Algorithms and Software Design

Mr. Mattie

12th April, 2025

Contents

1 Software: Design and Implementation

Correct Thinking leads to correct Code!

Algorithms and Software problems, even the ones that seem simple, can be very difficult to design and implement.

Purely technical algorithms frequently have mistakes in them until they have been debugged over a significant period of time. Business Logic in software is unique to the domain and there is little guidance there to help the developer.

The biggest failing is not one of coding, but rather a failure of imagination, a failure to conceive of all the scenarios and aspects of the use of the algorithms.

Not realizing that a list might be empty, missing a subtle aspect of definition that impacts the solution, or miscalculating the complexity of a naive solution - these oversights can wreck an algorithm or system.

We will go through a systematic process that is a discovery processs, finding all the ways the algorithm will be used and stretched. The code is intuitive, or at least tractrable, once all the facets of the problem domain have been discovered.

2 Requirements - Building the Domain Model

State & Clarify – Deep Comprehension Modeling

Domain Modeling is understanding the problem, in the language of the problem.

2.1 Comprehension

Study the given requirements with deep comprehension, looking to find a singular descriptive model, describing in entities and relationships the richness of the problem. Look for all the quirks, ambiguities, and anomolies of the problem.

The clarity of your understanding should reveal and resolve into a cohesive set of entities and relationships in the domain language.

Be very wary of premature optimization. This initial model is for understanding the problem fully, and proving correctness of the computational model that follows.

2.2 Solution

With the domain model, translate the entities into types, and relationships as operations on a type. Ideally the syntax of overloading operators facilitates an intuitive, logical, and well understood way of

using the types.

Designing types, with value semantics expressed in interrelated operations, organized along paradigm lines, build intuitive use and composition.

Types and operations that are intrinsicly interrelated, or tied to shared resources should be put into modules. Modules do not have to be a large collection, in fact a large number of small modules makes it easier to test code and manage complexity to a large degree.

2.3 Structuring

The structuring of the program has a large impact on how easy to develop, test, and maintain. Modules are the basic level of construction. Inside modules interrelated types internally construct and consume each other's types directly.

Modules present their functionality in one of three ways:

2.3.1 Types and Operations

Exported types, operations, and their interactions with each other, are directly consumable. This technique is Layering.

With a schema for types, a paradigm for operations, and syntatic constructs for composition, layering is a powerful API.

This type and paradigm interface has a high level of coupling, and is difficult to test, maintain, and change due to the diffusion of direct consumption throughout the consumer. For these reasons it should not be chosen lightly.

2.3.2 Dependency Inversion [17]

For intra-module coupling that does not rely on the consumer creating API types, Where a Singleton in the module API, or a Singleton that constructs all related types is usable, Dependency Inversion is the preferred approach.

Dependency Inversion is where the code that uses the object does not create types it consumes; instead consumed types are constructed in the orginating module, and the resulting the object is passed to the consumer at construction. This allows the entirety of the API to be hidden behind a abstract interface.

For example, a module for configuration could contain the information needed to construct resource objects such as Database objects, or OS objects.

Those resource objects would use the configuration objects for instantiation, and as a singleton into the resource can be easily injected wherever it is needed.

Instead of orchestrating the construction of a resource type, the consumer would use it through an abstract interface, and would receive the consumed object at construction.

In this paradigm all three modules, the configuration, the resource, and the consumer module are loosely coupled through abstract interfaces.

When this technique is used there is no need for extensive and complicated mocking for testing, use is well defined, configuration and construction is implemeneted in one place, instead of fanning out throughout the consumer.

The code is easily written, maintained, changed, and tested.

2.3.3 Message/Event Passing

System level design should define intermodule communication as message passing protocols. With dedicated API types decoupling the interface from the logic and solving code, the API and logic can evolve idependently.

messages/events should be simlple declarative types with some kind of versioning in how they are named, so type based dispatch can be used in an API object to process the message into the system.

2.4 Testing

Testing solutions is vital as any code of significant complexity is broken as first conceived.

Testing can be challenging and labor intensive when the type and paradigm API is used. You would have to use a number of tests with fixtures (pre constructed sets of types) to test it. Unit Testing isolation techniques and infrastructure is necessary.

Unit Testing is effective, and has been beneficial, but the mass of test code becomes baggage that discourages change, due to the amount of work to update it, especially in a layering API.

With a system that is designed as a large number of small modules, the kind of isolation techniques typical in Unit Testing are not needed, except for tracing a specific issue. It can be mocked through dependency inversion.

Instead all testing as much as possible should be black-box, without any insight to the component (type/module) under test. With a large number of test sets black box can be effective at assuring that the code is correct. Coverage analysis should be used to ensure that the testing is in fact covering most of the code.

Constructing tables of test data keeps the testing code flexible and thorough. Fuzzing is helpful for spotting corner cases that have been missed by pure analytical test cases.

3 SCENARIOS

State & Clarify – CASES and EXPECTATIONS

The Application Layer of the algorithm or system defines how it is used, and what it is expected to do. This is interaction and expectation at a high level of granularity.

3.1 USE CASES - (Contexts & Inputs)

CASES are Contexts and Inputs. Contexts are factors or constraints that shape the case beyond the input that is fed into the algorithm or system. Inputs are events and data that the solver consumes to produce a result in the EXPECTATIONS.

3.1.1 INPUTS?

State & Clarify - Types and Scale

The type, scale, and possible anomolies in the inputs to the algorithm or system have a huge impact on the design. Designing something for one thousand elements is a very different from designing for one million elements. A thousand will fit easily to memory, a million elements is a different design entirely.

3.1.2 RETURN?

State & Clarify - Results - (Entities and Constraints)

Entities of the EXPECTATIONS are the other side of the coin, and a crucial aspect of design. BEHAVIOR cannot be solved correctly without knowing the beginning and the end.

3.1.3 CHANGE?

State & Clarify - Before and After [1]

Sometimes the algorithm must make a scoped and persistent change in the system itself. This is less-desirable from a design and implementation standpoint, but if it is the EXPECTATION then it must be done well.

A good way of providing some formalism to describing state changes is Predicate Transformers [13] by Edsgar Dijkstra that have a pre-assertion, the change, and then a post-assertion of what the state looks like before, changed, and after. This level of formalism is not usually necessary unless you are dealing with complex state change issues like parallelism.

If a change must be made it is best to make the algorithm idempotent, or where repeated calls have the same result. For example: a light button as a toggle will alternate on/off the lights every time it's pressed. This is confusing if you simply want it to turn on, or turn off.

A proper switch instead, will turn the light off every time it is pressed in the off direction, and on when pressed on, no matter how many times it's pressed. That is idempotent.

3.2 EXPECTATIONS

State & Clarify - What is the desired outcome?

EXPECTATIONS and their qualifications are the definition of what correctly solves the CASE. They are what the algorithm should compute, do, or return, with type and scale of results. The qualifications are constraints on the solution such as latency, memory consumption, or resource utilization.

3.3 BEHAVIOR

State & Clarify

BEHAVIOR is the business logic and core logic, that from lead from inputs, events and data, to producing the EXPECTATIONS. It is vital to clarify the behavior and make sure it covers all the richness and facets of not only the inputs, but the outputs and changes.

4 Sketch the Code

Sketch the code, or module in functions and loops, with comments on purpose and O-notation complexity

1. **Initialize**: establish a return value, empty containers over nulls
2. **Terminate**: determine the base case. When is it done?
3. **First, Common, Last Cases**: The basic sequence of the algorithm
4. **Corner**: cases
5. **Input Validation**: events, values, completeness, and ranges
6. **State**: initialize, update, delete [1], and lifetime.

5 Design (Iteration)

5.1 Refine Types

Minimze Semantics of Types, and define operations in paradigm concepts

Define types as constant or mutable that have essential cohesion, where their definition of cohesion is perfectly minimal, in that they can only be defined with their set of interrelated properties, but have no properties that are not intrinsic to the type value semantics.

5.2 Isolate Operations

Maximize Idempotent side-effect free operations [1]

Breakdown interaction of types into paradigm derived operations, and try and maximize side-effect free functions. Where there is state handle it carefully defining the entire life-cycle of the state in entities.

5.3 Build Modules

Sets of interrelated types and operations that share resources are Modules

Modules are are interrelated types that construct each other, and share resources. If a type can stand independently it belongs elsewhere.

5.4 Message Passing (API)

APIs are message passing between functionally isolated components (API)

API's pass declarative and constant messages/events between modules that are described in protocols and modeled as seqeuence diagrams.

5.5 Paradigm

Solution Comprehension

Paradigm is what model best describes the problem (dynamic greedy, lazy, streams, Relational, divide and conquer) and most efficiently produces an answer.

Spot check the paradigm against the CASES to see if it adequately describes the problem. Find the right paradigm.

5.5.1 Recursion

$$\theta(\log_n) \tag{1}$$

Recursion is elegant and compact. In languages that support it, it simplifies and strips the implementation down to the core logic.

1. recurrence

   Distill the problem down into a solution that can be applied to all the elements.

2. termination

   Define the base case or **termination** as return of the solution that unwinds the recursion.

5.5.2 Divide & Conquer [5]

$$\theta(n * \log_n) \tag{2}$$

Divide and Conquer is a technique where the problem is dived into parts, each part is solved, and then the sub-solutions are combined into the complete solution.

Divide and conquer is also a natural fit with parallel implementations.

1. Decide the granularity of the division, divide the problem into n/x parts.
2. Solve the sub-problem. The reduced scale of n reduces the complexity or run time of the solution. [6]
3. Combine the solutions for the final solution

5.5.3 Dynamic

Dynamic Programming uses a technique of caching answers to frequently computed problems.

Memoization [7] is a powerful technique and in Python the "functools" package has a LRU [12]

5.5.4 Linguistic (DSL)

DSL stands for Domain Specific Languages. Thes can be simple declarative language processors, or full blown domain specific languages like "R" [15]. They can be used to define complex problems, and through the language implement a powerful and flexible solver.

5.5.5 Query

Query Languages like SQL can go beyond transactional into the space of analytical queries either providing processing of data, or even computations such as "GROUP BY" and MIN and MAX in SQL [16].

The underlying model behind relational databases is the Relational Algebra [4]

5.5.6 Logic

Logic systems are basically rule systems like Prolog [14] They are used in mathematical and logic applications. Their solution finding approach can also be useful in solving difficult problems like cross-wiring network links for redundancy and expert systems.

5.5.7 Single Pass

Single pass approaches are significant when the data set is so large it cannot be contained in memory. These kinds of problems are becoming more important as the size of data in general skyrockets.

5.5.8 Multi-Pass

Sometimes huge gains can be made by making multiple passes. This is basically a variant on Dynamic Programming. Database Indexes are a good example, when the data queried be found quickly in the index instead of a full table scan.

Sorting ahead of time makes possible binary searches or tree alogorithms for searching. If the search is executed many more times than the routine to maintain the order of the index, a massive performance increase can be realized.

5.5.9 Pre-Compute

Pre-Computing unlike multi-pass where the complete problem set is traversed, is instead the compilation of tables that are expensive to compute.

In the early days of computing the computation of sine/cosine and other graphic operations were prohibitely expensive.

Since the answers were a small table pre-computing the equations greatly sped up programs. Bitmaps were even compiled to machine code for faster rendering.

5.5.10 Dynamic Programming

Applied to recursion is (descent + memoization) recursively can be no cycles in the DAG of the recursion, or it will get into an infinite loop. It is fundamentally a brute force approach, good for computing min/max style answers.

5.5.11 Greedy Programming

Greedy algorithms, like the parser compiler packer function I wrote in my Emacs Parser Compiler used a greedy technique with packing to maximally fill functions with code [11].

5.5.12 Lazy Programming

When the computation may not be needed or when the problem cannot fit into memory it can be lazy loaded, or lazy computed. Here the sequence is produced on demand through a generator function with a internal state that is updated when a value is produced, streaming values from a compact single value generator.

5.5.13 Parallel Programming

Parallel programming [8] is a technique implemented in hardware with things like hyperthreading and multiple cores. Even basic functions like add instructions can be implemented in parallel.

Fundamentally parallel algorithms [9] exploit the ability of systems and software to execute two or more pieces of code simultaneously.

If the problem can be partitioned into seperate tasks, or ultimately partitioned and solved along the lines of the divide and conquer class of algorithms, massive speed ups are possible.

Fundamentally parts that are readers will always ”block” or stop parallel execution because they cannot proceed without the values to compute their next step. Writers do not need to block necessarily, but to maintain integrity they would.

When a program needs to synchronize to a single execution the work being done this serialized section is called a ”critical section”.

I won’t go into these vast details except to say that protecting value intergrity between threads and implementing critical sections is very complicated and error prone. The common model is to implement integrity in a database and use concurrent processes that don’t share memory.

5.5.14 Streams

Streams [2] are a finite sequence of discrete elements of the same type processed in a linear sequence of operations. What makes streams unique is that all of the types are consumers of the same stream type and are producers as well allowing them to be chained.

6 Data Structures

6.1 Array

Typed and indexed they are extremely fast with O(1) read/write for any element. Insert is very slow as the array elements have to be copied to make room for each insertion. The equal cost of access to any element makes algorithms like binary search, and some sorting algorithms possible.

6.2 List

Single or Double Linked lists have efficient inserts but perform poorly in most cases.

Counting length or adding to end is $\theta(n)$

6.3 Trees

Good for storing hierarchal data and a natural fit for recursive algorithms, trees require only $\theta \log_n$ to find an element.

Performance is maintained only when the tree is balanced, re-balancing on insert can be an expensive operation.

6.4 Stack/LIFO

Last In First Out

Stacks are an excellent structure for back-tracking problems. They are LIFO, or Last In First Out. They can be used as a substitute for recursion, and generally for back-tracking.

6.5 QUEUE FIFO

First In First Out

Good for processing in chronological order. It can also be used for a breadth traversal of a tree.

6.6 Hashes

A bread and butter data structure used pervasively to look up non-integer keys in $\theta(1)$ complexity.

A bread and butter data structure used pervasively to look up non-integer keys in $\theta(1)$ complexity.

7 Competitive Algorithm Coding

Here is a short condescend set of principles for competitive algorithm coding.

7.1 Redefine the problem in comments.

Carefully restate in comments the problem description in the mechanical terms of the problem statement, and the definition of a solution.

7.2 Find the simple solution

Taking care to understand the O Complexity of the problem as basically stated in comments find the difficult part of the challenge, which is usually some kind of combinitorial complexity.

7.3 Devise a fast solution

Once the fast solution is devised you can proceed to implementation.

7.4 Sketch the Code

Sketch the code, or module in functions and loops, with comments on purpose and O-notation complexity

1. **Initialize**: establish a return value, empty containers over nulls
2. **Terminate**: determine the base case. When is it done?
3. **First, Common, Last Cases**: The basic sequence of the algorithm
4. **Corner**: cases
5. **Input Validation**: events, values, completeness, and ranges
6. **State**: initialize, update, delete [1], and lifetime.

7.5 Test/Debug

Test and debug with print statements, never delete a print statement, just comment them out.

References

[1] H. Abelson, G.J. Sussman, and J. Sussman. In: *Structure and Interpretation of Computer Programs, second edition*. MIT Electrical Engineering and Computer Science. Cost of Assignment, Mutable State. MIT Press, 1996, pp. 175–178. ISBN: 9780262510875. URL: https://books.google.com/books?id=iL34DwAAQBAJ.

[2] H. Abelson, G.J. Sussman, and J. Sussman. In: *Structure and Interpretation of Computer Programs, second edition*. MIT Electrical Engineering and Computer Science. Stream Model. MIT Press, 1996, pp. 243–275. ISBN: 9780262510875. URL: https://books.google.com/books?id=iL34DwAAQBAJ.

[3] 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.

[4] Edgar F Codd. "A Relational Model of Large Shared Data Banks (1970)". In: (2021).

[5] T.H. Cormen et al. In: *Introduction to Algorithms, fourth edition*. MIT Press, 2022, pp. 77–125. ISBN: 9780262367509. URL: https://books.google.com/books?id=RSMuEAAAQBAJ.

[6] T.H. Cormen et al. In: *Introduction to Algorithms, fourth edition*. MIT Press, 2022, pp. 91–115. ISBN: 9780262367509. URL: https://books.google.com/books?id=RSMuEAAAQBAJ.

[7] T.H. Cormen et al. In: *Introduction to Algorithms, fourth edition*. MIT Press, 2022, pp. 391–399. ISBN: 9780262367509. URL: https://books.google.com/books?id=RSMuEAAAQBAJ.

[8] T.H. Cormen et al. In: *Introduction to Algorithms, fourth edition*. MIT Press, 2022, p. 749. ISBN: 9780262367509. URL: https://books.google.com/books?id=RSMuEAAAQBAJ.

[9] T.H. Cormen et al. In: *Introduction to Algorithms, fourth edition*. MIT Press, 2022, pp. 803–806. ISBN: 9780262367509. URL: https://books.google.com/books?id=RSMuEAAAQBAJ.

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

[11] Michael Mattie. *parser.el*. 2008. URL: https://github.com/bitcathedrals/grail/blob/main/archive/parser.el.

[12] promto-c. *Python LRU Cachce Usage Guide*. 2025. URL: https://gist.github.com/promto-c/04b91026dd66adea9e14346ee79bb3b8 (visited on 04/10/2025).

[13] WikiPedia. *Predicate Transformer Semantics*. 2025. URL: https://en.wikipedia.org/wiki/Predicate_transformer_semantics (visited on 04/10/2025).
[14] Wikipedia. *Prolog*. 2024. URL: https://en.wikipedia.org/wiki/Prolog.
[15] Wikipedia. *R Programming Language*. 2024. URL: https://en.wikipedia.org/wiki/R_(programming_language).
[16] Wikipedia. *SQL*. 2024. URL: https://en.wikipedia.org/wiki/SQL.
[17] wikipedia. *Dependency Inversion Priciple*. 2025. URL: https://en.wikipedia.org/wiki/Dependency_inversion_principle (visited on 04/11/2025).