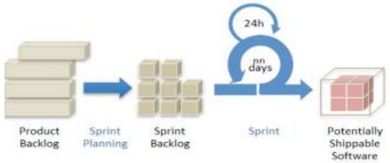
Factors Affecting SDLC
- Requirements
- Process (Resources, Time)
- Criticality: Consequences of not doing
- People: Competence, Technology

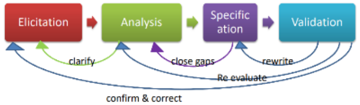
SDLC Models
- Waterfall
- Spiral
- Rapid Prototyping
- eXtreme Programming
- Rational Unified Process
- Test driven development
- Agile (SCRUM, Crystal, etc)



DevOps Processes
- Continuous Integration (CI):
Automate integration of code into main code base
- Continuous Delivery (CD): Code is automatically prepared for a release
- Continuous Deployment (CD): Code that passes every pipeline is released
- Continuous monitoring and logging
- Communication and collaboration
- Infrastructure as Code

Types of requirements
- Business req: Describe Why organization is implementing the system
- User req: Describe goals or tasks the user must be able to perform with the product
- System req: Hardware of Software issues
- Functional req: Specify the behaviour the product will exhibit
- Quality req /Non-Functional req: Describes how well the system performs
- Constraints: States the limitation on a design or implementation choices
- Data req: Describe data or structure

Requirements Development Phases
- Elicitation: Discover requirements
- Analysis: Analyse, Decompose, Derive, Understand, Nego Requirements, Identify gaps
- Specification: Written and illustrated requirements for comprehension, review.

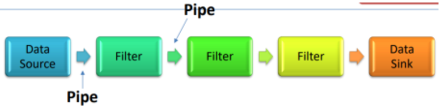


- Validation: Confirm correct set of requirements that enable developers to build a soln

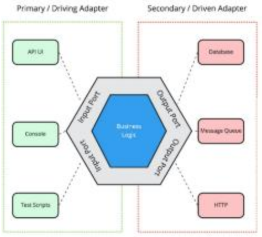
Taxonomy of Attributes
- Availability, Performance, Portability, Installability, Reliability, Usability, Reusability, Integrity, Robustness, Efficiency, Scalability, Interoperability, Modifiability, Verifiability
- Security: Security issues (Privacy, authentication, integrity)
- Safety: Whether a system can harm someone or something

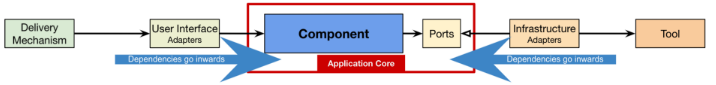
Software Architecture
- Control Flow: Reasoning is on computational order
- Data Flow: Reasoning is on data availability, transformation, latency
- Call and Return: Control moves from one component to another and back
- Message and Event: Communicate via Event Notifications / Message passing
- Decomposition / Componentizing or Packaging: Horizontal, Vertical Slicing.

Common Architecture Styles / Patterns
- Layered architecture (2/3/n tier Architecture)
  - Layers are independent in terms of development.
  - Layers have distinct and specific responsibility
- Data abstraction and OO – Organization
  - Component: Object encapsulates data representations and operations
  - Connector: Interactions that enable procedure invocations
- Pipe and Filter
  - Data enters the components one at a time until the data sink
  - Components: Source, Sink, Filters
  - Each components have inputs (read) and outputs (produce)
  - Filter transforms the input
  - Connectors are pipes that bring them from one to the next

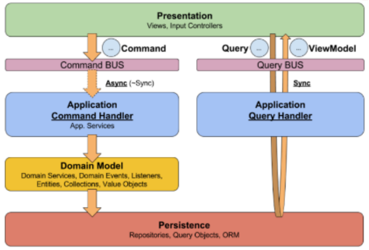


- Shared Repository
  - Maintains all data in a central repository shared with all components
  - Availability, quality and state of data triggers and coordinate control flow of app
  - Components: Central Data Structure + Independent components that operate it
  - Connectors: Interactions between repo and other components (Depends on system)
- Implicit Invocation
  - Event announcement implicitly causes the innovation of functions in other modules.
  - Broadcasts instead of invoking procedure directly.
  - Components register an interest by associating procedure with event
  - When event is announced, procedure is invoked
  - Components: Components who provide procedures and events
  - Connectors: procedure calls
- Hexagonal Architecture
  - Components: User interface, Application Core, Infrastructure Code
  - Adapters: Primary / Secondary
  - Primary / Driving Adapter: Tell the application to do something
  - Secondary / Driven Adapter: Told by application to do something
  - Ports: A way to be used / use application core
  - Application Layer: Organizing the application core
  - Domain Layer: contains data and logic to manipulate data. Independent of biz logic
    - Domain Service: Handles logic that spans multiple domain model objects
    - Domain Model: Contain the business objects that represent smth in the domain.





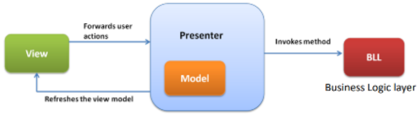
- Command Query Responsibility Segregation (CQRS)
  - Separate commands from queries
  - Commands: Change application state but return no data
  - Queries: Returns data but don't change the application state.
- Domain Driven Design (DDD)
  - Does not dictate any specific architectural style
  - Requires only model to be kept isolated from technical complexities
  - User Interface Layer: Interacts with external systems



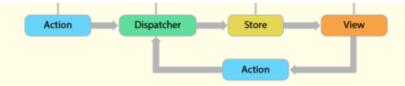
  - Application Layer: Business process flows are handled
  - Domain Layer: Core of the application where biz problems are solved
  - Infrastructure Layer: Where external Services are accessed
  - Domain: Critical and fundamental / foundational concepts behind the business
  - Ubiquitous Language: Shared language between domain experts and devs
  - Bounded Context: Explicit boundary within which a domain model exists
  - Repository: Deal with / abstract storage concerns
- Microservices
  - A independent, standalone capability designed as an executable that communicates with other microservices through standard lightweight inter-process communication
  - Aims to be
    - Organised around business capabilities, loosely coupled and highly cohesive, Owned by small team (Conway's Law), Independently deployable
  - Coupling
    - Domain coupling: Interactions between services model interactions in real domain
    - Temporal Coupling: Async / Sync / Caching
  - Deployment
    - Each service can have its own database or shared database
    - Each microservices have its own deployment, resource, scaling and monitoring req
    - Service Instance per host / Service instance per container
  - Orchestration: Rely on central brain to guide and drive the process
  - Choreography: Inform each part of the system of its job and let it work it out.
  - Service Discovery
    - Service instance registers/deregister with service registry
    - Client-Side Discovery: Client queries a service registry and sends request
    - Sever-Side Discovery: Client makes request to load balancer and load balancer routes request

How small is small	Communication Complexity	Architectural Complexity	Change Complexity	Deployment Complexity
Size is not the primary goal. It is to sufficiently decompose the app to facilitate deployment /development	Much more complex than a monolith due to IPC / Async / Sync / Partial failures	Partitioned Database architecture. Transactions pose the challenge of consistency	Implementing changes that span multiple services need to be carefully planned	Multiple moving parts that need to be configured scaled, deployed. Need service Discovery

- Model View Controller Architecture
  - Separation of concerns
    - Results in modularity
    - Output separated from user input handling
  - Facilitates extensibility, new view / controller can be added for new interfaces
  - Model: Triggers view update
  - View: Queries model for state, forward user actions to controller
  - Controller: Updates models as per actions, selects a new view if required.
- Model View Adapter Model (MVA)
  - All communication between model and view must flow through an adapter.
  - The controller becomes a communication hub
- Model View Presenter (MVP)
  - Model: Presents business entities or domain models
  - View: Light weight, only UI elements
  - Presenter: Presents user actions to the backend system. Presents it after getting a response from the user
- **Presenter Model (PM)**
  - Suited for Rich UI Applications
  - Model: Represents the state of the view
  - View: Contain UI Specific elements, forwards user actions



  - Presenter: Receives events from views, processes them, and updates the model. Also responsible for updating state. Invokes methods in Business logic.
- Model-View-View-Model (MVVM)
  - View: Only contains UI Elements. Commands, Binds, and notifies View Model
  - ViewModel: Equivalent to presentation model in PM Pattern
  - Model: Business logic layer of the application
- WebMVC
  - Controller also handles first HTTP request
  - Extra step in creating static bundles of HTML, CSS, JS for direct hosting via a simple View Controller.
- Flux
  - Action, Dispatcher, Stores, Views
  - Views may forward actions through the system in response to user actions.



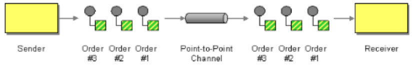
  - Action: Raised by the view when the user interacts with the UI controls in View
  - Dispatcher: Holds the context to data store and forwards the action from View to Store
  - Store: Registered with dispatcher and contains data. Creates change events to update view
  - View: Respond to change events and make appropriate changes

Messaging Patterns
- Can be Synchronous / Asynchronous
- Single of multiple receives
- Persistent or transient
- Synchronous Request Reply Pattern
  - Makes both processes believe they are in the same process
  - RPC style communications tend to be synchronous
- Asynchronous Request-Reply pattern
  - Decouple backend processing from a frontend host, where backend processing needs to be asynchronous for frontend still needs a response
  - Sends a request and receives a response. Client polls until it gets a different response
- Asynchronous Message passing
  - Communicate by inserting messages in queues
  - EG: Message Queues, JMS
  - Sender only guaranteed that message will eventually be inserted in recipient's queue
  - No read guarantees
- Persistent Communication (Store and forward delivery)
  - Messages are stored at each intermediate hop along the way until the next node is ready to take the delivery of the message (EG: Emails
- Transient communications
  - Messages are buffered only for small periods of time
  - If message cannot be delivered, it is discarded (EG: TCP/IP)

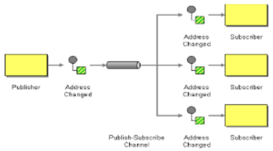
- Combinations

	Asynchronous	Synchronous
Persistent	- Sender Keeps executing without blocking - Message may take an arbitrary amt of time to reach receiver - Sender may or may not be running by the time the message reaches receiver - Guarantee that the message will eventually reach EG: Emails	- Sender is blocked until an ack for receipt is received - The message persists in the receiver's queue for an arbitrary amount of time - EG: Messaging / Chat Apps
Transient	- Sender continues execution after sending a message - Receiver must be running, otherwise message is discarded - Even if any router along the way is down, message is discarded - EG: UDP	3 Types - Receipt Based: Sender blocks until ack is received. Ack is a receipt and does not say anything about receiver - Delivery Based: Sender will block until receiver takes the delivery of the message. Ack comes a little bit later than receipt based. (Async RPC) - Response Based: Sender blocks until it receives a response (EG: RPC)

- Messaging patterns
  - Encapsulated method requests and data structures to be sent across the network
  - Includes a header specifies the type of info, origin, destination, size, and other metadata
  - Payload that contains the information
  - Message intent
    - Command message
      - Specifying a function or method on the receiver that the sender wishes to invoke
      - Sender tells receiver what code to run
    - Document Message
      - Enables sender to transmit one of its data structures to the receiver
      - Does not specify what the receiver should do with it
    - Event message
      - Notifies the receiver of the change in the sender.
      - The sender does not tell the receiver how to react. It just provides notification
- Message channel: Connect the collaborating senders and receivers using a message channel that allows them to exchange messages
- Request / Reply channels
  - Requestor: Sends a request message and waits for a reply message
  - Replier: Receives the request message and response with a reply message.
- Return address
  - The request contains a return address to tell the replier where to send the message.
- Correlation ID: Specifies which request this is for
- Request-Replying chaining
  - When a request causes a reply and the reply is another request, it causes chaining
  - Useful if the application wants to retrace the path of the messages.
- Request channels
  - Point to Point (p2p)
    - Request is processed by a single consumer



  - Publish-Subscribe channel (PubSub)
    - Request is broadcasted to all interested parties

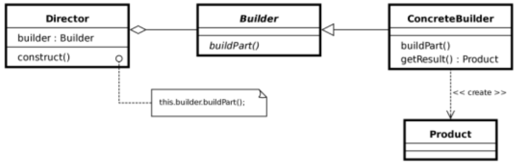


- Special Case
  - Invalid Message Channel: Error messages
  - Dead Letter channel Message that could not be delivered
  - Data type channel: All messages on the channel are the same datatype
- Message routing
  - Consumes msg from one channel and pushes them to another channel
  - Content-Based routers: Routes based on the data contained inside
  - Message Filter: Eliminates undesired messages from a channel based on condition
  - Context based routers: Decide based on context (load balancing, test, or failover)
- Message Splitter: Split 1 message into multiple messages
- Message Aggregator: Merge messages into a single message

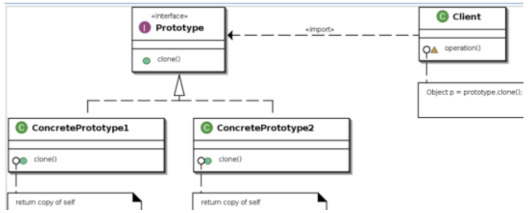
- Message Scatter-Gather: Sends a single message to several participants and reassembles it back into a single reply.
- Message translator: Converts from one message format to another
- Canonical Data Model: Provides additional level of indirection between app formats
- Message Endpoints
  - Interface between application and messaging system.
  - Can be used to send or receive messages but not both
- Polling consumer: Proactively reads message when it is ready to consume them
- Event-Driven consumer: Reactively processes a message on its arrival

Creational Patterns

- Builder: Separate the construction of a complex object from its representation so that the same constructor can be used to create different representations.

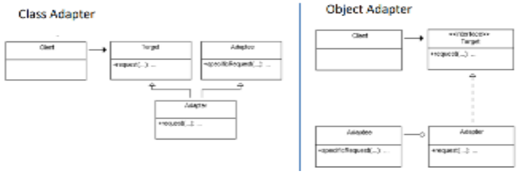


- Prototype: Create an object by cloning another as necessary

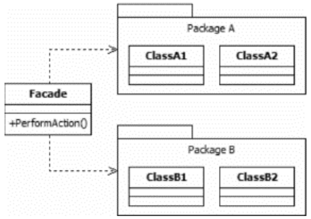


Structural Patterns

- Adapter: Convert the interface of a class into another interface client expects. Let different classes work together.

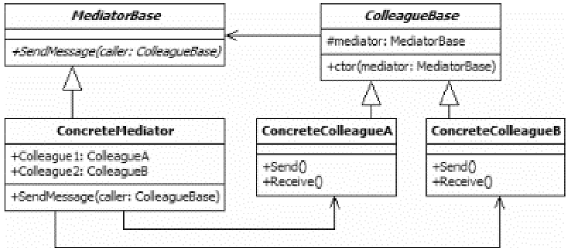


- Façade: Provide a unified interface in a subsystem. (IE: Gateway)

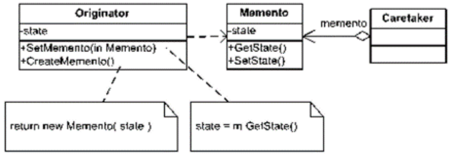


Behavioural Patterns

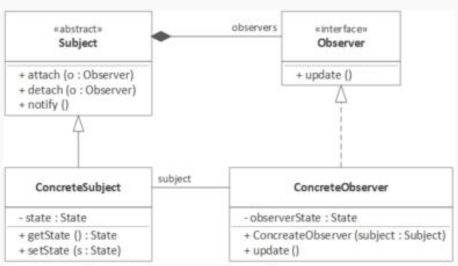
- Mediator: Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping object from referring to each other



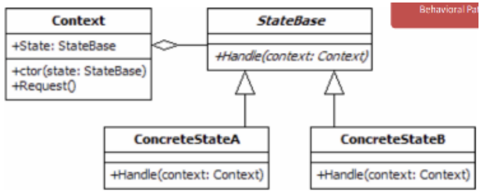
- Memento: Capture and externalize an objects internal state without violating encapsulation.



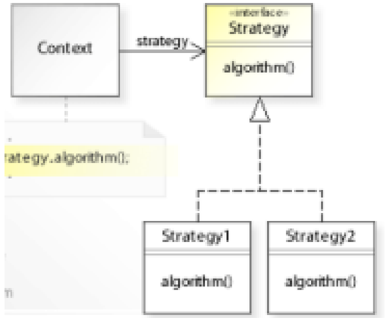
- Observer: Let objects observe the behaviour of other objects so they can sync.



- State: Allow object to alter its behaviour when the internal state changes

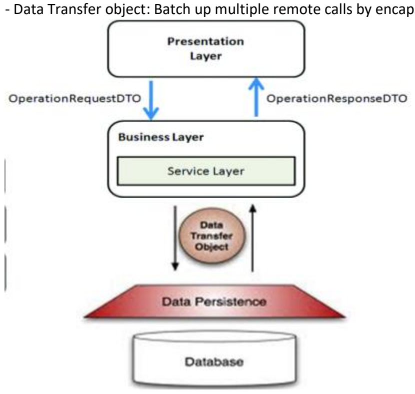


- Strategy: Define a family of algorithms, encapsulate each one and make them interchangeable.



Other Patterns

- Data Transfer object: Batch up multiple remote calls by encapsulating all the data to be sent


Presentation Layer
OperationRequestDTO
OperationResponseDTO
Business Layer
Service Layer
Data Transfer Object
Data Persistence
Database