram. If you wonder where my inspiration for the little symbols on the left has come from, please take a look at: ''The SOLID Principles, Explained with Motivational Posters'' article—I love how the author has added a fun twist on the principles 🐤.



SOLID

Footnote

This is not an exhaustive list of all the software engineering concepts but it is the basis of what we are going to use in the next article. I hope it gives you a good flavour of the contributing factors to building scalable software. Making the application design **resilient to changes** is key to building a successful solution—when the design process is rushed there is a fine to pay at the end of the project when errors are uncovered.

Good design is obvious. Great design is transparent.

Thanks for reading! Part 2 is coming soon...

I regularly write about Technology & Data on Medium—if you would like to read my future posts then please 'Follow' me!