Introduction

Introduction

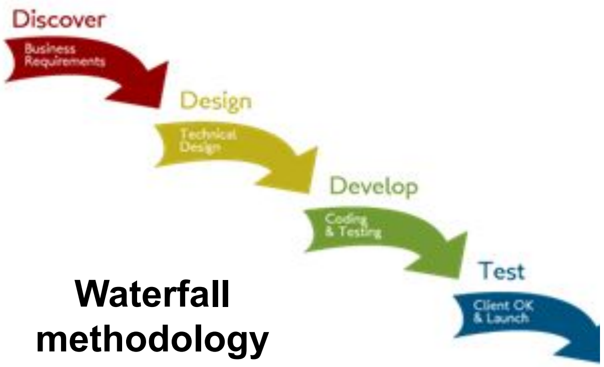
- Software Development Methodologies
- Software Architecture and Design
- The 4+1 View Model of Software Architecture
- Role of Design Patterns in Software Design
- Architectural styles/patterns
- Architectural patterns for Logical View
- Architectural Patterns for Implementation View
- Software Architecture and Design in Traditional and Agile Methodologies
- References

Software Development Methodologies

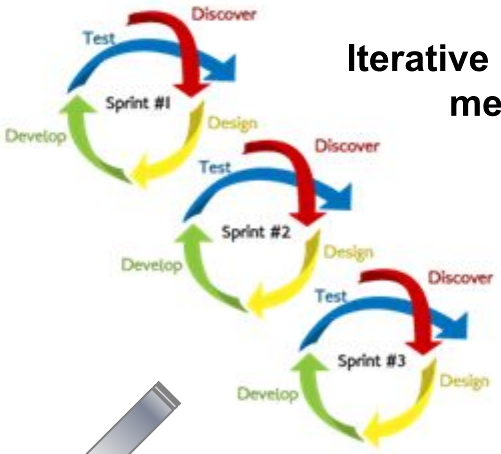
- **Software development methodologies** are a specific collection of principles and/or practices applied to develop a software system.

- Methodologies may include the pre-definition of specific deliverables and artifacts that are created and completed by a project team to develop or maintain an application.

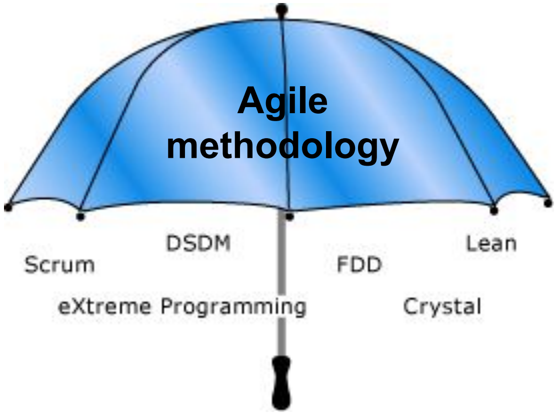
Software Development Methodologies



Waterfall methodology



Iterative and incremental methodology



Agile methodology

Adaptative vs predictive
Iterative vs waterfall
Code vs documentation

*Images extracted from: http://www.commonplaces.com/blog/agile-v-waterfall-how-to-approach-your-web-development-project/,

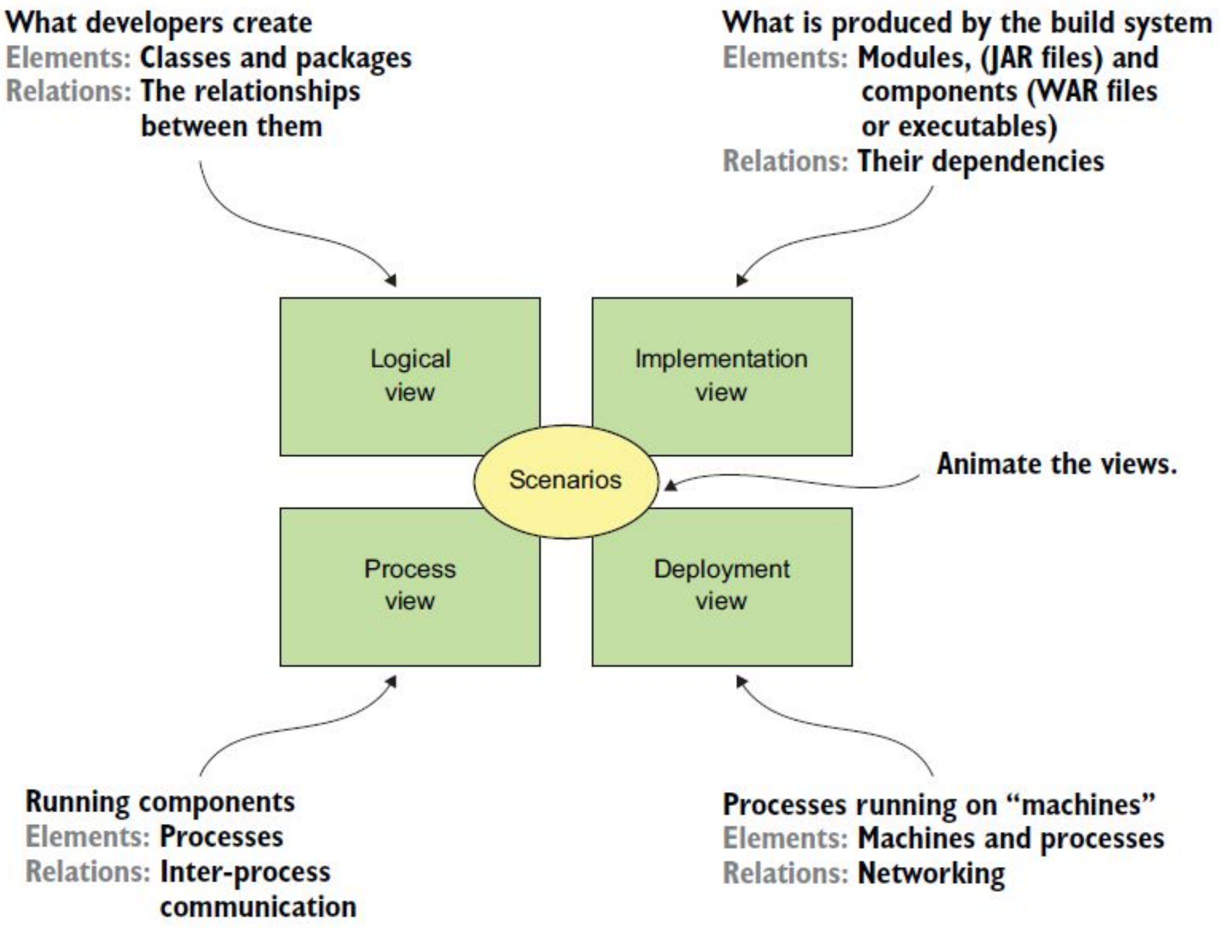
Software Design and Architecture

- **Software design** is the activity of applying different techniques and principles to define a system up to the level of detail needed to physically build. One output of the software design is the software architecture.

- **Software architecture** of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

- It is the decomposition into parts and relationships between those parts that determine the application non-functional requirements.

The 4+1 View Model of Software Architecture



What developers create
Elements: Classes and packages
Relations: The relationships between them

What is produced by the build system
Elements: Modules, (JAR files) and components (WAR files or executables)
Relations: Their dependencies

Animate the views.

Running components
Elements: Processes
Relations: Inter-process communication

Processes running on “machines”
Elements: Machines and processes
Relations: Networking

*Image extracted from: Microservices Patterns with examples in Java. Chris Richardson. Manning Publications Co.

Role of Design Patterns in Software Design

- **Design patterns** are general reusable solutions to commonly occurring problems within a given context in software design.

- Design patterns may be used in traditional and agile methodologies.

- Two types of patterns used at the design phase:
  - Architectural patterns
  - Design patterns

Role of Architectural styles/patterns in Software Design

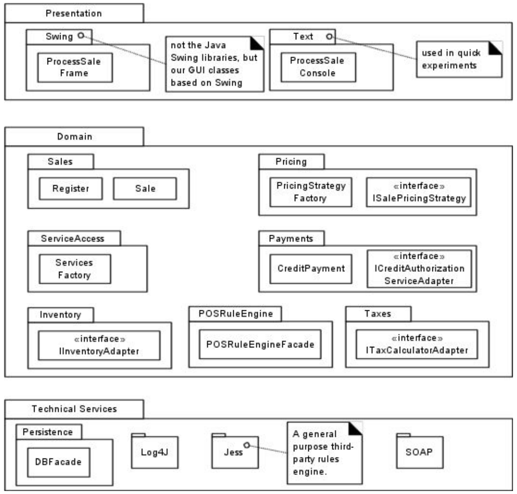
- **An architectural style/pattern** defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined (Chris Richardson (2019). *Microservices Patterns with examples in Java.* Manning Publications Co).

- Architectural patterns organize the four views of the 4+1 View Model.

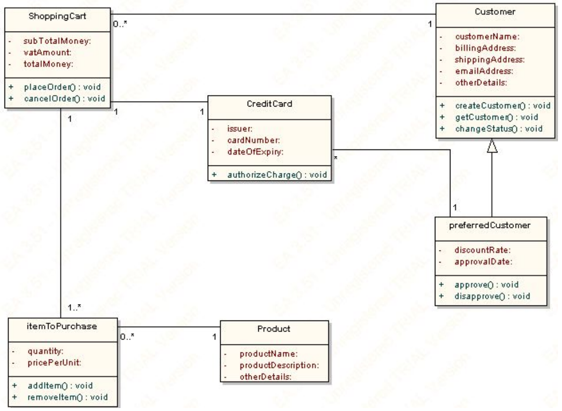
Role of Design Patterns in Software Design

- **A design pattern** provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly recurring structure of communicating components that solves a general design problem within a particular context. (F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stad. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley)
- Some design patterns
  - State
  - Expert
  - Controller
  - …

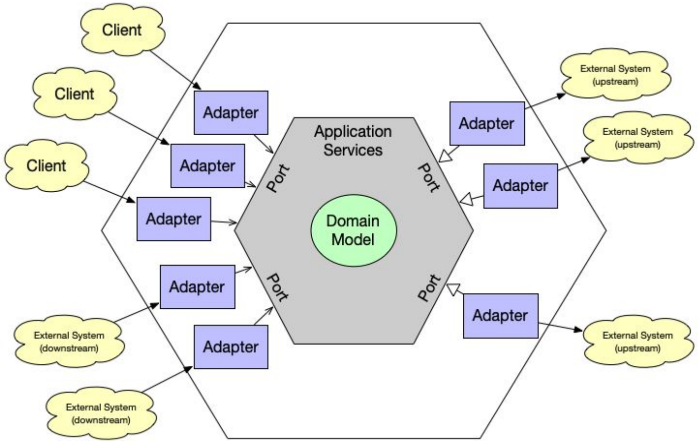
Examples of Architectural Patterns for Logical View


Presentation
Swing
ProcessSale Frame
not the Java Swing libraries, but our GUI classes based on Swing
Text
ProcessSale Console
used in quick experiments
Domain
Sales
Register
Sale
Pricing
PricingStrategy Factory
«interface» ISalePricingStrategy
ServiceAccess
Services Factory
Payments
CreditPayment
«interface» ICreditAuthorization ServiceAdapter
Inventory
«interface» IInventoryAdapter
POSRuleEngine
POSRuleEngineFacade
Taxes
«interface» ITaxCalculatorAdapter
Technical Services
Persistence
DBFacade
Log4J
Jess
A general purpose third-party rules engine.
SOAP

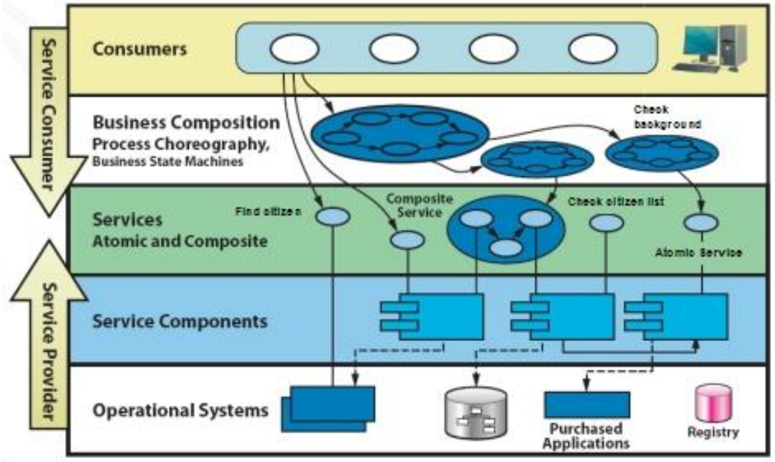
Layered



Object-Oriented Architecture

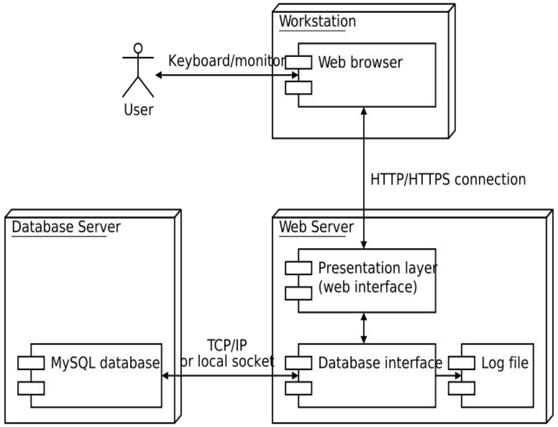

Client
Client
Client
Adapter
Adapter
Adapter
Adapter
Adapter
Application Services
Domain Model
Port
Port
Port
Port
Adapter
Adapter
Adapter
External System (upstream)
External System (upstream)
External System (upstream)
External System (downstream)
External System (downstream)

Hexagonal Architecture

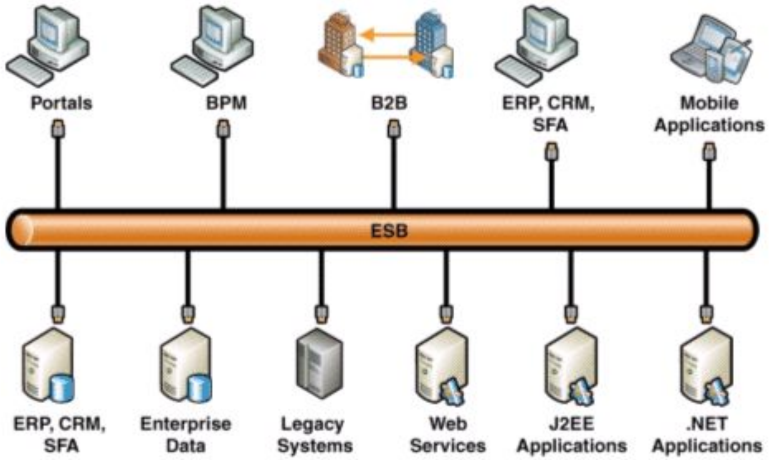


Service-Oriented Architecture

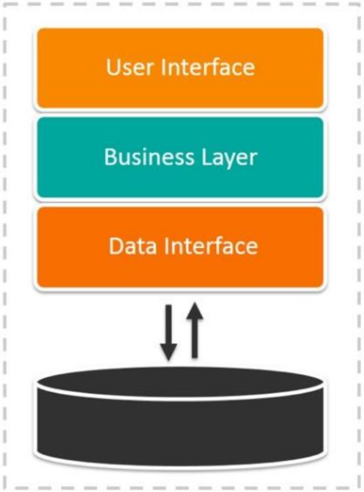
Examples of Architectural Patterns for Implementation View



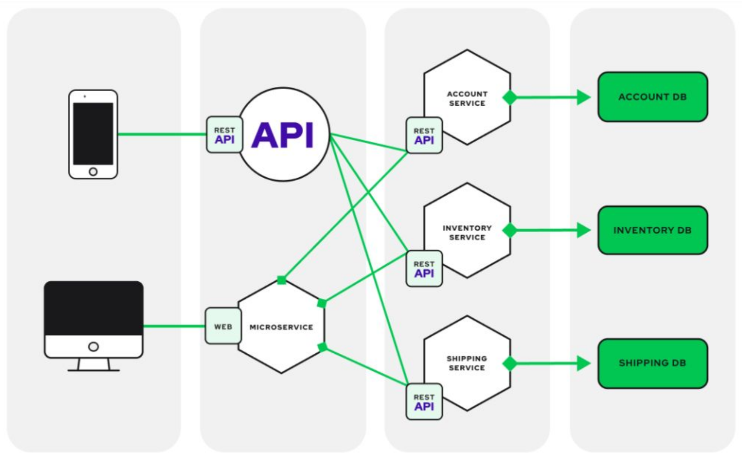
N-Tier/3-Tier Architecture



Message Bus



Monolithic Architecture



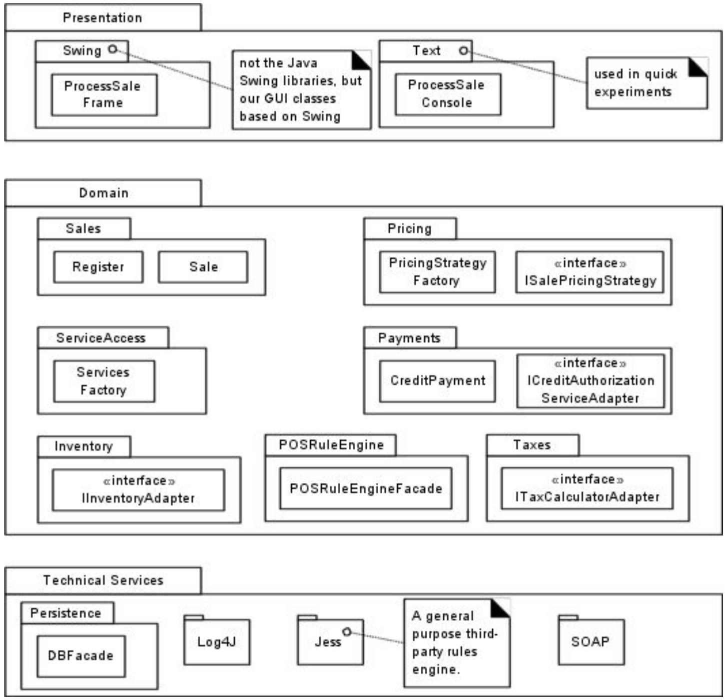
Microservice Architecture

Architectural patterns for Logical View

Layered Architecture

- **Layered architecture** organizes software elements into layers. Each layer has a well-defined set of responsibilities. A layer can only depend on either the layer immediately below it (if strict layering) or any of the layers below it.

Logical View


Presentation
Swing
ProcessSale Frame
not the Java Swing libraries, but our GUI classes based on Swing
Text
ProcessSale Console
used in quick experiments
Domain
Sales
Register
Sale
Pricing
PricingStrategy Factory
«interface» ISalePricingStrategy
ServiceAccess
Services Factory
Payments
CreditPayment
«interface» ICreditAuthorization ServiceAdapter
Inventory
«interface» IInventoryAdapter
POSRuleEngine
POSRuleEngineFacade
Taxes
«interface» ITaxCalculatorAdapter
Technical Services
Persistence
DBFacade
Log4J
Jess
A general purpose third-party rules engine.
SOAP

Architectural patterns for Logical View

Layered Architecture

- **Layered architecture** has several benefits:
  - Simplicity: Easy to understand and implement in any project.
  - Changeability inside the layer: If any changes are to be made, it is easy to find the object within the layer.
  - Easy discovery: Different tasks are assigned to layers so when a task has to be identified, it is easy to figure it out using the layering structure.

Architectural patterns for Logical View

Layered Architecture

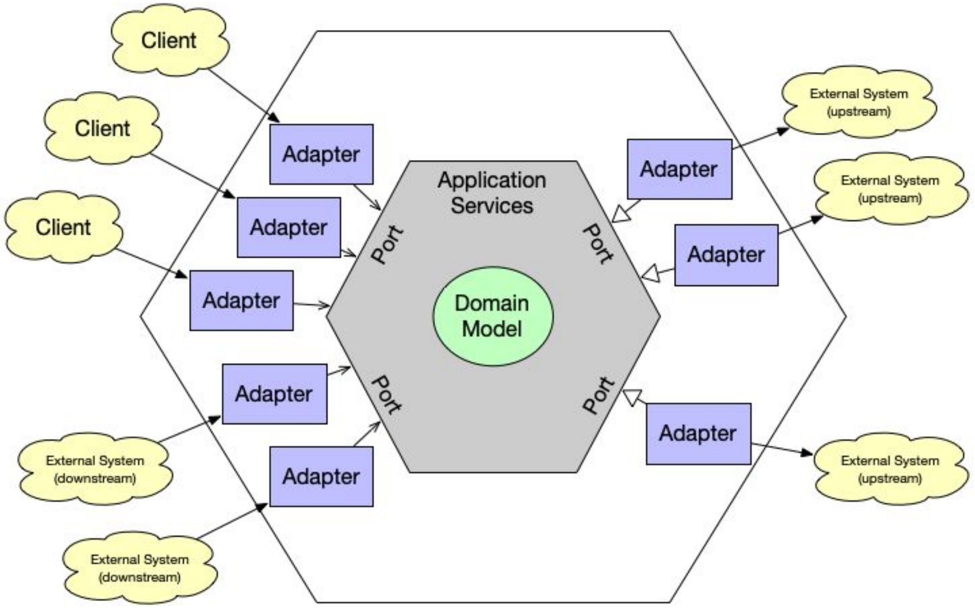
- **Layered architecture** has several drawbacks:
  - Single presentation layer: It does not represent the fact that an application is likely to be invoked by more than just a single system.
  - Single persistence layer: It does not represent the fact that an application is likely to interact with more than just a single database.
  - Defines the business logic layer as depending on the persistence layer : In theory, this dependency prevents you from testing the business logic without the database.

Architectural patterns for Logical View

Hexagonal Architecture

- **Hexagonal architecture** organizes the logical view and defines three types of components:
  - Business logic / Domain Model
  - Ports (edges of the hexagon)
  - Adapters (external components)

Logical View



Client
Client
Client
Adapter
Adapter
Adapter
Adapter
Adapter
Application Services
Port
Port
Port
Port
Domain Model
Adapter
Adapter
Adapter
External System (upstream)
External System (upstream)
External System (upstream)
External System (downstream)
External System (downstream)

*Image extracted from: https://vaadin.com/blog/ddd-part-3-domain-driven-design-and-the-hexagonal-architecture.

Architectural patterns for Logical View

Hexagonal Architecture

- **Ports** are interfaces that the application offers to the outside world for allowing actors interact with the application avoiding accessing the inside of the hexagon.

- **Inbound port** is an API exposed by the business logic, which enables it to be invoked by external applications. For example a service interface, which defines a service's public methods.

- **Outbound port** is an interface for a functionality, needed by the application for implementing the business logic. An outbound port would be like a required interface (for example, for accessing to a database).

Architectural patterns for Logical View

Hexagonal Architecture

- **Adapters** are software components that allows a technology to interact with a port. Given a port, there may be an adapter for each desired technology that we want to use. Adapters are outside the application.
- An **inbound adapter** uses an inbound port interface, converting a specific technology request into a technology agnostic request to a inbound port. For example, a REST API controller to convert REST API requests.
- An **outbound adapter** implements an outbound port interface, converting the technology agnostic methods of the port into specific technology methods. For example, a SQL adapter to implement an outbound port for persisting data by accessing a SQL database.

Architectural patterns for Logical View

Hexagonal Architecture

- **Hexagonal architecture** has several benefits:
  - It decouples the business logic from the presentation and data access logic in the adapters.
  - It is much easier to test the business logic in isolation.
  - The business logic can be invoked via multiple adapters.
  - The business logic can also invoke multiple adapters, each one of which invokes a different external system.

Architectural patterns for Logical View

Hexagonal Architecture

- **Example**: FTGO (Food To GO) is one of the leading online food delivery companies in the United States.
- Consumers use the FTGO website or mobile application to place food orders at local restaurants.
- FTGO coordinates a network of couriers who deliver the orders.
- It’s also responsible for paying couriers and restaurants.
- Restaurants use the FTGO website to edit their menus and manage orders.
- The application uses various web services, including Stripe for payments,Twilio for messaging, and Amazon Simple Email Service (SES) for email.

Architectural patterns for Logical View

Hexagonal Architecture

Logical View

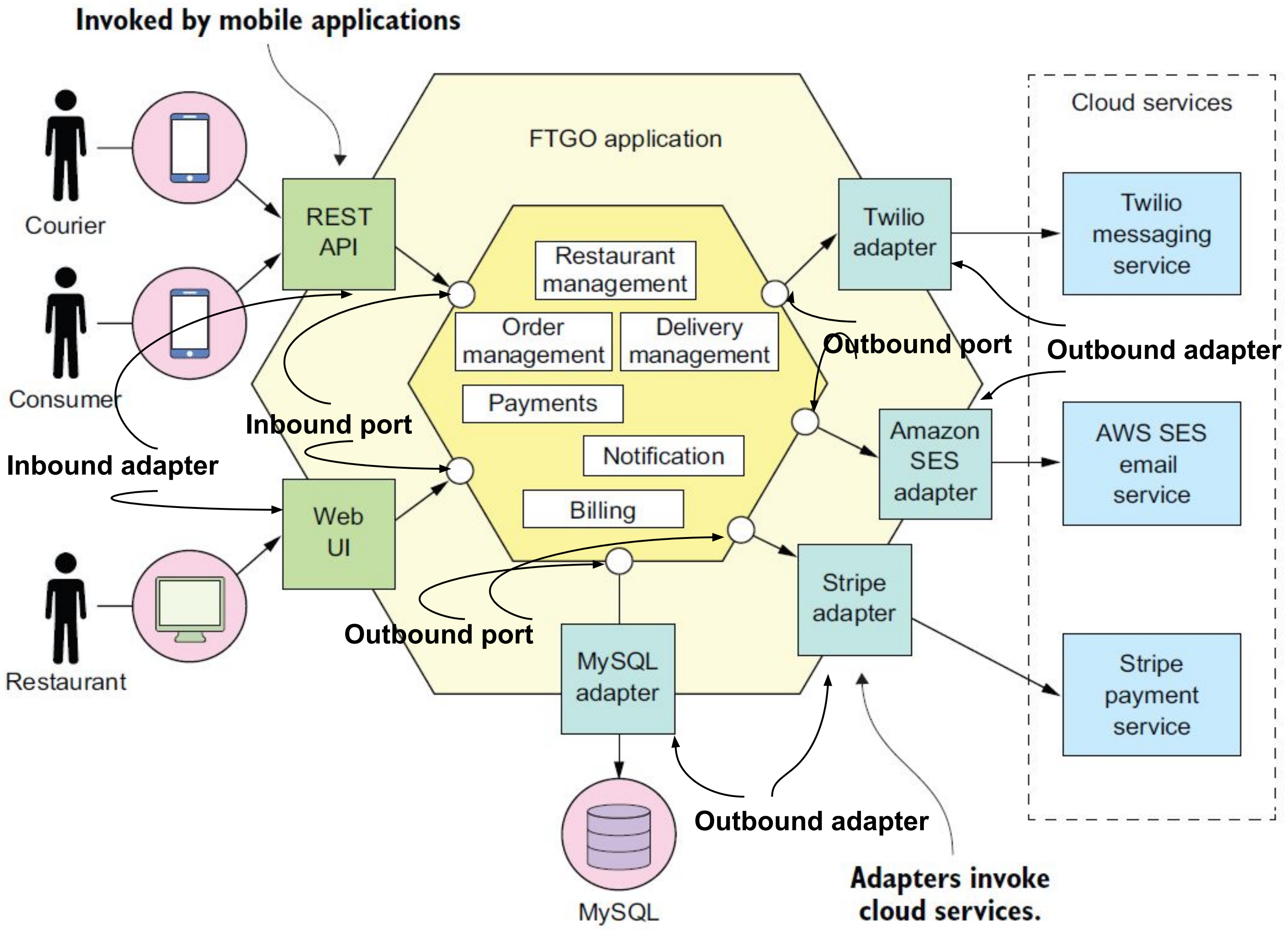


*Image extracted from: Microservices Patterns with examples in Java. Chris Richardson. Manning Publications Co.

Architectural patterns for Implementation View

Monolithic architecture

- **Monolithic architecture** is an architectural style that structures the implementation view as a single component: a single executable or WAR file.

- A monolithic application can, for example, have a logical view that is organized along the lines of a hexagonal architecture.

Architectural patterns for Implementation View

Monolithic architecture

- **Monolithic architecture** has several benefits:
  - Simple to develop: IDEs and other developer tools are focused on building a single application.
  - Easy to make radical changes to the application: You can change the code and the database schema, build, and deploy.
  - Straightforward to test: The developers wrote end-to-end tests that launched the application, invoked the REST API.
  - Straightforward to deploy: All a developer had to do was copy the WAR file to a server that had Tomcat installed.
  - Easy to scale: multiple instances of the application may be run behind a load balancer.

Architectural patterns for Implementation View

Monolithic architecture

- **Monolithic architecture** has several drawbacks:
  - Complex when the application is too large: Fixing bugs and adding new functionalities is difficult.
  - Development is slow: The edit-build-run-test loop takes a long time, which badly impacts productivity.
  - Path from commit to deployment is too long and arduous: Deploying changes into production is a long and painful process.
  - Scaling is difficult: Different application modules have conflicting resource requirements.
  - Delivering a reliable monolith is challenging: testing the application thoroughly is difficult, due to its large size.
  - Locked into increasingly obsolete technology stack: It would be extremely expensive and risky to rewrite the entire monolithic application so that it would use a new and presumably better technology.

Architectural patterns for Implementation View

Monolithic architecture

- **Example:** Like many other aging enterprise applications, the FTGO application is a monolith, consisting of a single Java Web Application Archive (WAR) file.
- Over the years, it has become a large, complex application.
- The pace of software delivery has slowed. To make matters worse, the FTGO application has been written using some increasingly obsolete frameworks.
- The FTGO application is exhibiting all the symptoms of monolithic hell.

Architectural patterns for Implementation View

Microservice architecture

- **Microservice architecture** is an architectural pattern that structures the implementation view as a set of multiple components: services and the connectors (communication protocols that enable those services to collaborate).

- A **service** is a standalone, independently deployable software component (executables or WAR files) that implements some useful functionality. A service has an API that provides its clients access to its functionality.

- Each service has its own logical view architecture, which is typically a hexagonal architecture.

Architectural patterns for Implementation View

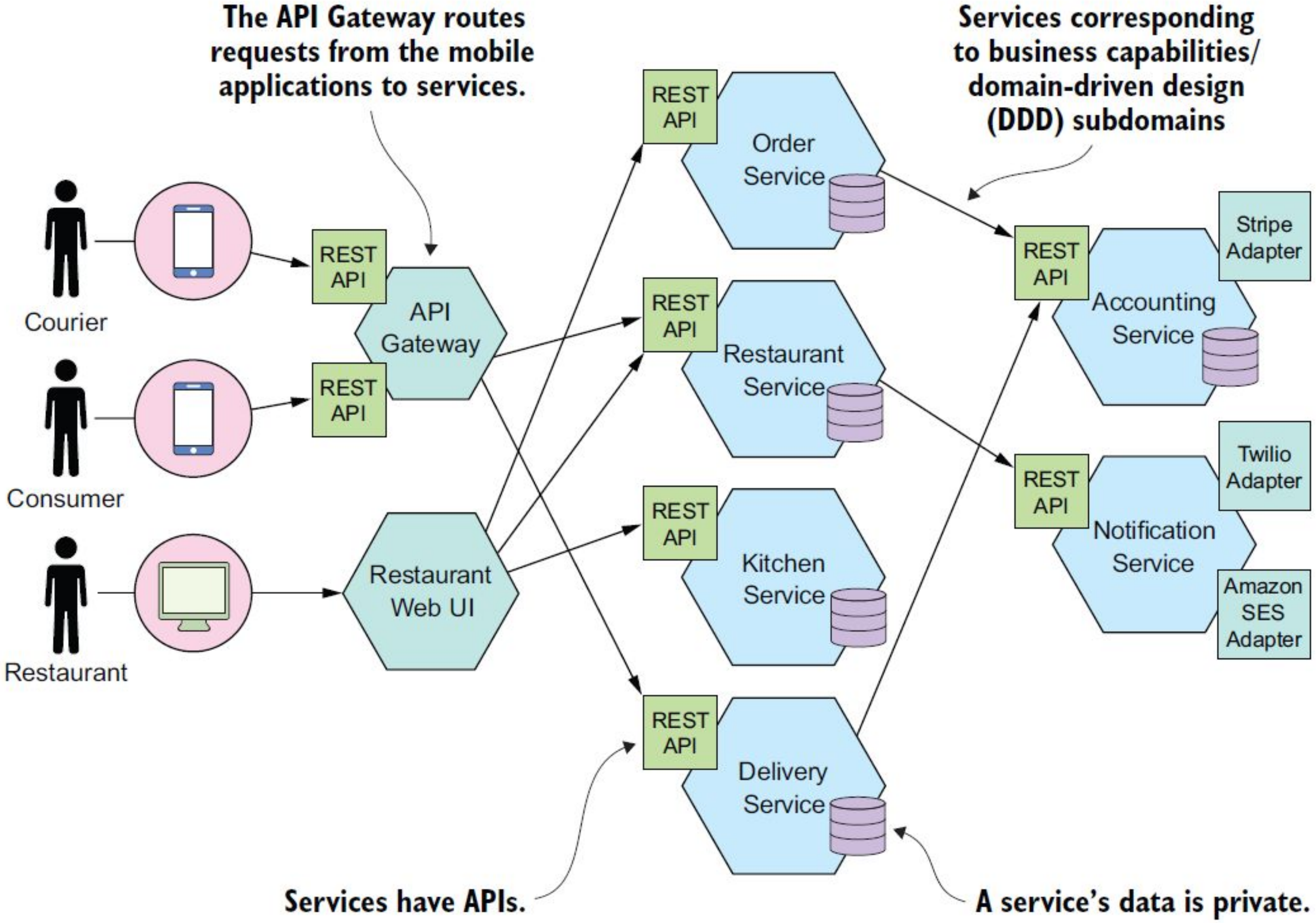
Microservice architecture

- **Microservice architecture** has several benefits:
  - It enables the continuous delivery and deployment of large, complex applications.
  - Services are small and easily maintained.
  - Services are independently deployable.
  - Services are independently scalable.
  - The microservice architecture enables teams to be autonomous.
  - It allows easy experimenting and adoption of new technologies.
  - It has better fault isolation.

Architectural patterns for Implementation View

Microservice architecture

- **Example:** FTGO application's microservice architecture.



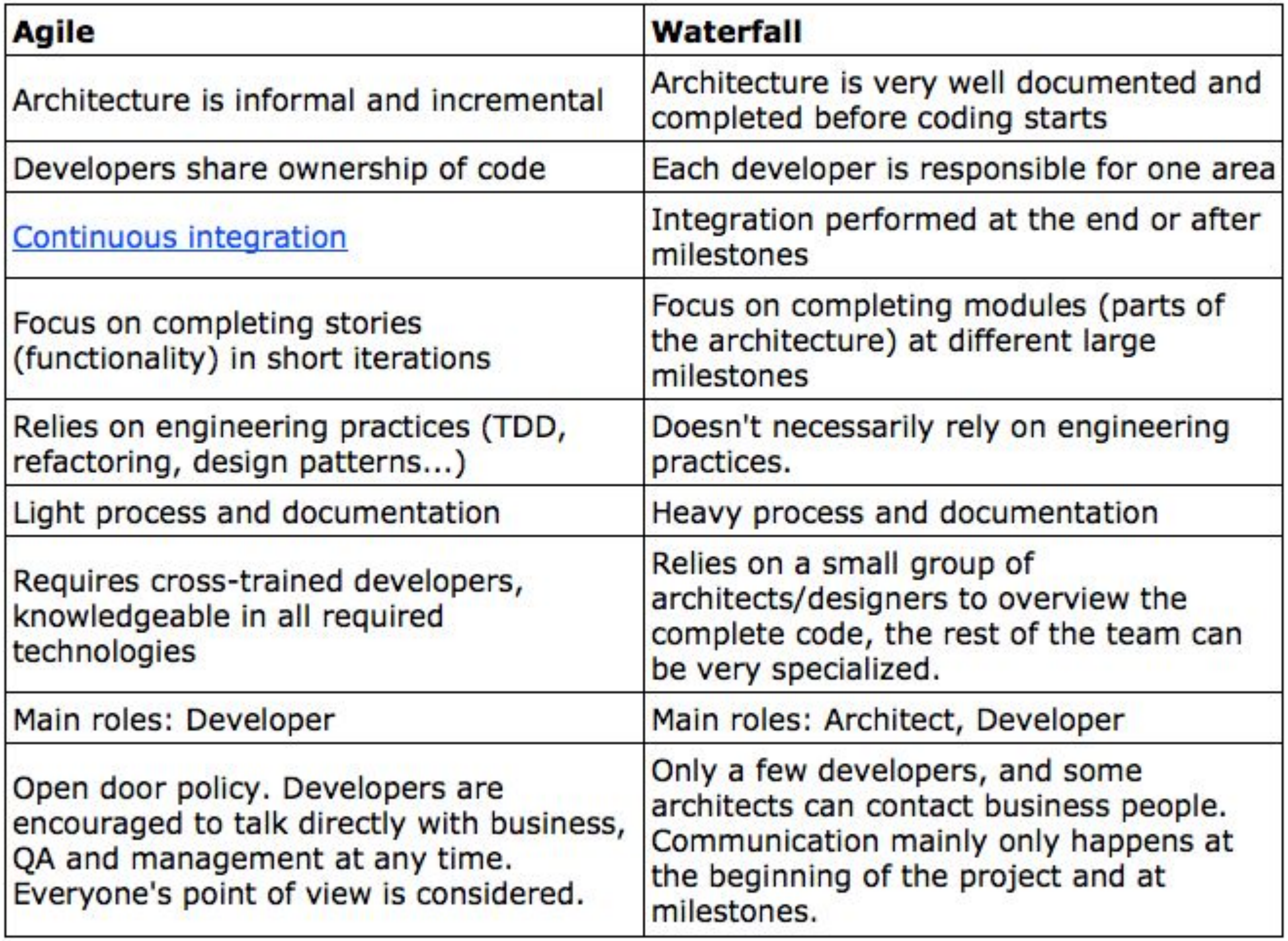
The API Gateway routes requests from the mobile applications to services.

Services corresponding to business capabilities/ domain-driven design (DDD) subdomains

Services have APIs.

A service's data is private.

Software Architecture and Design in Traditional and Agile Methodologies



Agile	Waterfall
Architecture is informal and incremental	Architecture is very well documented and completed before coding starts
Developers share ownership of code	Each developer is responsible for one area
Continuous integration	Integration performed at the end or after milestones
Focus on completing stories (functionality) in short iterations	Focus on completing modules (parts of the architecture) at different large milestones
Relies on engineering practices (TDD, refactoring, design patterns...)	Doesn't necessarily rely on engineering practices.
Light process and documentation	Heavy process and documentation
Requires cross-trained developers, knowledgeable in all required technologies	Relies on a small group of architects/designers to overview the complete code, the rest of the team can be very specialized.
Main roles: Developer	Main roles: Architect, Developer
Open door policy. Developers are encouraged to talk directly with business, QA and management at any time. Everyone's point of view is considered.	Only a few developers, and some architects can contact business people. Communication mainly only happens at the beginning of the project and at milestones.

*Table extracted from: https://dzone.com/articles/waterfall-vs-agile-development-business

References

- *Ingeniería del software. Un enfoque práctico*
  R.G. Pressman
  McGraw Hill, 2010 (Séptima edición), cap. 8, 9 and 10

- *Enginyeria del software: Especificació*
  D. Costal, X. Franch, M.R. Sancho, E. Teniente
  Edicions UPC, 2004

- *Applying UML and Patterns*
  C. Larman
  Prentice Hall, 2005 (3rd edition), ch. 33, 34 and 39

- Microsoft Application Architecture Guide (2nd edition)
  Microsoft
  http://msdn.microsoft.com/en-us/library/ff650706.aspx, ch. 1,2 and 3

- Is Design Dead?
  M. Fowler
  http://martinfowler.com/articles/designDead.html