Modern C++ Programming

23. Software Design I [DRAFT]

Basic Concepts

Federico Busato

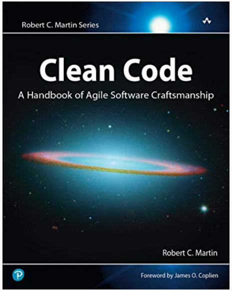
2024-11-05

Table of Contents

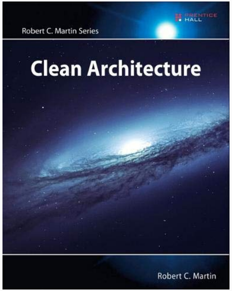
Table of Contents

Table of Contents

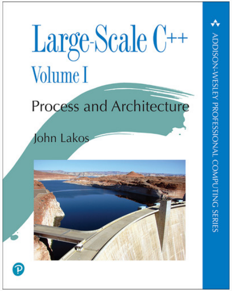
Books and References



Clean Code: A Handbook of Agile Software Craftsmanship
Robert C. Martin, 2008

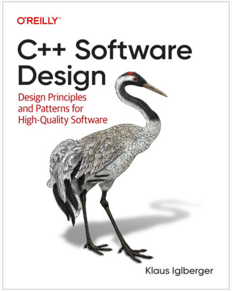


Clean Architecture
Robert C. Martin, 2017



Large-Scale C++ Volume I: Process and Architecture

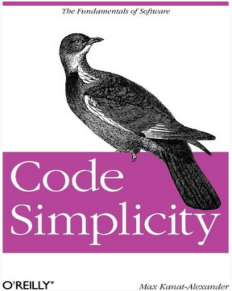
J. Lakos, 2021



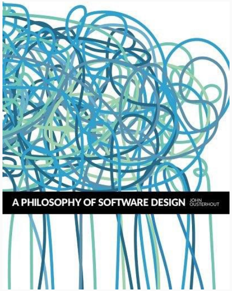
C++ Software Design

K. Iglberger, 2022

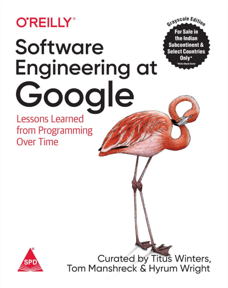
Books 3/3



Code Simplicity
M. Kanat-Alexander, 2012



A Philosophy of Software Design (2nd)
J. Ousterhout, 2021



Software Engineering at Google: Lessons Learned from Programming over Time
T. Winters, 2020
`(download link)`

Basic Concepts

Abstraction, Interface, Module, and Class Invariant

An **abstraction** is the process of *generalizing relevant information and behavior* (semantics) from concrete details

An **interface** is a communication point that allows iterations between users and the system. It aims to *standardize* and *simplify* the use of programs

A **module** is a software component that provides a specific functionality. Common examples are classes, files, and libraries

> *"In modular programming, each **module** provides an **abstraction** in form of its **interface**"*
>
> – **John Ousterhout**, *A Philosophy of Software Design*

Quotes

*"Most modules have more users than developers, so it is better for the developers to suffer than the users... **it is more important for a module to have a simple interface than a simple implementation**"*

– **John Ousterhout**, *A Philosophy of Software Design*

*"The key to **designing abstractions** is to understand what is important, and to look for designs that **minimize the amount of information that is important**"*

– **John Ousterhout**, *A Philosophy of Software Design*

Class Invariant

A **class invariant** (or **type invariant**) is a *property* of an object which remains unchanged after operations or transformations. In other words, *a set of conditions that hold throughout its life*. A *class invariant* constrains the object state and describes its behavior

Software Design Principles

Separation of Concern 1/2

"<u>Separation of concern</u>" suggests to organize software in **modules**, each of which address a separate "concern" or functionality

Benefits of a modular design includes

- *Decrease cognitive load*. Small consistent parts are easier to understand than the whole system in its entirety
- *Help code maintainability*. Fewer or no dependencies allow to focus on smaller pieces of code, isolate potential bugs, and minimize the impact of changes
- *Independent development*

Modular design can be achieved both with *vertical* and *horizontal* organization, i.e. layers of abstractions or functionalities at the same level

Separation of Concern 2/2

> *"The most fundamental problem in computer science is* ***problem decomposition****: how to take a complex problem and divide it up into pieces that can be solved independently"*
>
> – **John Ousterhout**, *A Philosophy of Software Design*

> *"We want to design components that are self-contained: independent, and with a single, well-defined purpose"*
>
> – **Andy Hunt**, *The Pragmatic Programmer*

Low Coupling, High Cohesion

Cohesion *refers to the degree to which the elements inside a module belong together*. In other words, the code that changes together, stays together.

See also the *Single Responsibility Principle*

Coupling refers to the degree of interdependence between software modules. In other words, how a modification in one module affects changes in other modules

The **Low Coupling, High Cohesion** principle suggests to minimize dependencies and keep together code that is part of the same functionality

Encapsulation and Information Hiding

Encapsulation *refers to grouping together related data and methods that operate on the data*. It allows to present a consistent interface that is independent of its internal implementation

Encapsulation is usually associated with the concept of **information hiding** that prevents

- Exposing implementation details
- Violating *class invariant* maintained by the methods

It also provides freedom for the internal implementations

Encapsulation and information hiding are common paradigms to achieve *software modularity*

Problem Decomposition

"Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces"

– **James C. Dehnert and Alexander Stepanov**

Fundamentals of Generic Programming

Code reuse

"Code reuse is the Holy Grail of Software Engineering"
– **Douglas Crockford**, *Developer of the JavaScript language*

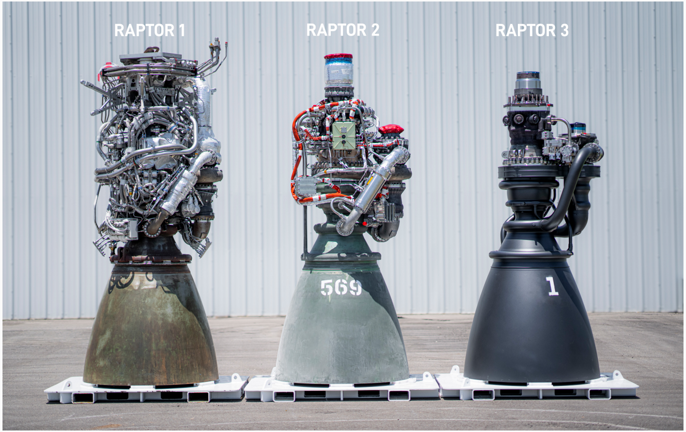
Software Complexity

Technical Debt

> *"Technical debt is most often caused not so much be developers taking shortcuts, but rather by management who pushes velocity over quality, features over simplicity"*
>
> – **Grady Booch**, *UML/Design Pattern*

Technical Debt

“Simplicity is the ultimate sophistication”



The SOLID Design Principles

Class Design

The Class Interface Principle

The Interface Principle

For a class `X`, all functions, including free functions, that both

- "mention" `X`, and
- are "supplied with" `X`

are logically part of `X`, because they form part of the interface of `X`

If you put a class into a namespace, be sure to put all helper functions and operators into the same namespace too

---

`Using namespaces effectively`

`What's In a Class? - The Interface Principle`

Why Prefer Non-Member Functions

Encapsulation: *Non-member functions* guarantee to preserve the class invariant as they can only call public methods, protecting the class state by definition.

Non-member functions helps to keep the class smaller and simpler $\rightarrow$ easier to maintain and safer

Member functions induce **coupling** forcing the dependency from the `this` pointer.

Member functions can be split or organized in several other functions, worsening the problem. Such methods are forced to perform actions that are only specific to such class. On the contrary, non-member function favor generic code and can be potentially reused across the program

Why Prefer Non-Member Functions

Cohesion/Single Responsibility Principle *Member functions* can perform actions that are not strictly required by the class, bloating its semantics

Open-Close Principle *Non-member functions* improve the flexibility and extensibility of classes by extending its functionality but without

Member Functions vs. Free Functions

> *"If you're writing a function that can be implemented as either a member or as a non-friend non-member, you should prefer to implement it as a non-member function. That decision increases class encapsulation. When you think encapsulation, you should think non-member functions"*
>
> – **Scott Meyers**, *Effective C++*

- `https://workat.tech/machine-coding/tutorial/design-good-functions-classes-clean-code-86h68awn9c7q`
- `Prefer nonmember, nonfriends?`
- `Monoliths "Unstrung",`
- `How Non-Member Functions Improve Encapsulation`
- `C++ Core Guidelines - C.4: Make a function a member only if it needs direct access to the representation of a class`
- `Functions Want To Be Free, David Stone, CppNow15`
- `Free your functions!,` *`Klaus Iglberger`*`, Meeting C++ 2017`

Member Functions

Functions that must be *member* (C++ standard):

- Constructors, destructor, e.g. `A()`, `~A()`
- Assignment operators, e.g. `operator=(const A&)`
- Subscript operators, `operator[]()`
- Arrow operators, `operator->()`
- Conversion operators, `operator B()`
- Function call operator, `operator()`
- Virtual functions, `virtual f()`

Member Functions

Functions <u>strongly suggested</u> being *member*:

- **Unary operators** because they don't interact with other entities
  - Member access operators: dereferencing `*a`, address-of `&a`
  - Increment, decrement operators: `a++` `--a`
- Any **method that preserves**
  - **const correctness**, e.g. pointer access
  - **object initialization state**, e.g. a variable that cannot be changed externally after initialization (invariant)

Functions <u>suggested</u> being member:

- In general, **compound operators** are expressed by updating private data members `operator+=(T, T)`, `operator|=(T, T)`, etc.

Non-Member Functions

Functions that must be *non-member* (C++ standard):

- Stream extraction and insertion `<<`, `>>`

Functions that are strongly suggested being *non-member*:

- Binary operators to maintain symmetry, see also "Implicit conversion and overloading"
  `operator+(T, T)`, `operator|(T, T)`, etc.
- `Template functions within a class template`
  Otherwise, it requires an additional `template` keyword when calling the function (see *dependent typename*) → verbose, error-prone

---

`Effective C++ item 24: Declare Non-member Functions When Type Conversions Should Apply to All Parameters`

Member Functions vs. Free Functions - Summary

More in general, *member functions* should be used only to **preserve the invariant properties** of a class and **cannot be efficiency implemented in terms of other public methods**

All other functions are **suggested** to be ***free-functions***

Some examples: `std::begin()/std::end()` C++14, `std::size()` C++17

Namespace Functions vs. Class `static` Methods

Namespace functions:

- Namespace can be extended anywhere (without control)
- Namespace specifier can be avoided with the keyword `using`

Class + `static` methods:

- Can interact only with static data members
- `struct/class` cannot be extended outside their declarations

→ `static` methods should define operations strictly related to an object state (*statefull*)

→ otherwise `namespace` should be preferred (*stateless*)

BLAS GEMM Case Study

BLAS GEMM

GEneralized **M**atrix-**M**atrix product API provided by **B**asic **L**inear **Al**gebra **S**ubroutine standard is one of the most used function in scientific computing and artifical intelligence

The API is defined in C as follow: $C = \alpha op(A) * op(B) + \beta C$

```
ErrorEnum sgemm(int m, int n, int k,
                OperationEnum opA,
                OperationEnum opB,
                float  alpha,
                float* a,
                int    lda,
                float* b,
                int    ldb,
                float  beta,
                float* c,
                int    ldc);
```

BLAS GEMM - Comprehension Problems

- `m`, `n`, `k` **describe the shapes of** `A`, `B`, `C` in a non-intuitive way. Except domain-expert, users prefer providing the number of rows and columns as matrix properties, not GEMM problem properties
- **Privatization of the return channel** for providing errors
- **Errors expressed with enumerators**. Need additional API to get a description of the error meaning
- **Domain-specific cryptic name**. e.g. `zgemm`: generalized matrix-matrix multiplication with double-precision complex type
- **The data type on which the function operates is encoded in the name itself** `zgemm` $\rightarrow$ any new combination of data types requires a new name.

BLAS GEMM - Flexibility Problems

- `A`, `B`, `C` matrices could have different types
- The compute type, namely the type of intermediate operations, could be different from the matrices. This is also known as *mixed-precision* computation
- Batched computation, namely having multiple input/output matrices, is not supported
- The API is **state-less** $\rightarrow$ preprocessing steps for optimization or additional properties (e.g. different algorithms) cannot be expressed
- Matrix sizes can be greater than `int` ($2^{31} - 1$), specially on distributed systems
- Even if we perform computations with relative small matrices, the strides, e.g. `row * lda` could be larger than `int` ($2^{31} - 1$)

BLAS GEMM - Flexibility Problems

- `alpha/beta` could have a different type from matrix types
- `alpha/beta` are typically pointers on accelerators (e.g. GPU) to allow asynchronous computation
- The underline memory layout is implicit (column-major). Row-major and other layouts are not supported
- `C` is both input and output. It is more flexible to decouple `C` and add another parameter for the output `D`
- Doesn't have an *execution policy* which describes *where* (host, device) and *how* (sequential, parallel, vectorized, etc.)

BLAS GEMM - Flexibility Problems

- Doesn't have a *memory resource* which provides a mechanism to manage internal memory
- *Memory alignment* is known only at run-time
- It is not possible to optimize the execution with compile-time matrix sizes

Most of all these points have been addressed by the **`std::linalg`** ↗ *proposal*

Owning Objects and Views

Objects vs. View

Object

An **object** is a representation of a *concrete entity* as a *value in memory*

Resource-owning object

Resource-owning object refers to RAII paradigm which ties resources to object lifetime

example: `std::vector`, `std::string`

View

A **view** acts as a *non-owning reference* and does not manage the storage that it refers to. Lifetime management is up to the user

example: `std::span`, `std::mdspan`, `std::string_view`

Objects vs. View

- lack ownership
- short-lived
- generally appear only in function parameters
- generally cannot be stored in data structures
- generally cannot be returned safely from functions (no ownership semantics)

Objects vs. View

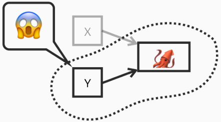
```
#include <string>
#include <string_view>

std::string f() { return "abc"; }

void g(std::string_view sv) {}

std::string_view x = f(); // memory leak
g(f());                   // memory leak
```

Regular, Revisited, *Victor Ciura*, CppCon23

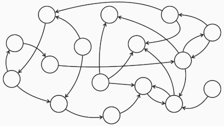
Value vs. Reference Semantic

Technical Debt: engineering cost: more coupled, more rigid, fragile (multiple references)

Spooky action: different references see an implicitly shared object. Modification to a reference affects the other ones



Incidental algorithms: emerges from a composition of locally defined behaviors and with no explicit encoding in the program. References are connection between dynamic objects



Visibility broken invariant: a modification to a reference can have a chain of actions that reflects to the original object, breaking the visibility of an action

Race conditions: spooky action between different threads

`Values - Safety, Regularity, Independence, and the Future of Programming`, *Dave Abrahams*, CppCon22

Surprise mutation: invisible coupling introduced by involuntary dependencies

```
void offset(int& x, const int& delta) { x += delta;}

int a = 3;
offset(a, a); // x=6, delta=6
offset(a, a); // x=12, delta=12
```

Unsafe operations mutation: A safe operation cannot cause undefined behavior

```
int a  = 3;
int b& = a;
a = b++;
```

see also, strict aliasing violation

`Property Models: From Incidental Algorithms to Reusable Components`, *Jarvi et al*, GPCE'08

Value Semantic

Regularity: $x = x$; $x == y \rightarrow y == x$; $x == \text{copy}(x)$; $x = y \iff x = \text{copy}(x)$

regular data type properties: copying, equality, hashing, comparison, assignment, serialization, differentiation

composition of value type is a value type

Independence: local and thread-safe

value semantic in C++

- pass-by-value gives callee an independent value
- a return value is independent in the caller
- a rvalue is independent

Global Variables

Global Variables

`The Problems with Global Variables`