System Design - Modern Architecture
"Imagine/Construct"

Vision

"building the impossible"

Vision is a declaration of goals and refinement of the business language nouns and verbs that delineates and articulates the scope, functions, and parameters of the solution.

Rational Unified Process is the high level view of the development process. This is a fully iterative approach, which can work if the entities (which are deepest rooted dependencies) are encapsulated inside services that pass well defined and simple messages.

RUP (Rational Unified Process)

A process framework defining a staged delivery to expose issues as early as possible.

- Inception - make a initial evaluation to determine the cost and value of the project, decide to proceed to elaboration. It should produce minimal viability through essential use cases.
- Elaboration - perform domain modeling and architecting. Produce a integration test suite to shake out the Representation Model and the architecture. This is a iterative process.
- Construction - complete the implementation of the Use Cases.
- Transition - Final release work not in the scope of continuous delivery.

Language

System Language is where concepts are identified, if there is a troublesome area use language to pinpoint, dissect, and and illuminate the concepts or stumbling blocks of a design.

System Domain

System Domain is the practice of distilling a vision description down into the essential components and features of a successful system that will leave a legacy as a elegant, aesthetic, and highly useful system.

Design

“Interrelation between use and model”

Perfect mathematical minimalism, elegant selection of mechanism and the interconnection into ingenious machinery is the ideal. Simplicity of design gives organic birth to behavioral complexity. Underlying the rich diversity of the problem is the solution’s essential truth in the USE/Model.

Design is the tight feedback loop between the learning and revising the USE/Model and validating it by writing integration tests in Python. The tests exercise the API’s and mockups of internal logic verify the needs of the internal logic are met by the API’s.

USE CASE/DOMAIN SPLITTING

USE cases are designed as sequence diagrams showing the interaction between the user and the system. Each USE case interaction with the system has a correlation identifier that identifies the case as it propagates through the system.

Service Domains can signal other Service Domains in a intra-domain vocabulary.

USE State Machines

Roles are built as state machines. Each state consists of a set of messages that can be sent from that mode. Each response from the system or choice by the performed by the user, or change in state is a transition to another immutable state.

Messages

Transitive entities, or messages passed between processes. This are read-only. They are indicated by italic UML objects in the domain file, and in the system file they are simply named.

Entities

Entities are persistent state with a cohesive, complete, and minimal set of attributes. They are refined by normalization where the identity of the Entity acts as the “key” to apply normalization.

In the domain file It is represented in UML as a bolded type, and can have a integrity label below in angle brackets. Integrity labels indicate that all the members are created and deleted as a cohesive set.

Service Domain Factoring

System components with related “knows about” topic relationships are gathered into Service Domains that consume and produce messages in that topic, serve persistent entities, and side effects. Each domain becomes a column in the sequence diagram.

Each Service Domain has its own description file.

SYSTEM VOCABULARY & CATALOG

A tool extracts messages and entities from the domain to construct a indexed message vocabulary and entity catalog.

Structure

“Systemic seperation of concerns”

From the Representation Model System layers shared across domains and the Data Fabric consisting of the combined vocabularies of the domains are combined into a single view. A tool might be needed to perform the synthesis of the data fabric.

Services

A cohesive interdependency of MESSAGES, ENTITIES, and API are represented as a UML package with the name of the Service in bold.

Service Ops

The OPS are described as one or more messages or entities in boxes that identify the vocabulary and/or catalog entities involved in an operation identified in an oval.

The operation is described as a series of processing rules, where USE case variations from the common case are indented and described.

Production

“The Right Way is the shortest path to success”

Layers

Application Layer

- Not the domain layer
- Responsible for coordinating the domain as tasks, system interfaces, service integrations, user interface, and parallelism if any.
- This layer is kept as thin as possible

Domain representation

- Representation of business concepts and rules. This is the Domain Model. Largely concerned with persistent state, function, and decomposition.
- Extremely loose coupling with the Application Layer and the Core layer.

Technical Core

- Implementation of processes and technical capabilities
- The technical details of system support, service integrations, storage.
- Purely technical factors such as performance, scaling, security, authentication, and configuration

Principles

“Principles are the oaths that are a Pyrrhic victory when discarded”

Twelve Factor App

1. One Code Base in Version Control
   The App codebase is one repository independent of environments configurations, and dependencies.
2. Explicit Dependencies and Dependency Isolation

Common code between repositories are packages, all dependencies are declared down to the operating system later via locking or static linking

3. Config Values in Environment Variables
   Config values are propagated from the environment bound launcher into environment variables consumed by the application processes.
4. Backing Services
   All resources are abstracted as config bound resources, local and remote.
5. Build, Release, Run
   Build is combined with config -> release , and finally produce a Run.
6. Stateless Processes
   All processes contain no locally attached state, all state is written to resources with ACID properties
7. Port Binding - no web server or reverse proxy
   The app binds to a port bare, no web-server or reverse proxy is needed to run it.
8. Scale via Processes
   Scale horizontally with processes.
9. Disposable Processes
   Make processes start and stop fast, make them disposable putting state in ACID resources
10. Dev/Prod Parity
    Keep Dev and Prod in sync so that changes can be rapidly promoted to Prod with confidence.
11. Logs - Log to stdout
    Log to stdout, use logging services to pick up the stream and make it analyzable.
12. Admin via one-off programs and REPL's
    Use one-off shell scripts and REPL's to do admin tasks, glue together dashboards out of ssh calls and log parsing

<u>Tests as Contracts</u>

Integration Tests as contracts - tests should reflect actual useful scenarios and not simply exercise coverage but miss the point.

Test the expected behavior of the interfaces on one level, and the implementation on another. Refactoring the implementation can

change the implementation tests; but the interface tests should remain unchanged.

To make it organized, and even possible to auto-generate docs and UML from reverse engineering the Unit Test code, make a Test object for each interface scenario being tested.

To share scaffolding put the scaffolding in module scope functions Taking the object as a single argument. Put a underscore under these functions.

Use DocStrings to briefly describe what the functionality of the scenario and and the circumstances and types of the errors.
A decorator for test functions should allow a iteration over a data set testing the method.

Outsourcing

Outsource anything outside of the Core Domain to libraries and services vastly accelerating development and the creation of value.

If the problem is in another domain it probably should be outsourced, especially if it is in another technical domain.

Beware of dependency hell by choosing libraries and services with extremely mature API's.

Pull the sources into the repositories as sub-modules and build packages into your own package repository so that the source for the entire system is maintained.

Side Effect Free Functions

As many functions as possible should return a result, and have no other effect upon subsequent calls, or other functions that would alter their outcome. This simplifies analysis, understanding, and eliminates vast numbers of difficult to solve bugs. All functions should strive to be side effect free.

Assertions

Invariants are like probes into the heart of the design and the code. Well stated as single invariants or as Hoare predicate transformers stating the pre and post conditions of the function.

The design or code can be better understood and verified under all scenarios instead of a test that only verifies the assertions of a single scenario.

Simplify Associations

Reduce direction and cardinality complexity of associations with constraints found in deeper understanding of the problem domain.

Stored Procedures

Use stored procedures to abstract Entities and Aggregates from the storage structure. This also abstracts storage quirks from the Technical Core layer.

Stored Procedures enforce locking and return denormalized rows for compound objects.

Factories

Factories are functions or boxes with underlined names that create, update, retrieve instances, or create complex or polymorphic objects.

If the factory is for a object with persistent storage zero args should create, one argument should write the object, and keyword args should query for a existing object. It should also be global so the storage code is centralized in one place.

Third Normal Form

"Integrity"

1. Every property must have a single value
2. Remove duplicates and every property must depend on the key and nothing but the key
3. No transitive dependencies, no field can depend on another non-key field