Software Development Process

Types

1. **Embedded**: Limited and specific set of functions. Found in industrial or consumer devices.
2. **Real-Time**: Where timing is important. Hard — must be on time; firm — little room for flexibility; soft — greater room.
3. **Concurrent**: Different computations run across the same or overlapping time periods.
4. **Distributed**: Runs across more than one computer. Usually connected via a network.
5. **Edge-dominant**: Computation performed at endpoints instead of centrally.

Agile Process

Agile is a methodology that emphasises iterative development and cross-functional collaboration.

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation (fastidious paperwork, not dev docs).
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

SCRUM. One popular Agile framework is SCRUM. Work is done in sprints, where a subset of the product backlog is cleared. There is often a daily 15 min SCRUM meeting.

Software Delivery

DevOps is a set of software development practices that combine development and operations. Intended to reduce the time between committing changes to the change hitting production, while ensuring quality.

- **C-Int.**: Auto build, unit test, deploy to staging, and acceptance test. Allows teams to detect problems early.
- **C-Del.**: Same as above, except with manual deployment to production. Ensures that every good build is potentially ready for production release.
- **C-Dep.**: Same as above but with auto deployment to production. Automates the release of a good build.

Specifying Software Requirements

A requirement is a capability needed by a user or must be met by a system, i.e. what should be implemented. Can be of the system behaviour, property or attribute.

Types of Requirements (Reqs)

- **Business Reqs**: Why the organisation is implementing the system, e.g. reduce staff costs by 25%.
- **User Reqs**: Goals the user must be able to perform with the product, e.g. check for a flight using the website.
- **Functional Reqs**: The behaviour the product will exhibit, e.g. passenger shall be able to print boarding passes.
- **Quality Attributes**: How well the system performs, e.g. mean time between failure $\geq$ 100 hours. A type of non-functional reqs.
- **System Reqs**: Hardware or software issues, e.g. the invoice system must share data with the purchase order system. Affects functional reqs.
- **Data Reqs**: Describes data items or structures, e.g. product number is alphanumeric.
- **External Interfaces**: Connections between your system and environment, e.g. must import files as CSV. Affects functional reqs.
- **Constraints**: Limitations on design and implementation choices, e.g. must be backwards compatible. A type of non-functional reqs.
- **Business Rules**: Actual policies or regulations. Constrains business, user and functional reqs.

Generally, the flow is: Business Reqs → Vision & Scope Docs → User Reqs → User Reqs Specification → Functional Reqs.

Quality Attributes

External

- **Availability**: Measure of the planned up time during which the system is fully operational.
- **Installability**: How easy it is to install the system for the end-user.
- **Integrity**: Preventing information loss and preserving data correctness.
- **Interoperability**: How readily the system can exchange data and services with other software and hardware.
- **Performance**: Responsiveness to user actions. May also compromise safety if e.g. a safety system responds poorly.
- **Reliability**: Probability of the software executing without failure for a specific period of time.
- **Robustness**: Degree to which a system performs when faced with invalid inputs, defects and attacks.
- **Safety**: Prevents injury or damage to people or property.
- **Security**: Authorisation, authentication, confidentiality, etc.
- **Usability**: User-friendliness and ease of use.

Internal

- **Efficiency**: How well the system utilises the hardware, network etc.
- **Modifiability**: How easily designs and code can be understood, changed and extended.
- **Portability**: Effort needed to migrate the software from one environment to another.
- **Reusability**: Effort required to convert a software component for use in other apps.
- **Scalability**: Ability to grow to accommodate more users, servers, locations, etc. without compromising performance or correctness.
- **Verifiability**: How well the software (components) can be evaluated to demonstrate that it functions as expected.

Reqs Development Phases

- **Elicitation**: Discover reqs via interviews, workshops, event-response tables, focus groups, prototypes, watching users, system interface analysis, user interface analysis, document analysis.
- **Analysis**: Analyse, decompose, derive, understand, negotiate, identify gaps.
- **Specification**: Written and illustrated reqs for comprehension and use.
- **Validation**: Confirm that we have the correct set of reqs.

The output of the phases should be:

- **Rights, Responsibilities and Agreements**: All major stakeholders confident of development within a balanced schedule, cost, functionality and quality.
- **SRS**: Discussed later.
- **Change Control**: Process to ensure that changes to a product or system are introduced in a controlled and coordinated manner.

A **Software Reqs Specification (SRS)** would contain System Reqs, Functional Reqs, External Interfaces, Quality Attributes and Constraints. It describes what the system must do. A **User Reqs Specification (URS)** describes end-user requirements.

Other Details

Prioritisation Techniques. High-Med-Low. Must-Should-Could-Wish. $100 approach. Quality Function Deployment.

Traceability. Requirements should be traced to a use case ref, design doc ref, code module ref, test case ref, etc.

Validation vs Verification. **Validation** is about whether you have the right reqs, and if they trace back to business objectives. **Verification** is whether you have written the reqs right, i.e. complete, correct, feasible, priority, unambiguous. Can be checked informally by passing the reqs around, or formally through formal inspection.

Test Planning. Test planning in parallel with requirements development can help detect many errors early in the software development process, reducing rework costs.

Software Architecture Styles

Architecture Styles are recurring architecture designs. They describe structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

- **Component**: Models an application-specific function
- **Connector**: Models interactions between components for transfer of control and/or data
- **Configuration**: Topology or structure

Architecture Styles

Main Program and Subroutine / Call-and-Return. A set of functionalities with a shared data store is a **subroutine**. A single thread of control passed from parent to child in a hierarchy. **Remote procedure call systems** are MPS systems with parts on different computers in a network. Can have any topo structure that links subroutines by subroutine calls (connector). **Pros.** Easy to modify. Can extend by adding modules. Easy to analyse control flow. **Cons.** Hard to parallelise. Hard to distribute. Awkward to handle exceptions.

Layered. Hierarchy with layers providing service to the layer above it and acting as a client to the layer below it. Layers contain various elements, and communicate via procedure calls (connector). **Pros.** Abstraction. Partitions complex problems. Easy to substitute one layer for another. Easy to reuse and port. **Cons.** Hard to "layerise" some systems, esp. when high-level functions needs coupling with low-level implementations. Performance degradation due to communication between layers.

Object-Oriented & Abstract Data Types. **OO**: Encapsulate data into objects. Communicates through methods (connector). **ADT**: Very similar to OO, except that the focus is on designing the data structures right, e.g. invariants, etc. **Pros.** Hierarchical sharing of code via inheritance. Polymorphism. Encapsulation. Low coupling. Easy to understand structure. **Cons.** System overhead to maintain objects. Need to explicitly reference name and interface of other objects. Potential side effects between coupled objects.

Communicating Process — Client/Server. Independent processes or objects that communicate through messages (connector). Messages are passed through named participants. A classic example is the client-server model, which can be asynchronous or synchronous. **Pros.** Easy to distribute. Easy to parallelise. **Cons.** Need to know names of communicating processes. Communication across networks may be slow.

Event-Based Implicit Invocation. Individual components announce data they will publish. Other components can register an interest, i.e. subscribe. When the data appears, the subscribers are invoked, i.e. event announcement "implicitly" causes the invocation of procedures. Messages (connector) are passed among unnamed participants. E.g. GUI responding to mouse & keyboard events. **Pros.** No need to know name of subscribers. Easy to parallelise. Decouple control. Real-time. Can easily replace components. **Cons.** Complex implementation. Hard to test. Subsystems don't know if or when events will be available. Event sequence not determined. Cannot assume other components will respond.

Batch Sequential Processing. Data flows through components, where one component finishes processing the data before passing it on, i.e. as batches instead of as a stream. **Pros.** Quickly processes data. Interchangeable components. Jobs can run in background. Easy to understand as business process steps. **Cons.** Not good at interactive applications. Hard to make concurrent. No event handling or fault tolerance.

Pipe-and-Filter. Filters are stream transducers that incrementally transform data, and is generally stateless. Pipes are stateless and simply transfer data. **Constraints.** Independence; i.e. filters should not share state. Anonymity; i.e. filters should not know which filter is before or after them. Concurrency; i.e. should be able to run the filters concurrently. **Pros.** No complex component interaction. Easy to reuse. Easy to parallelise. Easy to maintain and enhance. **Cons.** Not good at interactive applications. May need a common representation, i.e. pack and unpack costs.

Data Repository. Two main types of components: a central data structure that represents the current state, and a collection of independent components operating on the data store. Evolves into client-server if the clients are independently executing processes. Becomes **blackboard** if notifications are sent to subscribers when data of interest changes **Pros.** Independent clients. Decouples data storage from manipulation. Easy to add clients. Can share large amounts of data. **Cons.** Communication between clients may be slow. Subsystems must agree on the database structure.

Software Architecture Patterns

Architecture Patterns are ways to solve common architectural problems.

Hexagonal Architecture

This pattern allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual runtime devices and databases.

Ports. The application communicates over ports to external agencies. The protocol for a port is given by the purpose of the conversation, i.e. it acts as an API.

Adapters. An adapter converts the API definition to the signals needed by that external device and vice versa. E.g. GUI maps the movements of a person to the API of the port.

Primary and Secondary. There can be both primary and secondary ports and adapters. A primary actor drives the application, i.e. takes it out of quiescent state to perform one of its functions. A secondary actor is driven by the application, either to get data or to notify.

Analysis. A port identifies a purposeful conversation. There will typically be multiple adapters for any one port, for various technologies that may plug into that port. The primary purpose of this pattern is to focus on the inside-outside asymmetry, pretending briefly that all external items are identical from the perspective of the application.

Onion Architecture

Imagine an onion with many layers. Reduces coupling in n-tier architectures.

Domain Layer. Innermost layer. Contains domain objects, interfaces, etc. Should have no heavy code or dependencies. Flat.

Repository Layer. Provide object saving and retrieval, usually via a database. Contains data access pattern.

Services Layer. Interfaces with common operations, e.g. CRUD, and hold business logic. Communicates between UI and repository layers. Should maintain decoupled service interfaces.

UI Layer. Outermost layer, peripheral concerns like UI and tests.

Command Query Responsibility Segregation

Split the models into ones for write and read, i.e. Command and Query. For many problems, particularly in complex domains, having the same model for commands and queries leads to a more complex model that does neither well. Variations include having separate databases, and having separate subsystems that can be scaled based on read and write loads.

Domain Driven Design

by Hanming Zhu

Keep models isolated from complexities and have them focus on domain logic concerns. There are a few layers, though not necessarily in any particular stacking order.
UI Layer. Interaction with external systems.
Application Layer. Handles business process flows. Create and update domain entities. May handle persistence, transactions and security. But no business logic.
Domain Layer. Core of the application. Defines business rules, entities, value objects, etc. Minimal dependencies and third party libraries. Agnostic of persistence.
Infrastructure Layer. Accesses external services such as databases (and repositories), messaging systems, email services.
Domain. The subject area to which the user applies a program. Can be segmented into subdomains, which can be classified as **generic**, i.e. facilitates the business and is found across multiple domains, e.g. invoicing / communication; **supporting**, i.e. ancillary but directly related to business. Core domain is what makes the system worth writing.
Ubiquitous Language. A language structured around the domain model and used by all team members to connect all activities of the team with the software.
Bounded Context: Separates ubiquitous languages. As we have many problems, we may not be able to use the same language throughout. But we can cluster sets of problems where the same specialised language can be used, and make boundaries. Contains code for a single **subdomain**.
Context Map: Relates different bounded contexts.
Entity: Domain model object with unique identity. Design revolves around who they are.
Value Object: Domain model object defined by attributes. Two value objects with the same attributes are equal. Normally immutable. Design revolves around their values.
Aggregate: Composition of one or more entities, and may also contain value objects. The root is the **aggregate root**.
Repository: One per aggregate or root. Interfaces should be in domain model layer, implementation in infrastructure layer.

Repository vs Data Access Object

DAO is an abstraction of data persistence. Repository is an abstraction of a collection of objects. DAO would be considered closer to the database, often table-centric. Repository would be considered closer to the Domain, dealing only in Aggregate Roots. Repository could be implemented using DAO's, but you wouldn't do the opposite.

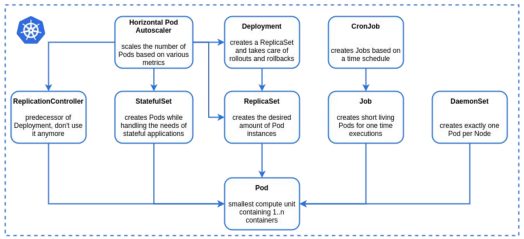
Microservices

A microservice is an independent, standalone **capability** that communicates through lightweight communication.
Loose Coupling, High Cohesion. Know as little as it needs to about other services. Related behaviour should be together.

- **Implementation vs Domain vs Temporal Coupling**: Temporal coupling occurs due to communication or caching.
- **Units of Cohesion**: Can be an aggregate or a bounded context (collection of aggregates).

Deployment. Each microservice can have its own deployment, resource, scaling and monitoring. Can have a service instance per host or per container. Can have a database per service, which may result in data duplication. An API gateway can also be used to provide a single entry point, encapsulating the internal architecture.
Coordination. Orchestration uses a central brain to guide and drive the process. Choreography informs each part of its job and let it work out the details.
Registry Patterns. Client-side discovery involves client querying a service registry (database of available instances) then determining the network locations of available instances (+ do load balancing). Server-side discovery involves letting the load balancer query the registry. Instances are registered with the registry on startup and deregistered on shutdown.

Kubernetes



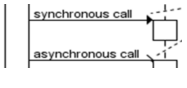
Containerisation allows for the running of applications in virtually isolated environments within a single virtual machine. We can use an image `registry/name:tag` to run a container. To increase availability, we can run multiple replicas in different VMs. Kubernetes helps to manage these.
Cluster. A set of nodes that run containerised applications.
Node. May be a virtual or physical machine. Each node is managed by the control plane and contains the services necessary to run Pods.
Kubelet. An agent that runs on each node. It makes sure that containers are running in a pod.
Kube-Proxy. Network proxy that runs on each node. Can specify network rules to allow network communication to your pods inside or outside of your cluster.
Container Runtime. Software responsible for running containers in each node. Can be Docker, containerd, CRI-O, etc.
Pod. Smallest deployable unit of computing that you can create and manage. A group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.
Control Plane. The container orchestration layer that exposes the API and interfaces to define, deploy, and manage the lifecycle of containers.
API Server. Lets you query and manipulate the state of API objects, e.g. pods, namespaces, configmaps and events.
Scheduler. Watches for newly created pods that have no node assigned, and selects best node for it based on factors like resource reqs, hardware constraints, data locality, etc.
Controller Manager. Daemon that embeds the core control loops, which watches the shared state of the cluster through the API Server and makes changes attempting to move the current state towards the desired state. For stateless apps, this is called reconcillation loop. Includes node controller, replication controller, endpoints controller, and service account & token controllers.

Message Design Patterns

Messaging systems or Message-Oriented Middleware (MOM) are software that provide messaging capabilities.

- Synchronous vs Asynchronous
- Single vs Multiple Receivers
- Persistent vs Transient. Persistent is store-and-forward, i.e. stored at each intermediate hop until next node is ready. Transient allows buffering of messages for small periods of time, and can be discarded, e.g. TCP/IP.

Communication Types



Synchronous Request-Reply. Wait for response. Makes both processes believe they are in the same process space. RPC allows a program to make a procedure execute in another address space. Marshalling and unmarshalling done by client and server stubs respectively.
Asynchronous Request-Reply. Backend runs asynchronously while frontend still needs a clear response. One way is to keep polling.
Asynchronous Message Passing. Async and persistent, with intermediate storage while sender and receiver are not active. Can be single or multiple receivers (multiple is pub/sub). Messages are guaranteed to be inserted into queues, but no guarantee on when or if the message will be read.

Messaging Patterns

Message Construction. A message contains:

- **Header**: Contains metadata. Can contain message intent — command message, which tells the receiver what to do; document message, which sends data but does not really tell the receiver what to do; event message, which simply notifies the receiver about a change.
- **Properties**: Optional. For message selection and filtering. Three kinds — application-related, provider-related and standard properties.
- **Payload**: Body, data structures.

Message Channels. Connect collaborating senders and receivers. A channel transmits one way. Two-way messages need two channels. Some concepts:

- **Return Address**: Request contains an address to tell the replier where to send reply to.
- **Correlation ID**: Specifies which request the reply is for. Can be chained.
- **Message Sequence**: Basically break the message down into smaller segments.
- **Point to Point (P2P)**: Request processed by single consumer.
- **Publish-Subscribe Channel**: Request broadcasted to all interested parties via topics.
- **Invalid Messages**: Queue to move a message to when it cannot be interpreted.
- **Dead Letter**: Queue to move a message to when it cannot be delivered.
- **Datatype Channel**: Separate channel for each type of data, e.g. XML, byte array, etc.

Message Routing. Consumes messages from one channel and reinserts them into different channels based on conditions. Some concepts:

- **Content-Based**: Examines message content and routes. Message filter is a special kind of content-based router that discards messages based on content.
- **Context-Based**: Decides destination based on context, e.g. load-balancing.
- **Message Splitter**: Splits a single message into multiple.
- **Message Aggregator**: Aggregates correlated messages into a single message.
- **Scatter-Gather**: Broadcast to multiple participants and aggregates replies into a single message.

Message Transformation. Transform application-layer data structures, data types of fields, data representations e.g. ASCII to Unicode, transport protocol, e.g. TCP/IP to sockets. Also called Message Translators or Channel Adapters. A **Canonical Data Model** may also be used, which is an "adapter" superset model.
Message Endpoints. Interface between app and messaging system. Channel-specific, one instance handles either send or receive. Can be synchronous, i.e. polling consumer, or can be asynchronous, i.e. event-driven receiver.

Technologies

Redis. In-memory solution is very fast, suited for real-time. Can be used as data store, database, cache, message broker. Can be scaled via:

- **Master-Slave Replication**: Master can write and replicate to slaves, slaves can read. If master goes down, need to manually configure one slave to become new master.
- **Redis Sentinel**: Provides monitoring, notif upon failure, acts as a source registry, and helps promte a replica to master.
- **Redis Cluster**: Multiple masters and up to 1000 slaves. Can have partitioning / data sharding, data duplication / redundancy, sychronisation and failover ops.

Java Message Service (JMS). Has Publish-and-Subscribe and Point-to-Point. Uses ActiveMQ message broker, which handles large messages better than Redis, has persistent queues, and has both PubSub and Point-to-Point. Apache Camel can be used to provide pattern implementations that ActiveMQ does not have, e.g. routing pattern. ServiceMix is a service bus that provides service-to-service communication.
Advanced Message Queuing Protocol (AMPQ). An open standard application layer protocol for MOM. Is language and platform agnostic. RabbitMQ is an implementation in Erlang. Messages pass through an exchange and get distributed to queues:

- **Direct**: Maps an exchange to a specific queue based on routing key.
- **Fanout**: Routes messages to all queues from bound exchange.
- **Topic**: Routes messages to queues based on either fill or portion of routing key matches.
- **Headers**: Same as topic but routes based on header values instead of routing keys.

Object Interaction Design Patterns

Creational

Builder. Construct complex objects step by step, instead of cramming everything into a huge constructor. Specify a `Builder` interface and a few `ConcreteBuilder`s. The `Builder` interface has methods to set each of the attributes. The `ConcreteBuilder`s then have a `getResult()` method.
Prototype. Specify a `Prototype` interface with a `clone()` method that can copy the object, including its private fields. Need to consider deep vs shallow copy.

Structural

Adapter. Basically "wrap" a service object to provide an interface that clients can use.
Facade. Provide a simplified interface to a library, a framework, or any other complex set of classes.

Behavioral

Observer. `Observer`s can subscribe to/unsubscribe from `Subject`s. `Subject` can either **push** information to `Observer`s, or notify to **pull** information.
Mediator. Define a `Mediator` that encapsulates how objects communicate, and the objects delegate that to the `Mediator`.
Memento. Allows an `Originator` that can create or restore from a `Memento`, which captures the former's state. A `Caretaker` can help to safekeep `Memento`s and clean them up when the `Originator` is deallocated.
State. Prevent massive switch cases by defining a class for each state, and delegating `Context` behaviour to those `ConcreteState`. Difference with Strategy is that `ConcreteState`s are aware of each other and transition the `Context`'s state accordingly.
Strategy. Encapsulate algorithms into `Strategy`s, then have the `Context` use the correct one. The `Strategy` can either be pushed data by the `Context`, or can be called and let it pull data directly from the `Context`.

Others

Data Transfer Object (DTO). Bundle all data items that might be needed into a single `DTO` used for querying or updating attributes together. Reduce multiple remote calls into a single call.

by Hanming Zhu