What is an Adapter?

An adapter is a class that transforms (adapts) an interface into another.

For example, an adapter implements an interface A and gets injected an interface B. When the adapter is instantiated it gets injected in its constructor an object that implements interface B. This adapter is then injected wherever interface A is needed and receives method requests that it transforms and proxies to the inner object that implements interface B.

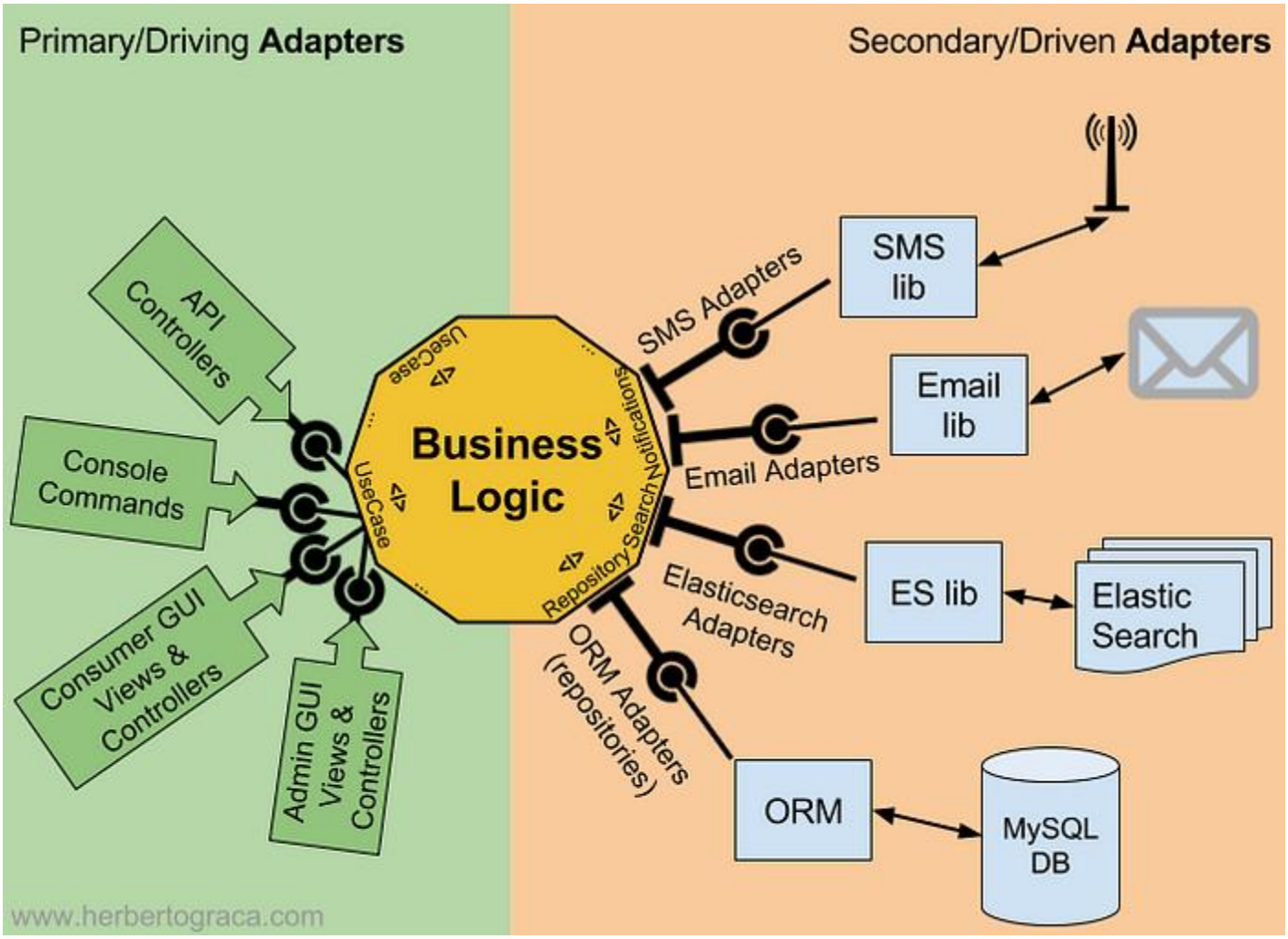
If I managed to confuse you, no worries, I give a more concrete example further below. ☐

Two different types of adapters

The adapters on the left side, representing the UI, are called the **Primary** or **Driving Adapters** because they are the ones to start some action on the application, while the adapters on the right side, representing the connections to the backend tools, are called the **Secondary** or **Driven Adapters** because they always react to an action of a primary adapter.

There is also a difference on how the ports/adapters are used:

- On the **left side**, the adapter depends on the port and gets injected a concrete implementation of the port, which contains the use case. On this side, **both the port and its concrete implementation (the use case) belong inside the application**;
- On the **right side**, the adapter **is** the concrete implementation of the port and is injected in our business logic although our business logic only knows about the interface. On this side, **the port belongs inside the application, but its concrete implementation belongs outside** and it wraps around some external tool.



What are the benefits?

Using this port/adapter design, with our application in the centre of the system, allows us to keep the application isolated from the implementation details like ephemeral technologies, tools and delivery mechanism

COMMAND:

The **command pattern** is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

Four terms always associated with the command pattern are *command*, *receiver*, *invoker* and *client*. A *command* object knows about *receiver* and invokes a method of the receiver. Values for parameters of the receiver method are stored in the command.

The receiver object to execute these methods is also stored in the command object by aggregation. The *receiver* then does the work when the `execute()` method in *command* is called. An *invoker* object knows how to execute a command, and optionally does bookkeeping about the command execution.

The invoker does not know anything about a concrete command, it knows only about the command *interface*. Invoker object(s), command objects and receiver objects are held by a *client* object, the *client* decides which receiver objects it assigns to the command objects, and which commands it assigns to the invoker.

The client decides which commands to execute at which points. To execute a command, it passes the command object to the invoker object.

Using command objects makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the class of the method or the method parameters.

Using an invoker object allows bookkeeping about command executions to be conveniently performed, as well as implementing different modes for commands, which are managed by the invoker object, without the need for the client to be aware of the existence of bookkeeping or modes.

System Design Strategy – Software Engineering

A good system design is to organize the program modules in such a way that are easy to develop and change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program.

Software Engineering is the process of designing, building, testing, and maintaining software. The goal of software engineering is to create software that is reliable, efficient, and easy to maintain. System design is a critical component of software engineering and involves making decisions about the architecture, components, modules, interfaces, and data for a software system.

System Design Strategy refers to the approach that is taken to design a software system. There are several strategies that can be used to design software systems, including the following:

1. Top-Down Design: This strategy starts with a high-level view of the system and gradually breaks it down into smaller, more manageable components.
2. Bottom-Up Design: This strategy starts with individual components and builds the system up, piece by piece.
3. Iterative Design: This strategy involves designing and implementing the system in stages, with each stage building on the results of the previous stage.
4. Incremental Design: This strategy involves designing and implementing a small part of the system at a time, adding more functionality with each iteration.
5. Agile Design: This strategy involves a flexible, iterative approach to design, where requirements and design evolve through collaboration between self-organizing and cross-functional teams.

The design of a system is essentially a blueprint or a plan for a solution for the system. The design process for software systems often has two levels. At the first level the focus is on deciding which modules are needed for the system, the specifications of these modules and how the modules should be interconnected. The design of a system is correct if a system built precisely according to the design satisfies the requirements of that system. The goal of the design process is not simply to produce a design for the system. Instead, the goal is to find the best possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.

The choice of system design strategy will depend on the particular requirements of the software system, the size and complexity of the system, and the development methodology being used. A well-designed system can simplify the development process, improve the quality of the software, and make the software easier to maintain.

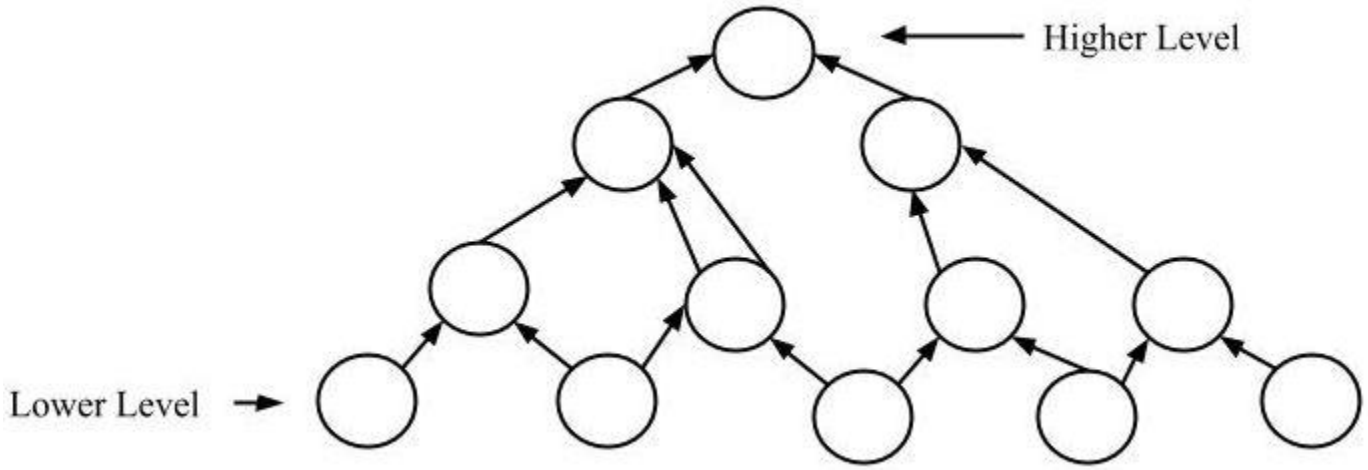
Importance of System Design Strategy:

1. If any pre-existing code needs to be understood, organized, and pieced together.
2. It is common for the project team to have to write some code and produce original programs that support the application logic of the system.

There are many strategies or techniques for performing system design. They are:

Bottom-up approach:

The design starts with the lowest level components and subsystems. By using these components, the next immediate higher-level components and subsystems are created or composed. The process is continued till all the components and subsystems are composed into a single component, which is considered as the complete system. The amount of abstraction grows high as the design moves to more high levels.
By using the basic information existing system, when a new system needs to be created, the bottom-up strategy suits the purpose.



Bottom-up approach

Advantages of Bottom-up approach:

- The economics can result when general solutions can be reused.
- It can be used to hide the low-level details of implementation and be merged with the top-down technique.

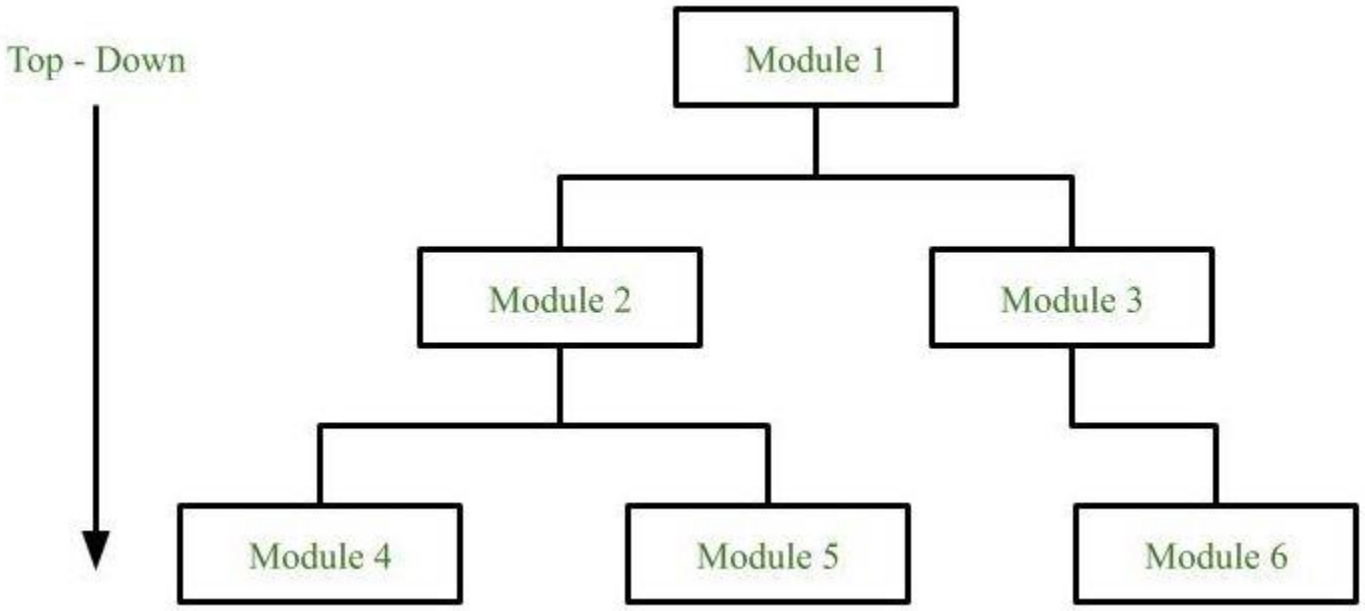
Disadvantages of Bottom-up approach:

- It is not so closely related to the structure of the problem.
- High-quality bottom-up solutions are very hard to construct.
- It leads to the proliferation of ‘potentially useful’ functions rather than the most appropriate ones.

Top-down approach:

Each system is divided into several subsystems and components. Each of the subsystems is further divided into a set of subsystems and components. This process of division facilitates forming a system hierarchy structure. The complete software system is considered a single entity and in relation to the characteristics, the system is split into sub-systems and components. The same is done with each of the sub-systems.
This process is continued until the lowest level of the system is reached. The design is started initially by defining the system as a whole and then keeps on adding definitions of the subsystems and components. When all the definitions are combined, it turns out to be a complete system.

For the solutions of the software that need to be developed from the ground level, a top-down design best suits the purpose.



Top-down approach

Advantages of Top-down approach:

- The main advantage of the top-down approach is that its strong focus on requirements helps to make a design responsive according to its requirements.

Disadvantages of Top-down approach:

- Project and system boundaries tend to be application specification-oriented. Thus, it is more likely that the advantages of component reuse will be missed.
- The system is likely to miss, the benefits of a well-structured, simple architecture.
- **Hybrid Design:**
  It is a combination of both top-down and bottom-up design strategies. In this, we can reuse the modules.

Advantages of using a System Design Strategy:

1. Improved quality: A well-designed system can improve the overall quality of the software, as it provides a clear and organized structure for the software.
2. Ease of maintenance: A well-designed system can make it easier to maintain and update the software, as the design provides a clear and organized structure for the software.
3. Improved efficiency: A well-designed system can make the software more efficient, as it provides a clear and organized structure for the software that reduces the complexity of the code.
4. Better communication: A well-designed system can improve communication between stakeholders, as it provides a clear and organized structure for the software that makes it easier for stakeholders to understand and agree on the design of the software.

5. Faster development: A well-designed system can speed up the development process, as it provides a clear and organized structure for the software that makes it easier for developers to understand the requirements and implement the software.

Disadvantages of using a System Design Strategy:

1. Time-consuming: Designing a system can be time-consuming, especially for large and complex systems, as it requires a significant amount of documentation and analysis.
2. Inflexibility: Once a system has been designed, it can be difficult to make changes to the design, as the process is often highly structured and documentation-intensive.

PUBLISH –SUBSCRIBE

The publisher-subscriber (pub-sub) model is a widely used architectural pattern. We can use it in software development to enable communication between different components in a system.

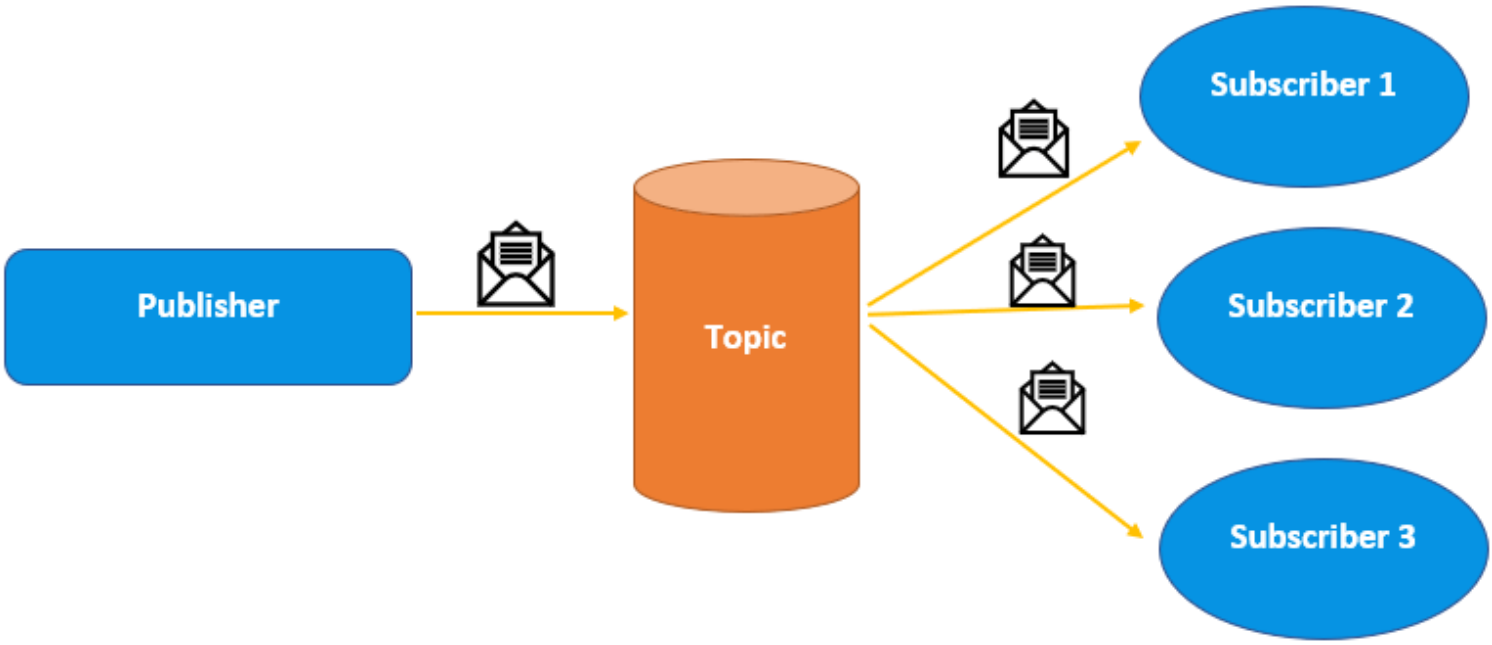
In particular, it is often used in distributed systems, where different parts of the system need to interact with each other but don't want to be tightly coupled.

In this tutorial, we'll explore the pub-sub model, how it works, and some common use cases for this architectural pattern.

2. Pub-Sub Model: Overview

The pub-sub model involves publishers and subscribers, making it a messaging pattern. Specifically, the publishers are responsible for sending messages to the system, while subscribers are responsible for receiving those messages.

Mainly, the pub-sub model is based on decoupling components in a system, which means that components can interact without being tightly coupled.



3. How the Pub-Sub Model Works

In this section, we'll discuss how this model works, including sending messages, checking for subscribers, receiving messages, registering for topics, decoupling publishers and subscribers, and additional features the message broker implements to enhance message delivery.

3.1. Sending Messages

A publisher sends a message to the message broker with a specific topic, which is a string that identifies the content of the message.

3.2. Checking for Subscribers

The message broker receives the message and checks the topic to see if any subscribers have expressed interest in receiving messages on that topic. Furthermore, if subscribers are interested in the topic, the message broker sends the message to all subscribers who have registered interest in that topic.

3.3. Receiving Messages

Subscribers receive the message from the message broker. Then, it can process the message as needed. However, the message is discarded if no subscribers are interested in the topic.

3.4. Registering for Topics

To receive messages on specific topics, subscribers can register interest in one or more topics with the message broker. Additionally, this feature enables subscribers to receive messages on topics they are interested in.

3.5. Decoupling Publishers and Subscribers

Publishers and subscribers do not need to know about each other's existence since they interact only through the message broker, which acts as an intermediary.

3.6. Additional Features

The message broker can also implement additional features such as filtering messages based on content, ensuring message delivery, and providing message ordering guarantees. These features enhance the reliability and efficiency of message delivery.

By decoupling publishers and subscribers, the pub-sub model allows them to interact through a message broker, which helps to reduce tight coupling between components in a system.

This makes it an ideal messaging pattern for use in distributed systems, where different parts of the system must interact without being tightly coupled.

4. Advantages and Disadvantages of the Pub-Sub Model

The pub-sub model has several benefits. The following table summarizes its main advantages:

The pub-sub model has several benefits. The following table summarizes its main advantages:

Advantage	Description
Scalability	The decoupled nature of the pub-sub model makes it highly scalable. The model can handle a large number of publishers and subscribers without affecting the performance
Reliability	A message broker ensures the reliable delivery of messages to interested subscribers, even if some subscribers are offline or disconnected
Flexibility	The pub-sub model offers high flexibility by enabling the addition or removal of publishers and subscribers without affecting the overall system
Loose coupling	The decoupled nature of the pub-sub model ensures that publishers and subscribers are loosely coupled, which allows them to evolve independently without affecting each other

However, the pub-sub model also has some drawbacks. The following table shows its main drawbacks:

Disadvantage	Description
Increased complexity	The use of a message broker adds complexity to the system, making it more difficult to implement and maintain
Higher latency	The use of a message broker can introduce additional latency into the system, which may be unacceptable for some real-time applications
Single point of failure	The message broker represents a single point of failure for the system, which may result in service disruption if it fails
Loose coupling	The decoupled nature of the pub-sub model ensures that publishers and subscribers are loosely coupled, allowing them to evolve independently without affecting each other

5. Use Cases for the Pub-Sub Model

In this section, we'll explore some use cases of this model, including **real-time updates in online games**, **smart homes with IoT**, and **data distribution in data analytics.**

5.1. Real-time Updates in Online Games

One of the use cases for the pub-sub model is online gaming, where publishers can send real-time updates on player positions, score changes, and game events to all subscribers.

Overall, this enhances the gaming experience and ensures all players receive the same updates simultaneously.

5.2. Smart Homes with IoT

The pub-sub model is also used in smart homes to send messages from sensors to actuators. For example, we can use this model to turn on the lights when someone enters a room.