Presents

Microservices

Design Concepts

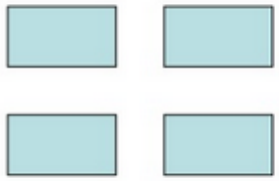
- Modularity
- Coupling
- Cohesion
- Suppleness
- Dependency Injection (DI)
- Dependency Inversion Principle (DIP)
- Inversion of Control (IoC)

Modularity

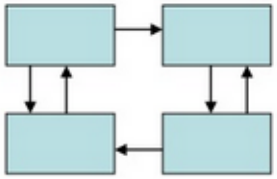
- Closely tied to abstraction
  - ✓ Allows the use of independent model implementation
  - ✓ Internal implementation is hidden
  - ✓ Communication is through stable interfaces
- Modules allow for reuse of components
  - ✓ Supports higher level architectural structuring of programs
- Enhances design clarity, simplifies implementation
  - ✓ Reduces cost and effort of testing, debugging and maintenance
- Modularity is not arbitrary
  - ✓ The choice of components is based on the domain

Coupling

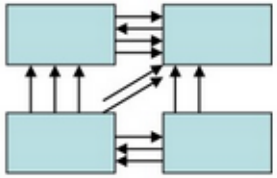
- Degree of dependence among components
  - ✓ High coupling makes modifying different parts of the system difficult
  - ✓ Modifying a highly coupled component affects all the other connected components
  - ✓ Often results in brittle and unusable systems



No dependencies



Loosely coupled-some dependencies



Highly coupled-many dependencies

LearnQuest

Cohesion

- The degree to which:
  - ✓ All elements of a component are directed towards a single task; and,
  - ✓ When all elements directed towards a specific task are contained in a single component
  - ✓ High cohesion is good
  - ✓ Highly cohesive components tend to have low coupling
- Often expressed as the single responsibility principle
  - ✓ Highly cohesive components specialize in implementing a single responsibility
  - ✓ There is only one component that implements that responsibility
  - ✓ Achieving cohesiveness depends on how we modularize an architecture

Suppleness

- Modular systems interact through interfaces
- Suppleness refers to the design of the interfaces
  - ✓ Follow good interface design rules, e.g., Open-Close principle
- Suppleness allows for future expansion of the architecture
  - ✓ Without breaking existing inter-module communications
- For example, anti-corruption layer
  - ✓ Links two modules with different internal representations with a “translation” from one representation to the other of some shared data item
  - ✓ E.g. Data seamlessly shared between databases that use different units (pounds versus kilograms for example)

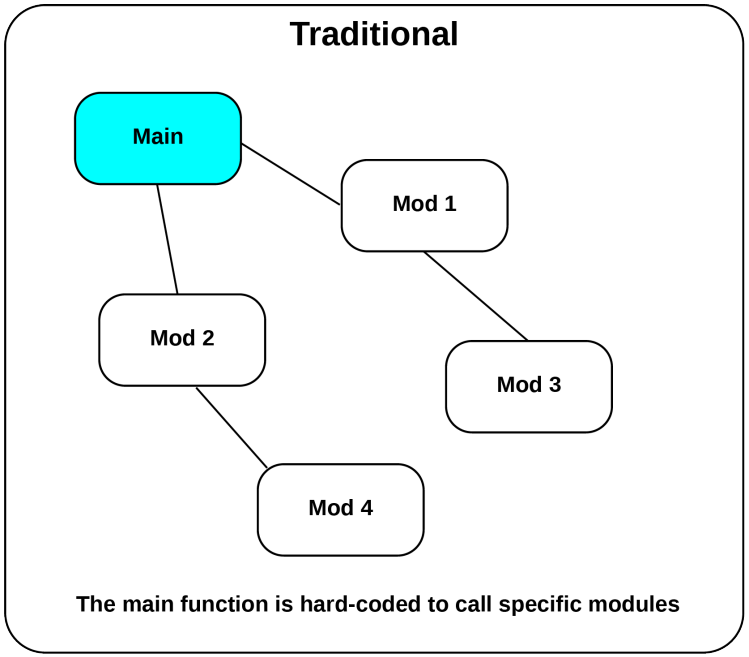
Dependency Injection

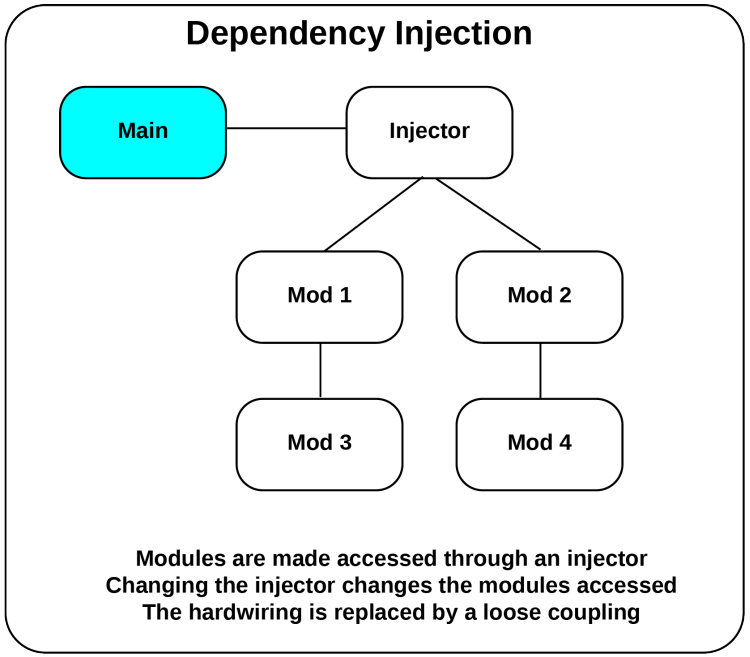
- Used when a client module depends on a service module
  - ✓ E.g., an invoicing module depends on a catalog module for pricing
- The service module provides an interface
  - ✓ Describes how to access the services of the service module
  - ✓ This interface is independent of the actual implementation of the service
  - ✓ The interface is “injected” or provided to the client module dynamically
  - ✓ At build time, the appropriate implementation of the service interface is used

Dependency Injection

- Requires that
  - ✓ The injected interface remain stable even if the implementation changes
  - ✓ The injected interface can be easily swapped out in the client code for a different interface

Dependency Injection



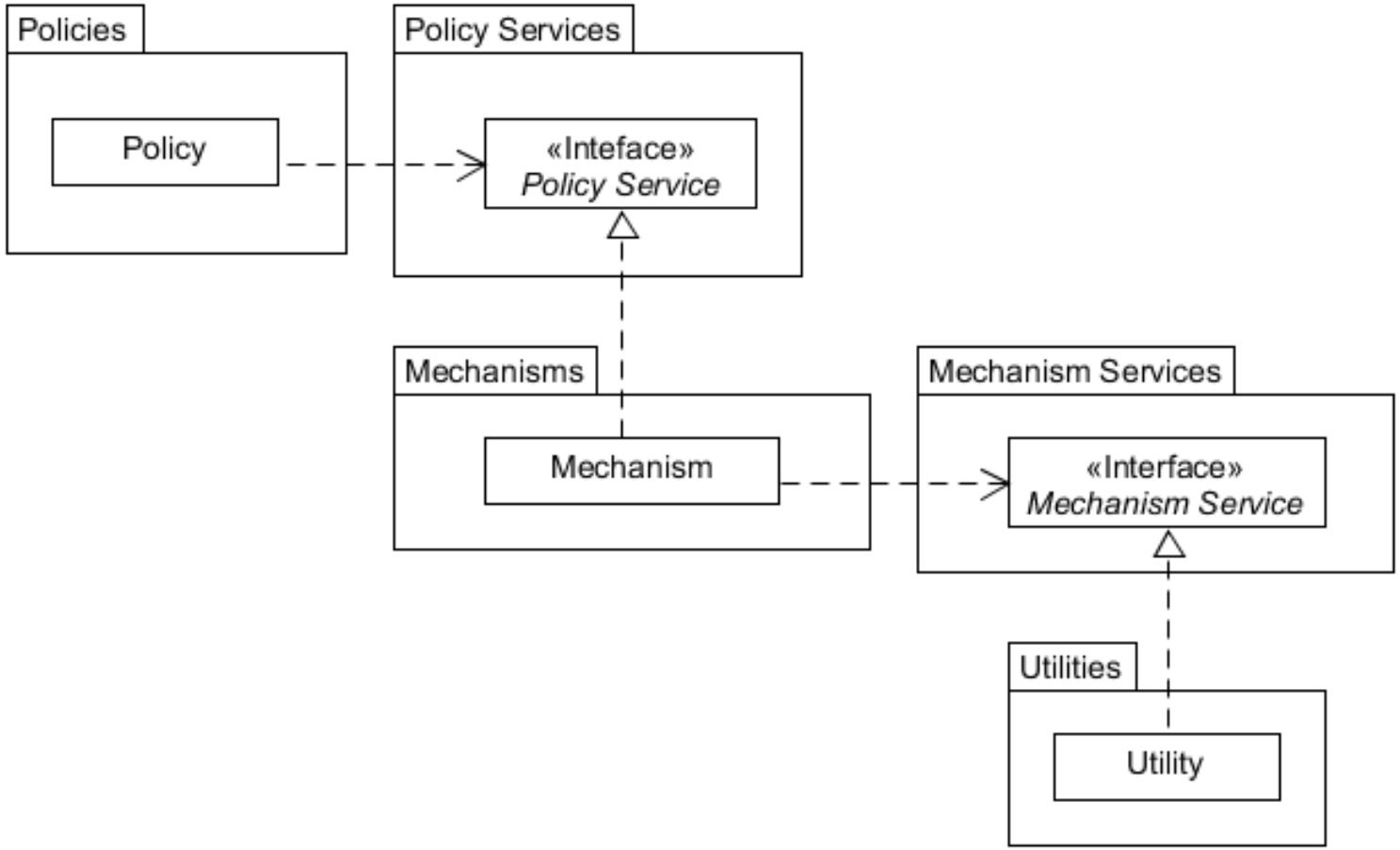


LearnQuest

Dependency Inversion Principle

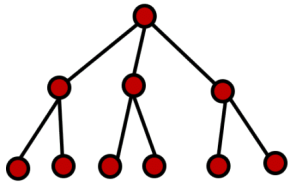
- Abstraction should be used in place of concrete implementations
  - ✓ High-level modules should not depend on low-level modules, and both types of modules should depend on abstraction (interface)
  - ✓ A concrete implementation should depend on a defined abstraction rather than an abstraction that is derived from a concrete implementation
- The purpose of the DIP is to manage the dependency between high and low-level modules abstractly
  - ✓ High and low-level modules are designed independently
  - ✓ The bindings between modules are done through abstractions implemented as interfaces (suppleness)
  - ✓ Interfaces are defined before the code is written

Dependency Inversion Principle

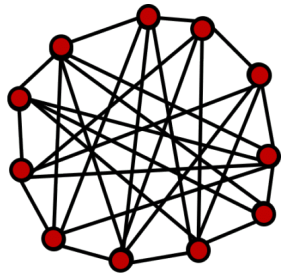


Inversion of Control

- Traditional top-down flow of control
  - ✓ Starts at a code entry point (e.g., the main() method)
  - ✓ Flow of control passes from calling modules to called modules
  - ✓ Then control returns to the calling module
- Ideal for execution of algorithms
  - ✓ Not so much for event driven applications



"Top-down"

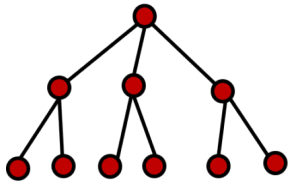


"Bottom-up"

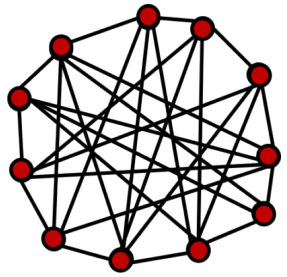
LearnQuest

Inversion of Control

- In IoC, control starts with an "event" or client request
  - ✓ Different flows of control depending on the event
- Assumes a framework of some type (module graph)
- When a module needs a service
  - ✓ A service is located
  - ✓ The service is bound to the module
  - ✓ Often bound using DiP and DI



"Top-down"

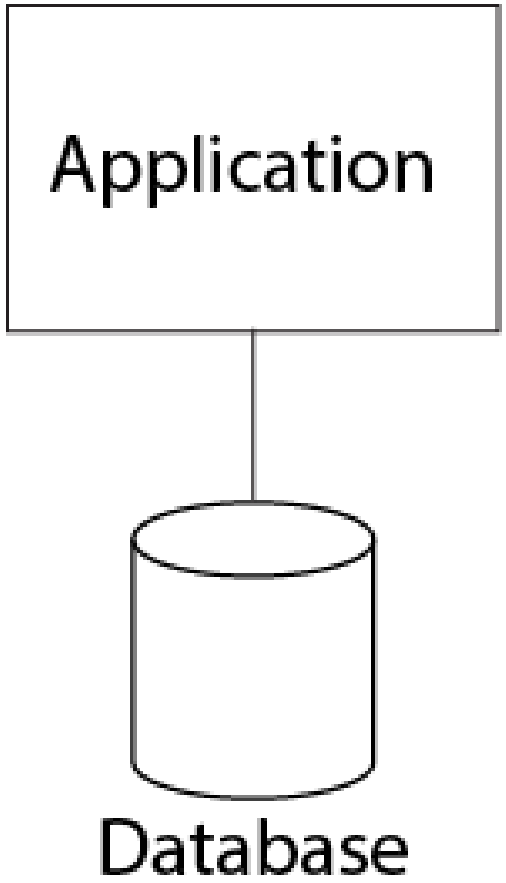


"Bottom-up"

LearnQuest

Single Process Monolith

- All code is packed into a single monolith
  - ✓ Often a good choice for small applications and organizations
- Does not scale well in terms of
  - ✓ Functionality
  - ✓ Performance
  - ✓ Complexity

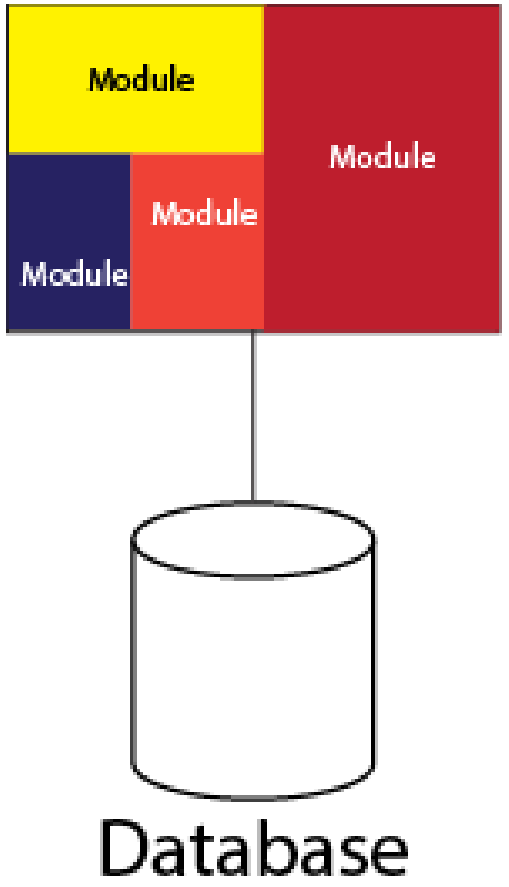


Business Analogy

- Startup businesses are monoliths
- There are a few people who do everything
- The enterprise is small enough that this model works
  - ✓ It's actually counterproductive to have a highly structured departmental organization with just a few employees
- Early versions of applications are similar
  - ✓ Simple enough that all of the code is manageable
  - ✓ Single code base for the whole app
  - ✓ Flat or minimal systems structure

Modular Monolith

- Process is divided into multiple modules
- All modules must be deployed together
- Does not scale well in terms of
  - ✓ Functionality
  - ✓ Performance
  - ✓ Complexity

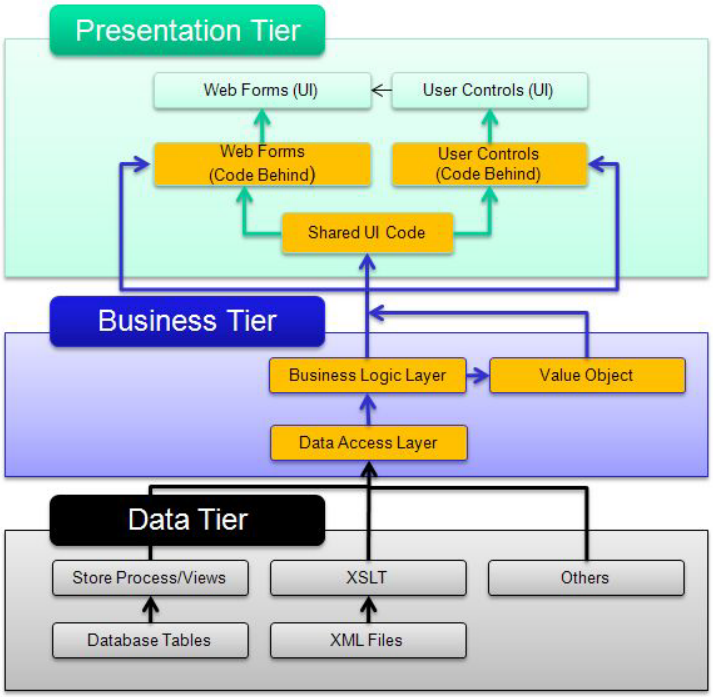


Business Analogy

- As startup grows, it adds more people
- The organization starts to become ineffective
- The entrepreneurial “all hands-on deck” model doesn’t scale well
  - ✓ Processes become chaotic
  - ✓ Difficult to manage
  - ✓ Productivity stalls
- At some point the business must reorganize along functional lines

Three-Tier Pattern

- Application is layered
  - ✓ Layers correspond to development roles
  - ✓ Each layer cuts across multiple business domains
- Changes to a business area needs changes in all three layers
- Tend to find:
  - ✓ Horizontal coupling within tiers
  - ✓ Vertical coupling between tiers



In Real Life

- A hospital provides a healthcare service
  - ✓ Made up of individual microservices
    - Laboratory, X-Ray, Pharmacy
  - ✓ Each microservice:
    - Specializes in a specific domain activity
    - Only that microservice performs that domain activity
  - ✓ Each microservice operates autonomously
  - ✓ Microservices make sense intuitively

LearnQuest

Real Life APIs

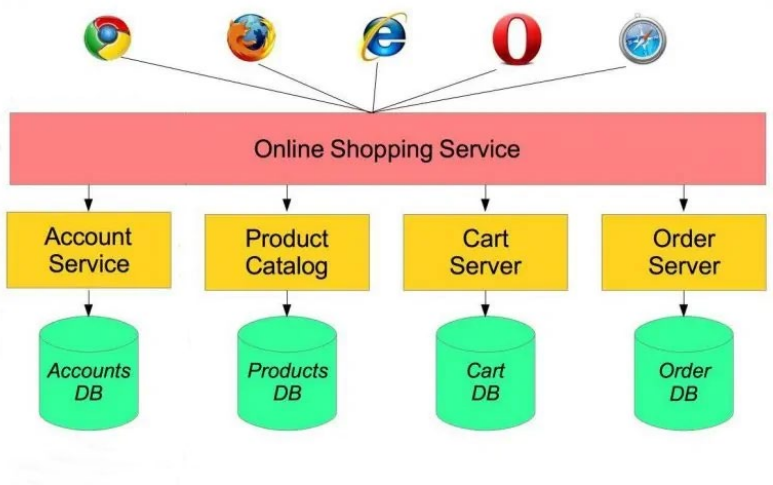
- Microservices request services from each other
  - ✓ They do not need to know the internal workings of the other microservice
- Requests are made through interfaces
  - ✓ Called APIs in software engineering
  - ✓ These are often paper based in the real world
  - ✓ Take the format of requisitions and official forms and paperwork used to make requests

COVID-19 VIRUS LABORATORY TEST REQUEST FORM[1]

Submitter information			
NAME OF SUBMITTING HOSPITAL, LABORATORY, or OTHER FACILITY*			
Physician			
Address			
Phone number			
Case definition:[2]	☐ Suspected case ☐ Probable case		
Patient info			
First name		Last name	
Patient ID number		Date of Birth	Age:
Address		Sex	☐ Male ☐ Female ☐ Unknown
Phone number			
Specimen information			
Type	☐ Nasopharyngeal and oropharyngeal swab ☐ Bronchoalveolar lavage ☐ Endotracheal aspirate ☐ Nasopharyngeal aspirate ☐ Nasal wash ☐ Sputum ☐ Lung tissue ☐ Serum ☐ Whole blood ☐ Urine ☐ Stool ☐ Other:		
All specimens collected should be regarded as potentially infectious and you must contact the reference laboratory before sending samples. All samples must be sent in accordance with category B transport requirements.			
Please tick the box if your clinical sample is post mortem ☐			
Date of collection		Time of collection	
Priority status			
Clinical details			
Date of symptom onset:			
Has the patient had a recent history of travelling to an affected area?	☐ Yes ☐ No	Country Return date	
Has the patient had contact with a confirmed case?		☐ Yes ☐ No ☐ Unknown ☐ Other exposure:	
Additional Comments			

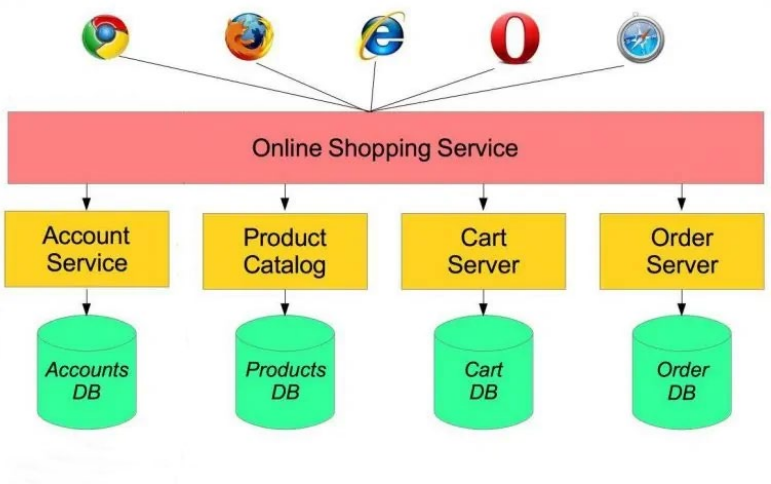
Microservices

- Software architectural design choice
  - ✓ Enterprise applications are deployed as a set of microservices
  - ✓ Each microservice represents a clearly defined area of domain functionality
  - ✓ Each service manages requests from other services through an API



Microservices

- Microservices are deployed as individual applications
  - ✓ Microservices are isolated from each other
  - ✓ They only communicate through APIs
  - ✓ They require an infrastructure to deploy them and ensure that they are all running and available
  - ✓ As an application executes, it makes requests of various microservices



Why Microservices

LearnQuest

Microservices Essential Principles

- Single Responsibility
- Discrete
- Carries its own data
- Transportable
- Ephemeral

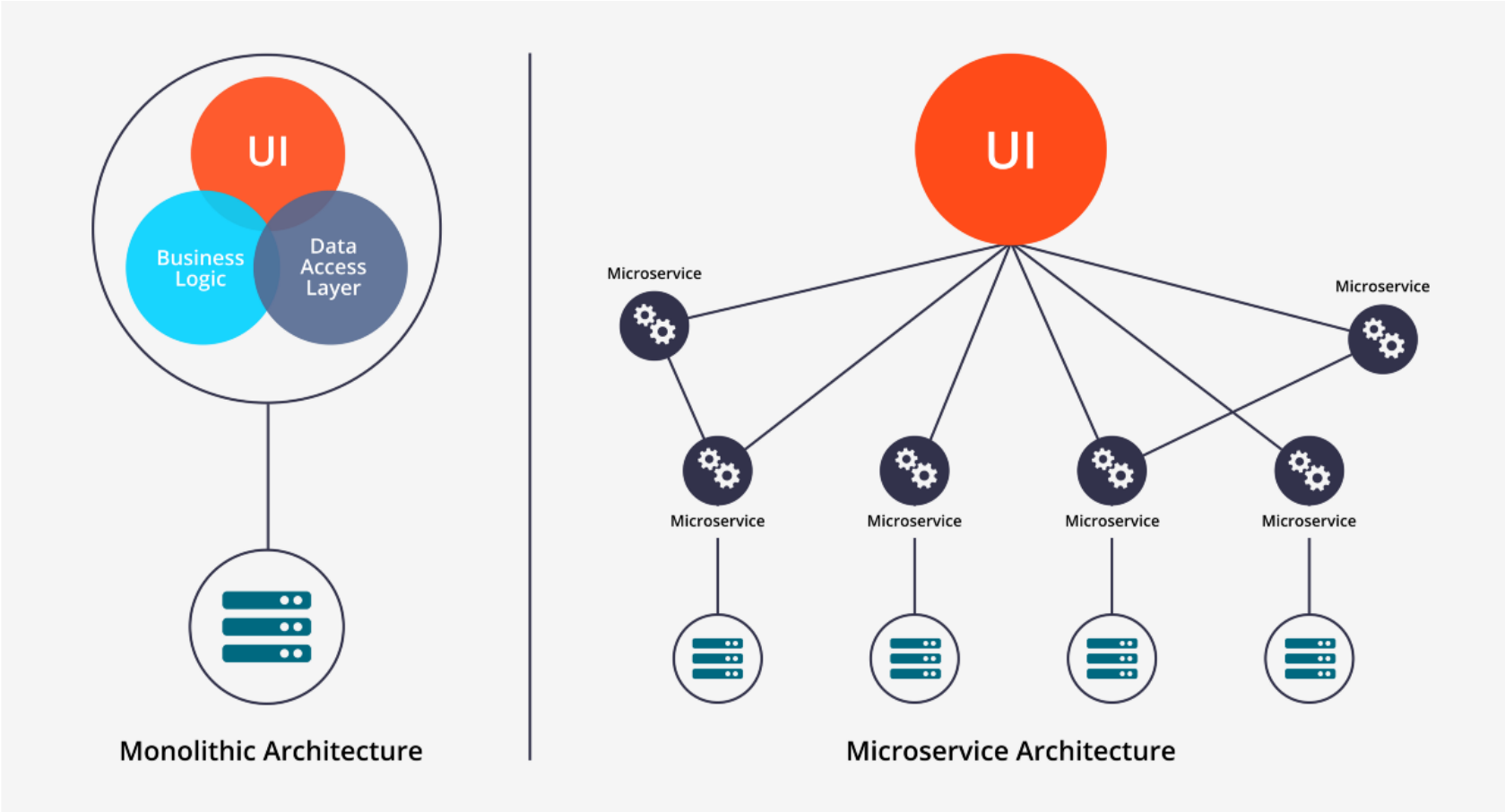
Microservices Core Concepts

- Microservices are small, independent, composable services
  - ✓ Each service can be accessed by way of a well-known API format Like REST, GraphQL, gRPC or in response to some event notifications
- Breaks large business processes into basic cohesive components
  - ✓ These basic process actions are implemented as microservices
  - ✓ The business process is executed by making calls to these microservices
  - ✓ Each microservice provides one cohesive service
  - ✓ Microservices do not exist in isolation
  - ✓ They are part of a larger organization framework

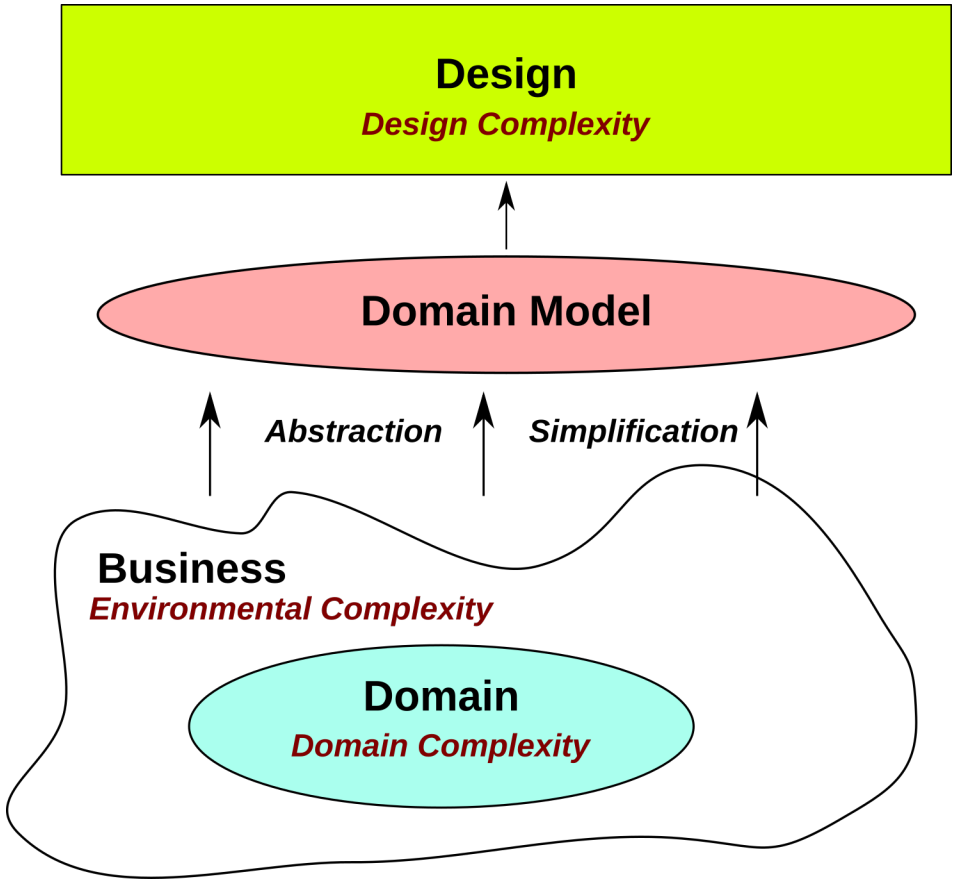
Microservices Core Concepts

- Microservices coordinate with other microservices to accomplish the tasks normally handled by a monolithic application
- Microservices communicate with each other synchronously or asynchronously
- Microservices make applications easier to develop and maintain
- BUT A LOT HARDER TO MANAGE

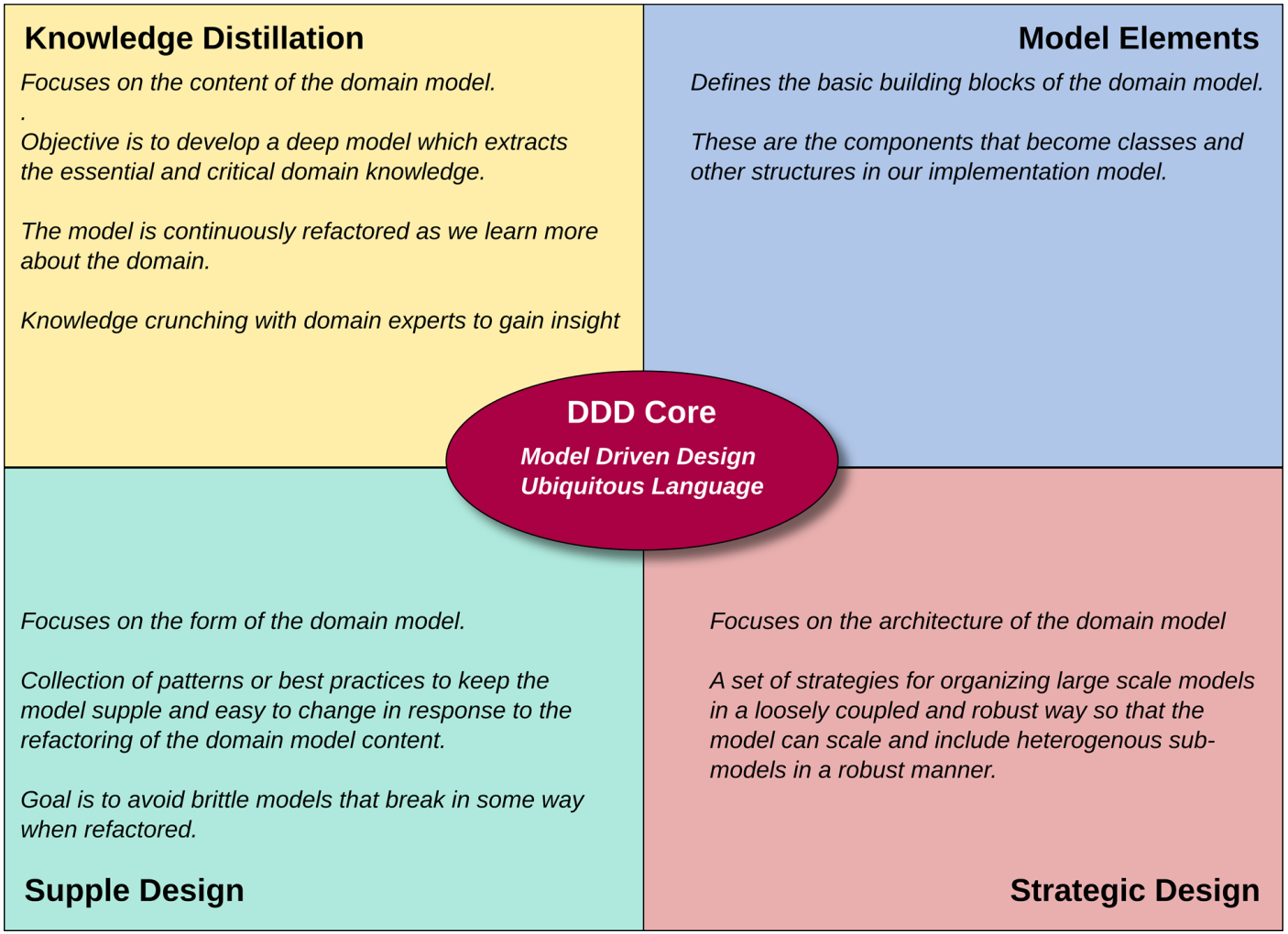
Monolith to Microservice



The Domain Model

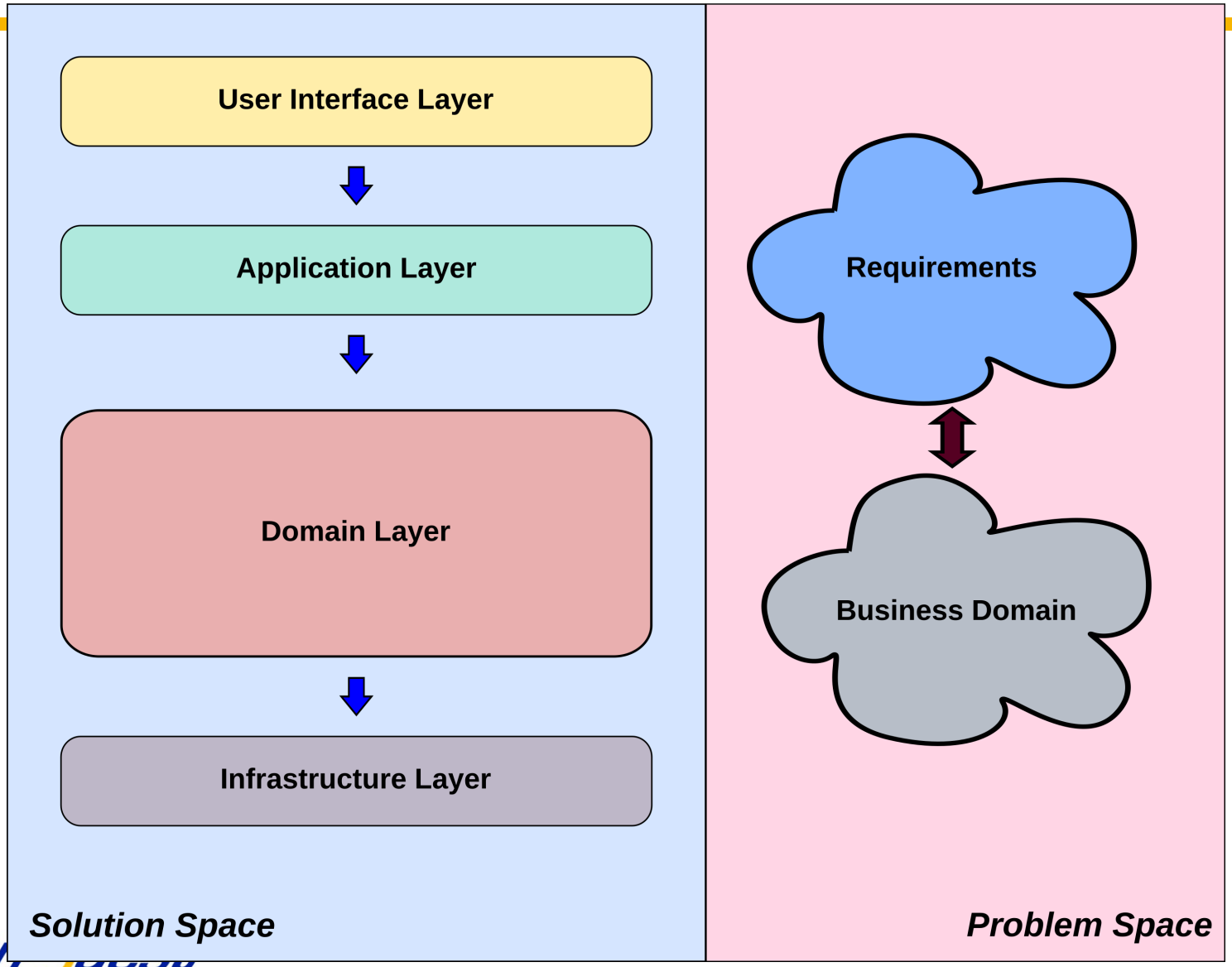


The DDD Quadrants



LearnQuest

The Layered Architecture



The Problem Space

The Problem Space

Represents the business "as it is now."

The Requirements

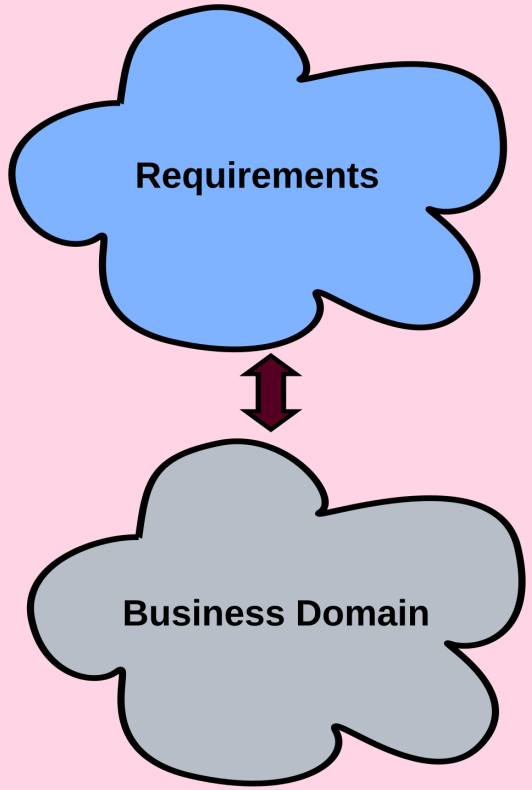
Requirements are statements of the needs of the stakeholders and users. Requirements describe what they need to do (eg. Add an account, make a payment, get a customer history).

These can also be thought of as the tasks that the business needs executed or "what" the business does.

The Business Domain

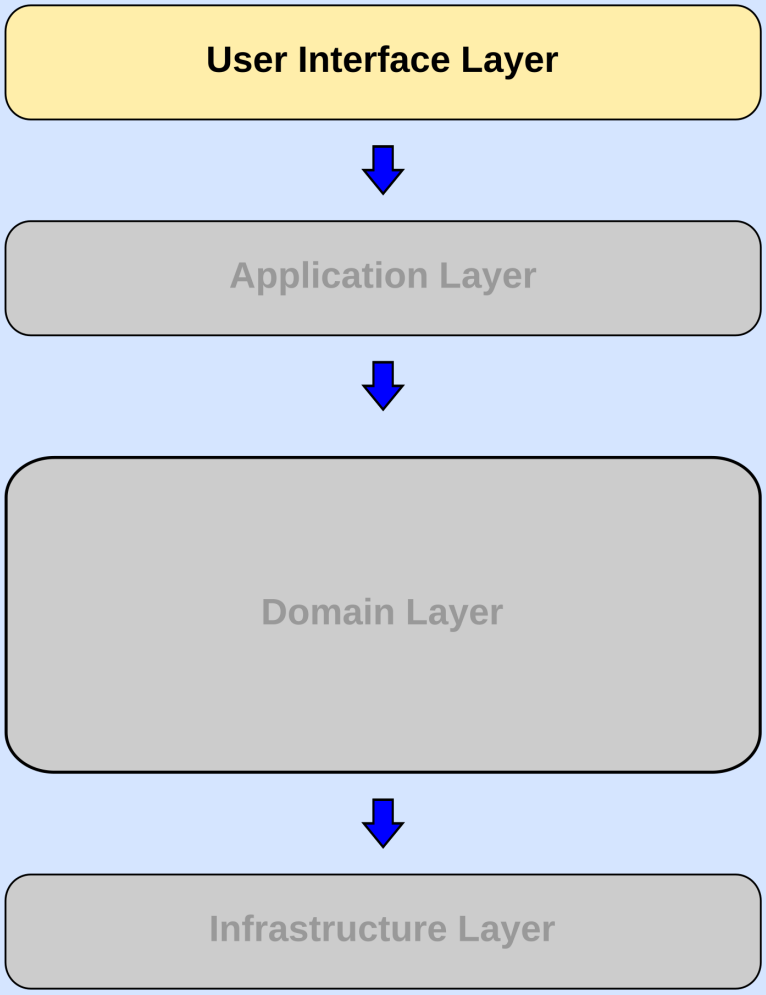
This contains the algorithms, business rules, data objects, files, conceptual artifacts (eg. accounts, payments, invoices, shipments) used by the business to meet the requirements of the stakeholders.

*The business domain describes "**how**" the business goes about executing the tasks requested in the business requirements.*



Problem Space

User Interface Layer



Solution Space

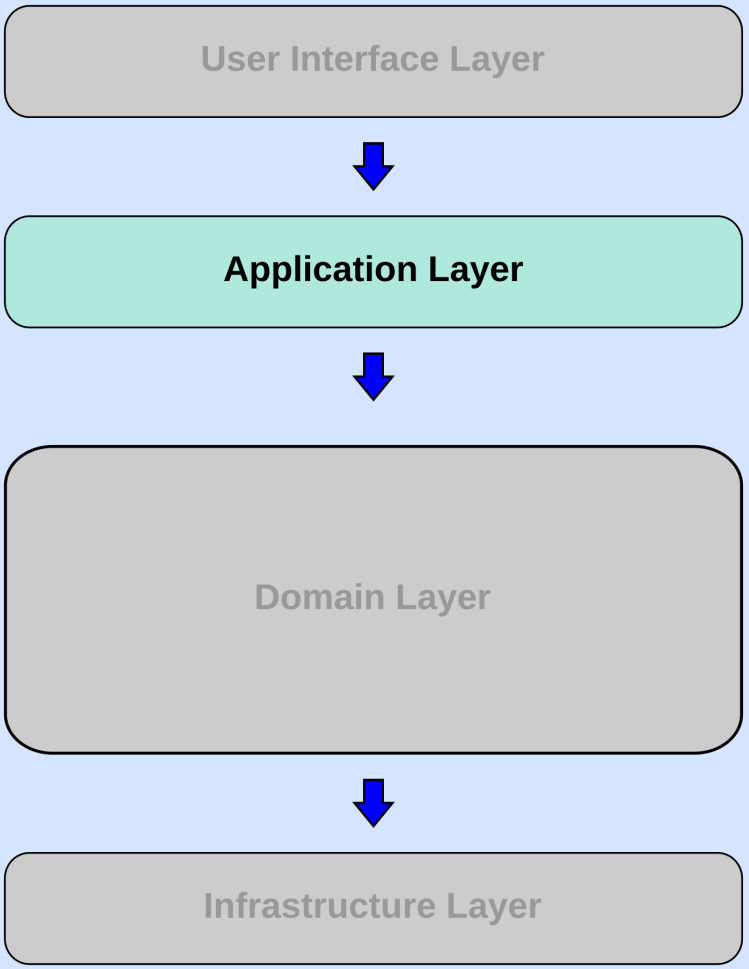
Responsible for showing information to the user and interpreting the user's commands.

This layer only calls the applications in the Application Layer. It may occasionaly call a service provided by the Intrastructure Layer.

There may be multiple actual interfaces in this layer (web, RESTful API. etc).

Users may be humans or they may be other systems or computers.

The Application Layer



Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems.

The tasks this layer are responsible for are meaningful to the business or necessary for interaction with the application layers of other systems.

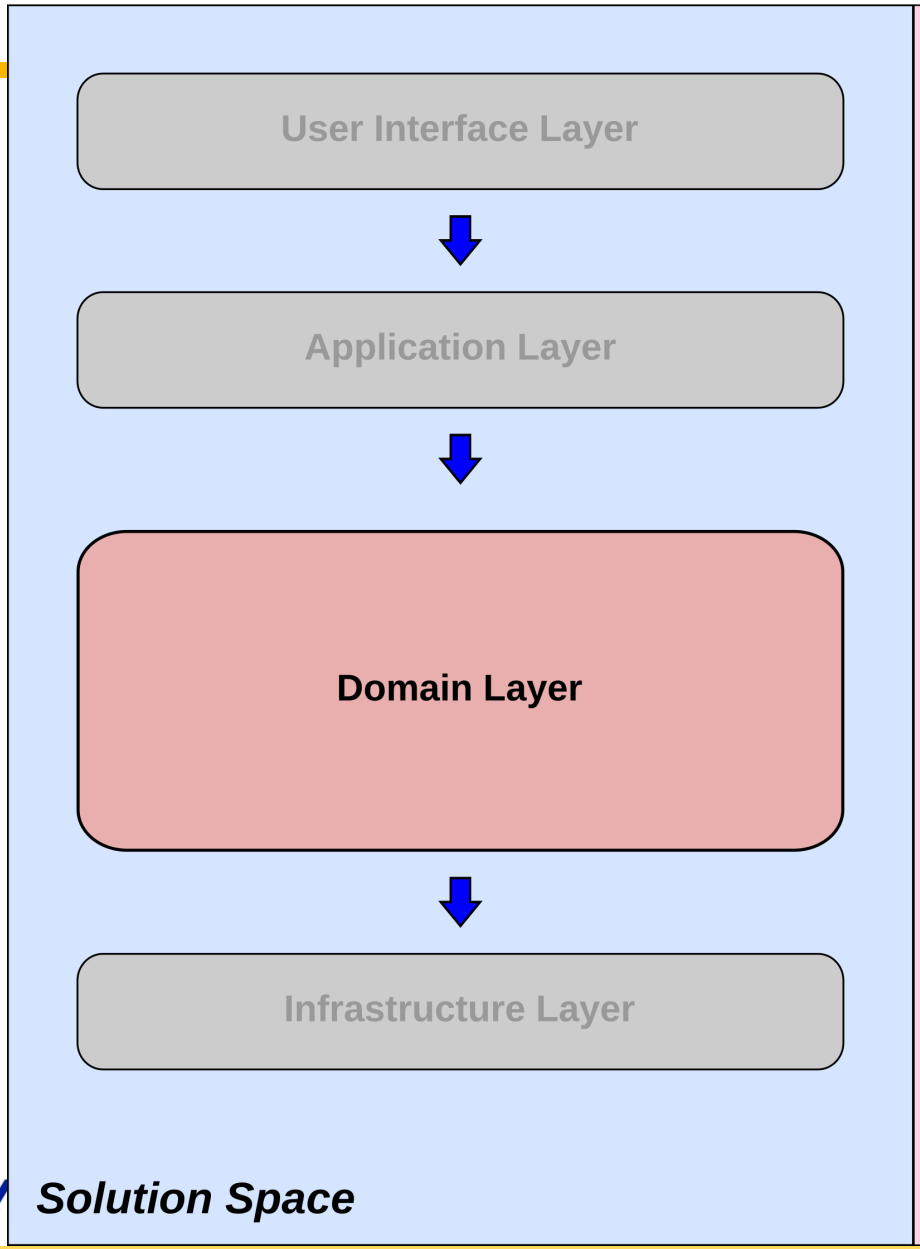
This layer is kept thin. It does not contain business rules or knowledge. It only coordinates tasks and delegates work to collaborations of domain objects in the domain layer. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.

The specifications for an application usually describe what happens at this layer. Application features are defined in terms of functional specification.

This is the layer at which we usually write ATDD tests.

This layer may reference services in the Infrastructure layer.

The Domain Layer



The domain layer is where the domain model lives.

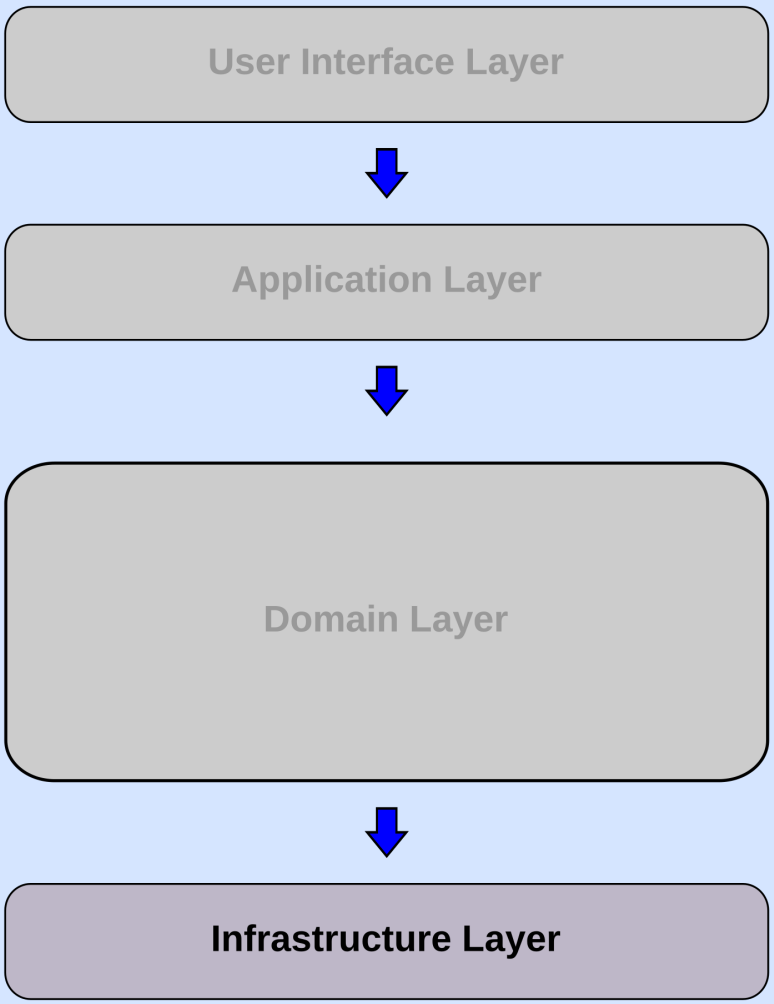
Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This layer is the heart of business software.

We often think in terms of business objects when considering data at this level. Data is represented by objects and their actual presistent storage is handled by the infrastructure layer.

Business logic is contained in objects representing algorithms, policies and procedures.

No implementation details are included in the domain model.

The Infrastructure Layer



Provides generic technical capabilities that support the higher layers: message sending for the application, persistence for the domain, drawing widgets for the UI, and so on. The infrastructure layer may also support the pattern of interactions between the four layers through an architectural framework.

The application and domain layers call on the SERVICES provided by the infra-structure layer.

For example, sending an email is done by requesting a "message sending service" from the infrastructure layer.

The domain or application may know what message to send and when to send it, but the "how" the message is sent is delegated to the infrastructure layer

APIs

- Application Programming Interface
- Interface to allow a client to make requests of and receive responses from a component
- A microservice API is the only part of the service that is visible to a client
- There are standardized APIs that allow for portability and ease of development and deployment
- Web services are APIs that use standard Internet web protocols
  - ✓ HTTP and HTTPS for example

Restful Web Service

- Uses HTTP to work with "resources" that are represented as URL endpoints
- REST uses HTTP methods to define operations
  - ✓ GET – Read a resource
  - ✓ POST – Create a resource
  - ✓ PUT – Update a resource
  - ✓ DELETE – Delete a resource
- Data is described in either JSON or XML
- Restful operations are atomic and transactional

Restful Web Service

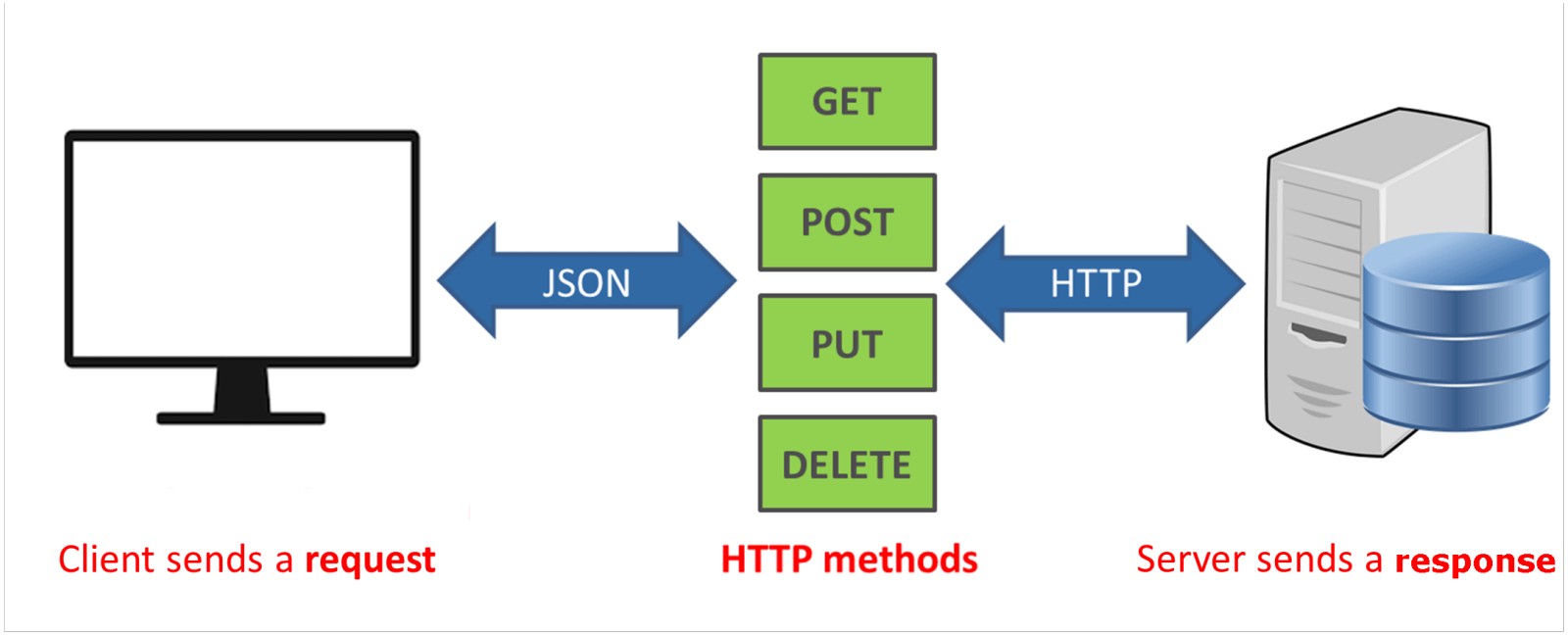
Representational state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services.

Web services that conform to the REST architectural style, called RESTful Web services, provide interoperability between computer systems on the Internet.

RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations."

Wikipedia

Restful Web Service



Restful Web Service Design

- Client-Server
  - ✓ There should be a separation between the server that offers a service, and the client that consumes it
- Stateless
  - ✓ Each request from a client must contain all the information required by the server to carry out the request
  - ✓ The server cannot store information provided by the client in one request and use it in another request
- Cacheable
  - ✓ The server must indicate to the client if requests can be cached or not

Restful Web Service Design

- Layered System
  - ✓ Communication between a client and a server should be standardized in such a way that allows intermediaries to respond to requests instead of the end server, without the client having to do anything different
- Uniform Interface
  - ✓ The method of communication between a client and a server must be uniform
- Code on demand
  - ✓ Servers can provide executable code or scripts for clients to execute in their context
  - ✓ This constraint is the only one that is optional

Restful Web Service Design

- The CRUD operations are indicated by the HTTP operation
- All endpoints should refer to some meaningful entity in the domain
- Operations can be represented by some form of transaction object
  - ✓ For example, recording that something has been sold can be done by sending a “sale” object with the appropriate data
- The objects referred to in the API are always determined by business domain

Restful Web Service Design

GET http://api.coolcars.io/cars/

GET http://api.coolcars.io/cars/{id}

DELETE http://api.coolcars.io/cars/{id}

POST http://api.coolcars.io/cars/

```
 {
  "make":"chevrolet",
  "model":"Silverado 3500",
  "year": 2004,
  "vin":"1GCJK33104F173427"
  }
```

Response: {”data”:{“id”: “8b7138db-0c7c-4e2e-8494-bd5daf1788e0”}

Bad Restful Web Service Design

- The REST endpoints are verbs (imperative)
- Creates a link between application logic and the web service API
- Add or changing app logic requires changing the web service
- For example, adding a “lease” capability to the business
- All business process logic, state information and interaction with the client is done at the application layer
  - ✓ Remember the hospital, patient interactions with X-ray, etc. are accessed by the doctor
  - ✓ The “application” is “hospital visit”

Restful Web Service Design

```
GET http://api.coolcars.io/cars/

GET http://api.coolcars.io/cars/sell/{id}

GET http://api.coolcars.io/cars/buy/{id}

POST http://api.coolcars.io/cars/trade
{
  "to ": {
    "make": "chevrolet",
    "model": "Silverado 3500",
    "year": 2004,
    "vin": "1GCJK33104F173427"
  },
  "from": {
    "make": "honda",
    "model": "civic",
    "year": 1990,
    "vin": "2HGED6349LH506746"
  }
}
```

LearnQuest

Web Service vs Microservice

- Web services are applications that use web protocols
  - ✓ Usually REST but can also include other technologies like SOAP
  - ✓ Defined by how clients interact with them
- Microservice are an architectural pattern
  - ✓ Can use any technology, not just web protocols
  - ✓ Defined by its structure and behavior
- Microservices are often build as web services because the technology is easy to deploy as a microservice
- The 12 factor app principles are guidelines for converting web services to microservices

12 Factor App for Microservices

Factor	Description
I. Codebase	One codebase tracked in revision control, many deploys
II. Dependencies	Explicitly declare and isolate dependencies
III. Config	Store config in the environment
IV. Backing services	Treat backing services as attached resources
V. Build, release, run	Strictly separate build and run stages
VI. Processes	Execute the app as one or more stateless processes
VII. Port binding	Export services via port binding
VIII. Concurrency	Scale out via the process model
IX. Disposability	Maximize robustness with fast startup and graceful shutdown
X. Dev/prod parity	Keep development, staging, and production as similar as possible
XI. Logs	Treat logs as event streams
XII. Admin processes	Run admin/management tasks as one-off processes

Questions