Philosophy of Code

Michael Mattie

31st March, 2024

Contents

1 Philosophy of Code

Rapid development of effective software requires the knowledge and experience to design, implement, deploy, and iterate on software technology.

Along the way many projects ditch tooling and process for speed, thinking that this is the quickest way from: A -> B. The result is a mock-up at best that has to be rewritten on the investor or customer dime.

At Gauge Security, we translate principles into products. Effective, reliable, and fast.

We use innovations, automations, tooling, experience, and plain hard work to deliver as promised.

2 Experience

Our Design Process is derived from extensive experience working for Fortune 10/100 Corporations:

- ToolBuilders (Static Analysis of C, Large Scale C++)
- OpenEye (Agile, Distributed Version Control)
- Cobalt/ADP (Enterprise Scale Datbase Ingest)
- Disney (Rapid Python Development)
- Comcast (System Integration/Micro-Service Architecture)
- Oracle (Cloud BackPlane/Security/AAA)
- Amazon 1 (Global Network Capacity Mapping and Visulization)
- Amazon 2 (AWS Security for Capital One)

3 System Design

Vision - "Building the Impossible"

System Design is a crucial aspect of building effective and flexible systems that can be modified and enhanced without rewrites of monolithic codebases.

My major influence is the UNIX command line. Utilizing a standard data format highly configurable and narrowly scoped components are composed - making it possible to achieve astounding power, flexibility, and re-use.

Conversely, many poor designs revolve around slapping together a database that is shouldering a massive monolithic app.

Despite monolithic apps being replaced by Micro-Service architecture and Application Integration architecture - huge monoliths are still stood up today. Especially for ”1.0” implementations.

4 System Design - Modern Architecture

”One bulls-eye is luck, three is skill”

Vision is a articulation of goals and a refinement of the application scope using business language. The business case defines the scope, operations, and constraints of the solution.

Being able to build something that resembles the vision, and doing it over and over again requires a process. A method for doing it right every time.

When you need precision you make a ”jig”. A measuring stick, or frame for cutting the pieces or spacing out pieces. Every time you measure you get a different measurement. A jig is the same in every instance.

For example if you are spacing rows for a machine to go down, and the end of the tape has 1/16” play, then after 16 rows your off near a full inch.

Like a jig the software development process frames every aspect of the development and delivery of the software. If the process is ad-hoc, poorly defined, or not followed you can find yourself critically off at the end.

If developers fail to uncover major issues in the design process, if developers make structural mistakes in implementation, if developers don’t have the tooling to reliably push into production - then you face serious anomalies in the timeline.

Any one of these failures can lead to delayed deliveries, penalties, and could ultimately result in termination of projects, reputation damage, or lost clients.

4.1 RUP (Rational Unified Process)

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.

This service architecture allows for rapid iteration within well-defined silos while maintaining the intergrity of the overall system.

Changes in the services have a low impact. Changes in the message passing or service scope ripple throughout the layers, and cause conflicts in the system and the developers.

Each of the services perform staged deliveries, allowing the overall system to be a staged delivery model. This builds and exercises the entire delivery toolchain and the deploy process.

- Inception - make a initial evaluation to determine the cost and value of the project. It should make a business case of the minimal viable delivery definition through essential use cases.
- Elaboration - perform domain modeling and architecting. Produce a integration test suite to shake out the service factoring and the overall 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.

Domain Language is where concepts are identified. If there is a troublesome area use domain language to pinpoint, dissect, and and re-arcticulate the design stumbling blocks.

Designing is the practice of distilling a vision description down into the essential components and features of a successful system. Design's legacy is: eleganance, aesthetic, pragmatic, maintainable software.

4.2 Design

"Weeks of Coding can save hours of Planning"

Simplicity of design is first principles. The insights into the domain produce a model of the problem and a fully conceived solution.

A CASE/DOMAIN iteration process is learning and refinement in nature. It is not a scribbling of ideas, but testing and stretching them to see if they fit the problem.

4.2.1 USE CASE/DOMAIN modelling

USE cases are designed as sequence diagrams showing the interaction between the user, the system, and the problem. The user interacts with the system, and the system interacts with the problem - or DOMAIN.

4.2.2 Roles

System Roles are built as state machines. Each state consists of a set of messages that can be sent from that state. Each response from the system or operation performed by the user is a potential response and/or transition to another state.

In planning roles each role is a board, each state is represented by a ticket, and the messages as sub-tasks. Boards and tickets are for system planning, not time tracking and employee metrics.

4.2.3 Messages

Messages are transitive immutable entities, passed between components and users. They are in a Data Catalog which is a documentation of all the messages in a layer.

4.2.4 Entities

Entities are persistent state with a cohesive, complete, and minimal set of attributes. They are refined by BNF [22] normalization where the narrow and precise scoping of the Entity is used for normalization.

Entities never appear in the layer definitions as they are always encapsulated by services.

4.3 Summary

The design process doesn't have to be slow. If it is slow then it won't be used. Looking at tools like markdown, plantuml, and mindmapping tools, a design can rapidly evolve.

If it's slow your tools are in the way with too much formatting and styling. Design docs can be primitve at first, and styled with fancy tools later for presentation.

5 Implementation

Implementation is not a straight to code path, it involves a second phase of iteration: enriching the layers, services, and operations along cohesive encapsulation lines.

5.1 Service Factoring

"Systemic Seperation of Concerns"

Operations under the same "knows about" topic are gathered into a service that encapsulates the topic. It produces and consumes messages that soley reference entities and operations within it's scope.

In a sequence diagram of the messaging between the services each service has a column.

5.2 Layers

System layers are defined by a data catalog and relationships shared across the components and services. If it comes from the same data catalog it's in the same layer. Different catalog, different layer.

5.2.1 App Layer

The app layer is responsible for all the library, compotent, and service integrations, initialization, error handling, and shutdown.

5.2.2 Domain Layer

The Domain Layer should be a structure encompassing and modelling the full scope of the problem.

The Domain layer is focused on representation, and it's parts are concerned with traversing, structuring, and partitioning the Problem Space.

The Domain layer should fit on everything from a laptop for development, to pyspark clusters for large scale data processing.

The structure of the Domain Layer should represent the real world relationships between the pieces of data.

A good example is the MacOS device model which has representations in a network graph for connectivity and in planes such as power management. Querying the device model is by passing a dictionary of attributes providing encapsulation.

5.2.3 Technical Core

The Technical Layer ties into both the Application Layer and the Domain Layer to provide the Business Logic and Algorithmic capabilities of the system.

6 Principles

Principles are wisdom that when discarded produce a Pyrrhic victory

6.1 Twelve Factor App

- One Code Base in Version Control (This can be decomposed into multiple repostories with advanced tooling capabilities) independent of environments configurations, and dependencies.
- Explicit Dependencies and Dependency Isolation
- Code sharing between repositories is packages. All dependencies of each service are checked in as git submodules. All code in the repository is built into the package for that repository.

  submodule dependencies including executables are static linked down to the OS layer.
- Config Values in Environment Variables. Config values are propagated from the environment bound launcher into environment variables consumed by the application processes.

- Backing Services: All resources are abstracted as config bound componets, local and remote.
- Code and Build, bind environment config and build for release.
- Stateless Processes. All processes contain no locally attached state, all state is written to resources with ACID properties
- Port Binding - no web server or reverse proxy. The app binds to a bare port. No extra components are needed to run it.
- Scale via Processes. Scale horizontally with processes.
- Disposable Processes Make processes starting, stopping, and scaling fast. make them disposable putting state in ACID resources. Death of processes should not impact the system in any significant way.
- Dev/Prod Parity: Keep Dev and Prod in sync so that changes can be rapidly promoted to Prod with confidence.
- Logs - Log to stdout: Log to stdout, use logging services to pick up the stream and make it analyzable.
- Admin via one-off programs and REPL's. Glue together dashboards out of logging services and dashboards.

6.2 Tests as Contracts

Tests as contracts. Tests should reflect actual useful scenarios and not simply exercise coverage. Test the expected behavior of the interfaces on one level, and the performance on another.

To make it organized, and even possible to auto-generate docs from the unit test code - make a test file for each operation being tested. Enumerate the cases in the file.

Documentation should briefly describe what the behavior of the mode, and the circumstances and types of the errors.

6.3 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 with minimal sub-dependencies. Small libraries with narrow scope and functionality should be avoided.

6.4 submodules vs. packages

Pull the sources for outsourcing into the service repositories as git submodules. Build packages and store in your own package repository so that the source, builds, and repeatable builds for the entire system is preserved.

6.5 Side-Effect Free

Side Effect Free Functions: as many functions as possible should return a result, and have no other effect upon subsequent calls, or alter the outcomes of other function.

This simplifies analysis, understanding, and eliminates vast numbers of difficult to solve.

6.6 Assertions

Assertions are Invariants that are like probes into the heart of the design and the code. Invariants should be used primarily in tests. well stated is single invariants or as predicate transformers [24] stating the pre and post conditions of the function.

6.7 Simplify Associations

Simplify Associations: Reduce connections and cardinality complexity of relationships with constraints and layers found in the deeper understanding of the problem domain.

Use Stored Procedures or Object Relational Mappers 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.

6.8 Factories

Factories are for the construction of compound objects, objects with post-construction intialization, or selecting between objects with different class lineage, but the same API.

6.9 Tests as contracts

One of the main reasons why documentation is such a problem is drift, no one notices when the code changes, but the documentation doesn't. Attempts to integrate the two have been "Literate Programmming" by Knuth [23]

The tools however were not time efficient enough due to the emphasis on typesetting. More recently markdown has emerged as a fast way of creating documents.

Now there is a even better way that has evolved in Emacs. It is called org-mode and it allows for code blocks to be mixed with markdown like document syntax.

Not only does it rapidly generate documentation, the code blocks can even be executed inside the org-mode document, or written to files.

This allows for a new paradigm where the tests and the API documentation are the same document. The tests illustrate the API, verify the documentation, and "tangled" into files a test suite is generated.

6.9.1 Structure

The test-suite/API documentation has the structure of a document with a preamble introducing the API. Each operation in the API is a mixture of code and documentation.

Each Operation generates a test-suite file. In each operation test file the CASE's are enumerated exhaustively, testing the code and validating the documentation.

The result is a test-suite, and API documentation in sync.

7 My Readings

Here is list of my most influential sources, with a short description of what they are, or the influence they had on me.

7.1 The 12 Factor App

The 12 Factor app [6] is a seminal document on Architecture and implementation of horizontal scaling Micro-Service Systems. It's lessons are from the blood, sweat, and tears of years - if not decades - of writing scalable and maintainable systems.

7.2 Semantic Versioning

Semantic Versioning [21] is the state-of-the art in release practices for version formatting and the semantics of the version scheme.

It's commentary on release practices is priceless.

7.3 Git Flow

Git is powerful, but does not impose a Workflow. This has lead to a lot of chaos, but has also allowed for a lot of research into the best Workflows for version control.

Git-Flow [2] Is the best of the Workflows and is tooled as ”git-flow” on most systems. The combination of a well thought out, experience driven, powerful paradigm is a huge asset to any project.

7.4 Conventional Commits

Most commit messages arise from a anarchy of practices leading to git logs that are difficult to understand and impossible to automate with tools.

Convential Commits [3] provide a standard for different types of commits and what the types mean. With git flow you can understand the logs easily and also you can use tools to process the logs.

7.5 Introduction to Algorithms

MIT Introduction to Algorithms [4] is the definitive work on the most common algorithms. It is the ten-ton-hammer of algorithms with precise detail and thorough presentation of every algorithm. This belongs on every programmer's shelf.

7.6 Applied Cryptography

Applied Cryptography [18] is the seminal text on cryptography theory, algorithms, and application.

The principles are explained in a precise and lucid manner. Not a daily-driver for most programmers, but as a reference on cryptography it has no peers.

7.7 Design Patterns

Design Patterns [7] are definitely one of the most influental books on programming ever written. It introduces abstract definitions of powerful code mechanisms in a high level description This should be read cover-to-cover many times.

7.8 Domain-driven Design

Domain-driven Design [5] is a foundation of design principles for system design and process. It is a cover-to-cover read.

7.9 Logic in Computer Science

Logic in Computer Science [8] deals with the modeling and reasoning about computer code and systems. This is a powerful book but very dense with predicate logic.

7.10 Structure and Interpretation of Computer Programs

The original MIT intro to CompSci book [1] ss my bible. It's thorough presentation of programming fundamentals in the scheme language makes it a pleasant read.

It is a tour-de-force of fundamentals, and a fascinating treament of both functional and procedural programming.

7.11 The Art of Computer Programming

Quite possibly the most famous series in programming. Written by Donald Knuth, typeset in Tex - a system created to typeset the book correctly, It is possibly the most correct text on programming.

Knuth famously wrote checks to anyone who could find a mistake in the books. The checks were rarely cashed, they were one of the most prized awards in programming culture. The series is four volumes currently

- Vol 1: Fundamental Algorithms [12]
- Vol 2: Seminumerical Algorithms [11]
- Vol 3: Sorting and Searching [12]
- Vol 4: Combinitorial Algorithms [10]

7.12 The Structure of Scientific Revolutions

This classic text [13] by Kuhn seperates revolutionary ideas from incremental progress. It defines revolutionary changes as paradim shifts to new models. This classic pinpoints the tidal shifts in scientific thinking.

7.13 Unix Power Tools

One of the most influential of my books Unix Power Tools [16] . It teaches the command line by examples with as a teaching mechanism.

If you learn by example, and want to deep dive into the command line this is the best book.

7.14 Hackers, Heroes of the Computer Revolution

Steven Levy's [14] ”Hackers” is an amazing presentation of the early MIT years of computer programming, personal computers, and early video game programming.

An easy read, and a good one.

7.15 The Art of Unix Programming

The Art of Unix Programming [17] is a very influential book on designing systems the UNIX way and how to decompose complex behavior into simple parts.

7.16 The Cuckoo's Egg

The Cuckoo's egg [19] was my first introduction into the world of programming and UNIX. It inspired me to become a programmer.

7.17 The Design and Evolution of C++

A lesser known work by Bjarne Stroustrup [20], in this book he discusses the context and the decisions that drove the creation and evolution of C++. A must read for insight into the creative and design process behind software.

7.18 The Design of Every Day Things

The Design of Every Day Things [15] spawned modern inteface design, and the rise of the product designer. A must read for programmers to create intuitive software.

7.19 The Soul of a New Machine

The Soul of a New Machine [9] is a great real world example how a small nimble team using a simple clear vision and design can build a revolutionary product in a very short amount of time.

References

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs, second edition*. MIT Electrical Engineering and Computer Science. MIT Press, 1996. ISBN: 9780262510875. URL: `https://books.google.com/books?id=iL34DwAAQBAJ`.

[2] Atlassian. *Git Flow*. 2024. URL: `https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow` (visited on 03/13/2024).

[3] Conventional Commits. *Conventional Commits*. 2024. URL: `https://www.conventionalcommits.org/en/v1.0.0/` (visited on 03/17/2024).

[4] T.H. Cormen et al. *Introduction to Algorithms, fourth edition*. MIT Press, 2022. ISBN: 9780262367509. URL: `https://books.google.com/books?id=RSMuEAAAQBAJ`.

[5] E. Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. ISBN: 9780321125217. URL: `https://books.google.com/books?id=xColAAPGubgC`.

[6] 12 Factor. *The 12 Factor App*. 2024. URL: `https://12factor.net/`.

[7] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN: 9780321700698. URL: `https://books.google.com/books?id=6oHuKQe3TjQC`.

[8] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004. ISBN: 9781139453059. URL: `https://books.google.com/books?id=eUggAwAAQBAJ`.

[9] T. Kidder. *The Soul of a New Machine*. Modern Library Series. Modern Library, 1997. ISBN: 9780679602613. URL: `https://books.google.com/books?id=oUpGAQAAIAAJ`.

[10] D.E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms*. Pearson Education, 2022. ISBN: 9780137926817. URL: `https://books.google.com/books?id=6tnPEAAAQBAJ`.

[11] D.E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms, Volume 2*. Pearson Education, 2014. ISBN: 9780321635761. URL: `https://books.google.com/books?id=Zu-HAwAAQBAJ`.

[12] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching, Volume 3*. Pearson Education, 1998. ISBN: 9780321635785. URL: `https://books.google.com/books?id=cYULBAAAQBAJ`.

[13] T.S. Kuhn and I. Hacking. *The Structure of Scientific Revolutions*. University of Chicago Press, 2012. ISBN: 9780226458144. URL: `https://books.google.com/books?id=3eP5Y_OOuzwC`.

[14] S. Levy. *Hackers: Heroes of the Computer Revolution - 25th Anniversary Edition*. O'Reilly Media, 2010. ISBN: 9781449393809. URL: `https://books.google.com/books?id=JwKHDwAAQBAJ`.

[15] D. Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, 2013. ISBN: 9780465050659. URL: `https://books.google.com/books?id=qBfRDQAAQBAJ`.

[16] S. Powers. *Unix Power Tools*. Nutshell handbook. O'Reilly Media, 2003. ISBN: 9780596003302. URL: `https://books.google.com/books?id=Xk6THylQxRUC`.

[17] E.S. Raymond. *The Art of UNIX Programming.* Addison-Wesley Professional Computing Series. Pearson Education, 2003. ISBN: 9780132465885. URL: https://books.google.com/books?id=H4q1t-jAcBIC.

[18] B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C.* Wiley, 2017. ISBN: 9781119439028. URL: https://books.google.com/books?id=Ok0nDwAAQBAJ.

[19] C. Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage.* Turtleback, 2000. ISBN: 9781417642625. URL: https://books.google.com/books?id=KxlrPwAACAAJ.

[20] B. Stroustrup. *The Design and Evolution of C++.* Pearson Education, 1994. ISBN: 9780135229477. URL: https://books.google.com/books?id=hS9mDwAAQBAJ.

[21] Semantic Versioning. *Semantic Versioning.* 2024. URL: https://semver.org/.

[22] Wikipedia. *Backus–Naur Form.* 2024. URL: https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form.

[23] Wikipedia. *Literate Programming.* 2024. URL: https://en.wikipedia.org/wiki/Literate_programming.

[24] Wikipedia. *Predicate Transformer Semantics.* 2024. URL: https://en.wikipedia.org/wiki/Predicate_transformer_semantics.