CS 247

Software Engineering Principles

Chapter 5

Interface Specifications

Victoria Sakhnini

Table of Contents

Introduction

Chapter 4 taught us how to physically separate a program into modules: header files for the interface, source files for the implementation, and the build process that fuses them. But a header file describes a module. It tells the compiler the name and type of every public operation, and nothing else. It does not tell the client what those operations *do*. Two functions with identical signatures can have entirely different behaviour, and the C++ type system is powerless to tell them apart.

Consider these two functions:

```
int find(const std::vector<int>& v, int val);

int locate(const std::vector<int>& v, int val);
```

Same signature. Are they the same function? Should you use one in a loop and the other in a binary search? Does either of them work on an empty vector? Does either return -1 or `v.size()` when the value is absent, or do they crash, or hang forever? The signature does not say. The function name hints, but hints are not contracts.

This chapter is about closing that gap with a **specification**: a precise written contract between the implementer of a module and its clients. We will learn the CS 247 specification form (the // requires, // modifies, // ensures, // returns, // throws clauses), how to apply it to functions and classes, and the deeper conceptual content the form encodes: the *specificand set* of valid implementations, how to compare specifications for strength, and when checking the preconditions is a kindness to clients versus a needless cost.

By the end of the chapter, you should be able to read a specification and tell whether a given implementation conforms to it; write a specification that is restrictive enough to forbid bad implementations and general enough to allow good ones; and recognize the design smells of specifications gone wrong (vague language, missing modifies clauses, preconditions that are impossible to check).

Modules and Interfaces

A **module** is a software component that encapsulates a design decision, a function, a class, a package, a library, or a microservice. The unit can be as small as a single function or as large as a multi-file subsystem. The defining property is that it has a clear inside (the implementation) and a clear outside (the boundary across which clients interact with it).

An **interface** is the abstract public description of a module. It is everything a client needs to know to use the module correctly, and *nothing more*. The interface serves two purposes that pull in the same direction:

- **Information hiding.** The interface withholds implementation details from clients. Clients cannot accidentally depend on them, so the implementer is free to change them without breaking client code.

- **Reducing information overload.** "Information overload is what happens when there's more stuff thrown at the reader/viewer than their brain can process at once." A good interface compresses what the client needs to know into something they can actually keep in their head.

Best practice. An interface in our discipline consists of two parts:

- A **signature**, which specifies the *syntactic* requirements: the types and names of operations, their parameters, and their return values. The signature is what the C++ compiler enforces.
- A **specification**, which describes the *behaviour*: what each operation does, what it assumes about its inputs, and what it promises about its outputs. The specification is what humans (and, in some disciplines, formal verifiers) enforce.

Signatures live in header files; specifications live in comments. Both are part of the interface; neither suffices on its own.

Benefits of Modularity

There are three categories of benefit, each of which becomes concrete only once you have actually worked on a multi-person codebase:

- **Simplifying development.** Decomposing software into components (e.g., classes) allows each component to be considered and developed in isolation. You can reason about one class without holding the entire program in your head.
- **Facilitating change.** Modularity *consolidates* the information you need to know about other modules into their interfaces. As long as (1) your component depends only on the interfaces of other components, and (2) those interfaces remain stable, the implementations behind them can change without forcing you to change.
- **Supporting teamwork.** Interfaces draw a boundary between each component's implementation and its client code. Different developers can own different sides of the boundary, working in parallel, with the interface as the only thing they have to agree on.

Every one of these benefits depends on the interface actually capturing what the module does. A signature alone is too thin: a client cannot "depend only on the interface" if the interface does not tell them what the operations do. That is the gap a specification fills.

The Purpose of an Interface

Restating the same idea from the client's point of view: an interface is the *public persona* of a module. It provides enough information for the client programmer to use the module effectively, without necessarily revealing all of the module's design details. From this principle, two practical consequences follow:

- **Encapsulation and information hiding.** The interface exposes *services*, what the module does, and hides the *mechanism* by which those services are delivered. Mechanism is the implementer's business, and changing the mechanism without changing the interface is invisible to the client.

- **Reduced information overload.** A class with thirty private helper functions exposes only the half-dozen public operations its clients actually need. The other twenty-four are noise from the client's perspective and would only get in the way of understanding.

An interface answers two questions: "What services does this software unit provide?" and "How do other units access those services?" Anything not needed to answer those two questions belongs inside the module, not in its interface.

Interface Specification as a Contract

The defining metaphor of this chapter is that an interface specification is a **contract** between the module's provider and the client programmer.

Obligations, Benefits, and Invariants

In everyday life, a contract is what is written when one party commissions another to provide a service or product. Each party gets benefits and incurs obligations. The carpenter agrees to build a deck by Friday for $3000; the homeowner agrees to pay $3000 by Friday for a deck. Both sides accept obligations because both sides expect benefits. The contract exists to make those obligations and benefits explicit, so that if the deck is late or the cheque bounces, it is clear who has failed whom.

Software contracts work the same way.

> 💡 **Tip:** *A contract is an agreement between a module's provider and its user. The contract covers mutual obligations (the preconditions), benefits (the postconditions), and consistency constraints (called invariants). An interface specification defines what the unit requires, in the way of services or assumptions, for it to work correctly… as well as what it provides to its environment.*

Three terms there are worth pulling out:

- **Preconditions.** Obligations on the *client*, what must be true before the operation is called. The client is responsible for ensuring these hold.
- **Postconditions.** Benefits to the *client*: what the operation guarantees will hold after it returns, provided the preconditions are met. The implementer is responsible for delivering these.
- **Invariants.** Consistency constraints that hold *at all observable moments*, before each operation, after each operation, and (for class invariants) between operations as well. We will return to invariants in detail when we discuss class design later in the course.

An interface specification has two uses, depending on which side of the boundary you stand on:

- It **documents the design of a future module**. Before any code is written, the specification is the agreement that defines what "working" will mean. Implementers know what they are committing to; clients know what they can assume.

- It **documents the correct usage of an existing module**. After the code exists, the specification tells anyone who reads it what assumptions they may make and what obligations they have to meet.

Both uses are the same agreement viewed from different sides of time.

The CS 247 Specification Form

CS 247 uses a particular informal notation for writing specifications. The notation is informal enough to read like comments, but disciplined enough that you can compare specifications, check implementations against them, and reason about which is stronger. Every clause begins with a recognized keyword in a comment, and every clause has a fixed meaning.

Preconditions: the requires Clause

A **precondition** is a constraint that must hold *before* the method is called. If the precondition is not met, the implementation owes the client nothing; the contract is silent on what the function may do. (We will explore the practical consequences of this freedom later in the chapter.)

Preconditions are written with the // requires: keyword:

```
// requires:  necessary assumptions about the program state
```

There is only ever one requires clause per function, but it can list multiple conditions joined informally. Examples:

```
// requires:  v is not empty
// requires:  v is sorted in non-decreasing order
// requires:  denom != 0
// requires:  0 <= i && i < v.size()
```

Preconditions are expressed in terms of **public state**, things the client can see and reason about. Phrases like "the cache must be primed" or "the internal buffer must not be flushed" are not legitimate preconditions because the client has no way to know or control them. If a precondition cannot be stated without referring to a private state, the design has leaked.

Postconditions: modifies, throws, ensures, returns

A **postcondition** is a constraint that holds *after* the method returns, on the assumption that the preconditions held when it was called. CS 247 splits the postcondition into four clauses, each with a clear focus:

Clause	Records	Example
`// modifies:`	Which objects or variables may be changed by the method?	`// modifies: v`
`// throws:`	Which exceptions the method may throw, and under what conditions.	`// throws: DivByZero if denom == 0`
`// ensures:`	Guaranteed side effects on modified objects (i.e., the state after the call).	`// ensures: v is sorted`
`// returns:`	Description of the return value.	`// returns: smallest element of v`

Two important rules govern these clauses:

1. **All expressions are over the public state.** Just like preconditions, the postconditions are written in terms of values the client can observe. You may not write `// ensures: this->cache_ is empty` because `cache_` is private. Write the observable consequence instead.
2. **If a clause is silent, the method makes no commitment about that thing.** In particular, omitting modifies is *not* the same as saying "modifies nothing", but in practice, we treat it as such, because the absence of a modifies clause is the implementer's signal to the client that the method is pure. Be explicit if you want a pure method to be known as such.

Two notational conveniences appear in our specifications:

- x@pre denotes the value of x *at the moment the method was called*. This lets postconditions refer to both the before-state and the after-state simultaneously, which is essential for any operation whose effect is described as "the input had this property; the output has that property."
- Plain x in a postcondition refers to the value of x *after* the call. So `v[i] = newElem` in an ensures clause means "after the call, `v[i]` equals `newElem`".
-

⚠ **Warning:** The *requires* clause is *not* a place to dump conditions for thrown exceptions. If the function throws when denom == 0, then denom == 0 is *not* a precondition, the function defines behaviour in that case (it throws), so the client is allowed to call it that way. The precondition should *exclude* cases that the function handles via exceptions. Mixing the two is a common beginner mistake.

Worked Examples

Example 1: sumVector

Start with the easiest case: a function that does not modify anything and has no preconditions worth stating.

```
// returns: the sum of vector elements
int sumVector(const std::vector<int>& vect) {
    int sum = 0;
    for (int i = 0; i < vect.size(); i++) {
        sum += vect[i];
    }
    return sum;
}
```

One-line specification. No requires (the sum of an empty vector is unambiguously zero), no modifies (the parameter is const), no throws, no ensures beyond what returns already implies. This is the steady state for a pure function: the entire specification fits in a single comment.

> ❓ **Question:** Is "the sum of vector elements" actually a complete description?
>
> It is complete enough that no reasonable implementer would get it wrong. "Sum" for integers is unambiguous (set-theoretic, no rounding); "of vector elements" is unambiguous (all of them, in any order, exactly once). For more subtle types, floating-point numbers, for which addition is not associative, we might need to tighten the wording. But for int, this is fine.

Example 2: replace

Now, a function that mutates its argument. The body alone:

```
// replaces element in vector; returns position of new element
int replace(std::vector<int>& vect, int oldElem, int newElem) {
    for (int i = 0; i < vect.size(); i++) {
        if (vect[i] == oldElem) {
            vect[i] = newElem;
            return i;
        }
    }
}
```

The one-line comment looks innocuous, but it leaves several questions unanswered. What if `oldElem` is not in the vector? The loop falls through the bottom of the function without a return, undefined behaviour. What if `oldElem` appears multiple times? Only the first occurrence is replaced. What is unchanged in the vector? Only the matching element changes, but the comment does not promise that. A proper specification:

```
int replace(std::vector<int>& vect, int oldElem, int newElem);
// requires:  oldElem in vect
// modifies:  vect
// ensures:   let i be the first index such that vect@pre[i] == oldElem.
//            Then vect[i] == newElem, and all other elements of vect
//            are unchanged.
// returns:   i  (such that vect@pre[i] == oldElem and vect[i] == newElem)
```

Notice four things this specification does that the original comment did not:

- It **rules out the undefined-behaviour case** by requiring `oldElem` to be present. The implementation's missing return is now the client's problem to avoid, not the function's bug.
- It **commits to a particular occurrence**, the first, so clients can rely on which element gets replaced when there are duplicates.
- It **constrains everything that is not the target index**: "all other elements of `vect` are unchanged." Without this, an implementation that sorts the vector before returning would technically satisfy the rest of the spec.
- It uses `vect@pre` to talk about the input state and `vect` to talk about the output state, so that the ensures clause can simultaneously describe both.
-

💡 **Tip:** A useful exercise on any function you write: ask "what does this function *not* promise?" and then make every desirable thing it actually does into an explicit part of the postcondition. The bug is rarely in what the spec says; it is usually in what the spec forgot to say.

Example 3: isSubstring

A trickier case:

```
#include <string>
using std::string;

// check whether word is a substring of text
bool isSubstring(string text, string word) {
    if (text.length() == 0) return false;
    if (word.length() == 0) return true;
    int tl = text.length();
    int wl = word.length();
    for (int tIndex = 0; tIndex < tl; tIndex++) {
        int wIndex = 0;
        for (int ti = tIndex;
             ti < tl && wIndex < wl && text[ti] == word[wIndex];
             wIndex++, ti++) {
            if (wIndex == word.length() - 1) return true;
```

```
            }
        }
        return false;
    }
```

Suppose a colleague reads this and tries to write a specification. Their first attempt might be a literal description of the algorithm:

```
// This method scans text from beginning to end, building up a match
// for word, and resetting that match every time the running match fails.
// If text is empty, always returns false. If word is empty, always
// returns true. Returns true if word appears anywhere in text...
```

⚠ **Warning:** A *complicated description* of a function's behaviour is a strong signal that something is wrong, either the design is bad, or you have not yet understood the problem clearly. If your spec reads like a description of the algorithm rather than a description of *what the function delivers*, step back and re-think.

A much cleaner specification:

```
bool isSubstring(string text, string word);
// returns: true iff there exist (possibly empty) substrings A and B
//          such that text == A + word + B
```

This single line captures all of the corner cases the algorithmic description fumbled:

- If word is empty, then taking A == text and B == "" satisfies the equation. The function returns true. ✓
- If text is empty and word is empty, the same logic returns true. The original code returned false; its early-out for empty text disagrees with the clean specification.
- If text is empty and word is non-empty, no decomposition exists, and the function returns false. ✓
- All non-edge-case behaviour follows directly from the existential statement, with no special-case reasoning.

Notice that *the original implementation is wrong* against the clean specification, it disagrees on the case where both strings are empty. Once you have a clean spec, the bug surfaces; before you had one, the bug was invisible because the loose description allowed multiple interpretations. The implementation must now be rewritten to conform.

> ❓ **Question:** Should the specification have mentioned that case explicitly?
>
> Once the spec is precise, it does mention that case, by implication. The reader doing case analysis ("what happens when text is empty?") gets a unique answer from the existential formulation. The danger is when the spec is *just* "true iff word is a substring of text" without the existential formula: that phrasing is ambiguous on edge cases. The lesson is that *phrasing precisely*, even one extra clause, is often what separates a usable spec from a misleading one.

Specifying Exceptions

Interface specifications can, and should, supersede C++'s exception specifications. C++ lets you declare `noexcept` or, historically, exception-specification lists, but the language-level mechanism is shallow: it can say *this function might throw*, but not *this function throws* `DivByZero` *exactly when* `denominator` *is zero*. The interface specification fills that gap.

Three rules govern exception specifications:

1. **Declare the complete list of possible exceptions.** The throws clause should name every exception type the function might propagate. "And maybe other exceptions too" is not acceptable; clients cannot write robust code if they do not know what they have to handle.
2. **Describe the conditions under which each exception is thrown.** Knowing the type is not enough; the client needs to know *when* the exception happens, so they can either prevent it or recover from it.
3. **Exclude exception-throwing cases from the precondition.** If the function throws an exception when given a particular input, then by the contract, the function *defines its behaviour* on that input (the behaviour being "throws"). A precondition is supposed to *exclude* inputs whose behaviour is undefined. Mixing the two means the spec is internally inconsistent.

A canonical example:

```
double quotient(int numerator, int denominator);
// throws:   DivByZero, if denominator == 0
// returns:  numerator / denominator
```

There is no requires clause here. The function is *defined* on `denominator == 0`, it throws `DivByZero`. A client who calls `quotient(5, 0)` and forgets to handle the exception will get an unhandled exception at runtime, but that is a bug in the client, not a violation of the contract. The function's behaviour was specified.

Compare with the alternative form, where the function instead requires a non-zero denominator:

```
double quotient_strict(int numerator, int denominator);
// requires: denominator != 0
// returns:  numerator / denominator
```

This is a different contract. quotient_strict promises nothing when `denominator == 0`, it may return zero, return garbage, divide by zero and crash, format your hard drive, or anything else. The two functions could in principle have identical implementations (both might just compute `numerator / denominator` and crash on zero); the difference is in what they promise their clients, and therefore in what their clients are allowed to assume.

> 💡 **Tip:** If you find yourself listing exception conditions and *also* requiring those conditions to not occur, you have written contradictory clauses. Pick one. Throwing is more client-friendly when the condition is hard to check in advance; requiring is appropriate when the client can trivially avoid the problem and you do not want the implementation cost of detecting it.

The modifies Clause vs. const

Both the modifies clause and the C++ const qualifier identify which data is modified by an operation. They are not, however, the same mechanism, and a thoughtful designer uses both.

	modifies clause	const qualifier
Granularity	Names specific objects, including globals, parameters, and member fields.	Marks whole parameters or whole methods as not-modifying their target.
Coverage	Lists ALL data changed, including global variables and other side effects.	Only covers what is visible through the const-qualified entity.
Enforcement	Documentation only, not checked by the compiler.	Immutability is enforced by the compiler at compile time.
Strength	Can describe arbitrary mutations of arbitrary objects.	Can only forbid modifications through this particular access path.

The modifies clause is *more expressive*: it can talk about side effects that const cannot reach (changing a global, writing to a logger, mutating a member through a non-const pointer parameter). The const qualifier is *more reliable*: it is checked by the compiler, so a function declared const that tries to modify its target will fail to compile.

Used together, they complement each other:

```
class Logger {
  public:
    void log(const std::string& msg);
    // modifies: this->log_file_, std::cerr
    // ensures:  msg is appended to the log file and written to stderr

    std::size_t size() const;
    // returns: the number of messages logged so far
    // (no modifies: const ensures nothing about *this changes;
    //  the specification reinforces that no other state changes either)
};
```

The const on `size()` tells both the compiler and the human reader that `*this` cannot change. The *absence* of a modifies clause then assures the client that nothing else changes either, no globals, no statics, no I/O. That second guarantee is something const alone cannot provide.

Specifying Classes

So far, we have specified individual functions. Specifying a class is more involved because a class is a *collection* of operations sharing some state, and the specification has to describe that state in client-visible terms before any individual operation's spec can refer to it.

Specification Fields

A class's public state, what the client is allowed to see and reason about, is given by **specification fields**. These are not C++ data members. They are abstract, named pieces of state that appear in the specification and may or may not correspond directly to private fields.

Specification fields are listed at the top of the class with a comment like // Specification fields:, followed by one line per field giving its name and meaning. The class's operations then refer to these fields in their modifies, ensures, and returns clauses.

The Account Example

```
class Account {
    // Specification fields:
    //   ActNo   = unique id of Account
    //   balance = amount of money owed for phone services
    //   fee     = monthly fee
  public:
    explicit Account(const AccountNo& num);
    // ensures: initializes *this to an Account whose
    //          ActNo   == num,
    //          balance == 0,
    //          fee     == 30
```

```
  virtual void bill();
     // modifies: this->balance
     // ensures:  this->balance == this@pre->balance + fee

     virtual void print() const;
     // modifies: cout
     // ensures:  cout == cout@pre + (textual representation of *this)
};
```

Several things in this example deserve attention:

- The three specification fields, `ActNo`, `balance`, `fee`, are stated as the conceptual state of an `Account`. They may map one-to-one onto private data members of class `Account`, or they may not; that is an implementation choice the client never needs to know.
- The constructor's ensures clause sets initial values for the specification fields. "Initializes `*this` to a fresh `Account` with the given state."
- The `bill()` method has a modifies clause naming the specific field it touches. Other clients can rely on `ActNo` and `fee` being unchanged, because they are not listed.
- The `print()` method modifies `cout`, note that this is a global, and we list it explicitly. C++ const on the method only protects `*this`; the modifies clause has to mention any other side effects.
-

❓ **Question:** Why cout == cout@pre + textual representation instead of just "prints the account"?

Because the client may care about more than the fact that something was printed. They might be redirecting stdout to a buffer, or composing multiple print() calls. Stating the postcondition as *append* rather than *replace* tells the client exactly what they can expect about the state of stdout after the call. Vague specifications hide useful guarantees.

Specifying Derived Classes

Derived classes inherit not only interface signatures but also specifications. When class `CheapAccount` inherits from class `Account`, every operation that `CheapAccount` does not override carries forward `Account`'s specification unchanged, and every operation that `CheapAccount` *does* override must still satisfy `Account`'s contract (this is the **Liskov Substitution Principle**, which we will study in detail in a later chapter).

Two practical rules follow:

- **List all specification fields in the derived class, inherited and new, or just the new fields.** Either is acceptable; the all-fields form is clearer for the reader but more verbose.
- **For overridden methods, write the complete specification, not an extension.** It is tempting to write "same as `Account::bill()` but also resets minutes"; the discipline is to write out the

whole modifies and ensures in full. This makes it easier to verify that the derived spec remains compatible with the base spec.

Concretely:

```
class CheapAccount : public Account {
    // Specification fields (new, inherited fields ActNo, balance, fee
    //                       are implicit):
    //   minutes  = number of minutes called since the last bill()
    //   freemin  = number of free minutes per billing period
    //   rate     = charge per non-free minute
  public:
    explicit CheapAccount(const AccountNo& num);
    // ensures: initializes *this to a CheapAccount whose
    //          ActNo   == num,
    //          balance == 0,
    //          minutes == 0,
    //          fee     == 30,
    //          freemin == 200,
    //          rate    == 1

    virtual void bill() override;
    // modifies: this->balance, this->minutes
    // ensures:  this->balance == this@pre->balance + fee
    //                          + max(0, minutes@pre - freemin) * rate
    //           this->minutes == 0
};
```

The `CheapAccount::bill()` specification overrides the base `Account::bill()` specification. Notice three things:

- It **does the work the base spec promises**: balance still grows by fee, consistent with `Account::bill()`. A `CheapAccount` used through an `Account*` pointer still does "what billing does" in the sense the parent contract promised.
- It **does additional work**: balance also increases by usage charges, and minutes gets reset to zero.
- Its modifies clause is *larger* than the base's, it now lists minutes in addition to balance. This is the override expanding what it touches; the contract permits it as long as the additional changes do not violate any base-class promise.

⚠️ **Warning:** The most common bug when overriding a method's spec is to *contract* the postcondition, to make the derived method promise less than the base did. That breaks substitutability: client code holding an Account* expects the full base contract, and a CheapAccount that delivers less will surprise them. Always check: does the derived method still satisfy the base method's ensures?

> 💡 **Tip:** The modifies clause can *grow* in a derived override (the derived method may touch additional state), but the ensures clause should always *imply* the base's ensures. A common pattern is: "the base promised X; we still promise X, and also Y." If you cannot honestly say "we still promise X", the override is unsafe.

The Specificand Set

Move now from how to write specifications to how to reason about them. Some terminology:

Term	Meaning
Interface specification	Describes the behaviour of some software unit (function, class, module).
Implementation	A concrete program intended to realize that behaviour.
Conforms	An implementation conforms to a specification if it satisfies every clause of the specification.
Specificand set	The set of all implementations that conform to a given specification.

Two questions arise naturally and are worth asking precisely:

- **Does this implementation conform to that specification?** This is the conformance-checking question. The answer is yes iff every clause of the spec holds for that implementation.
- **Does this specification represent that implementation?** This is the reverse question: given code, can we find a specification it satisfies? The answer is trivially yes (the spec "this function does whatever the code does" is satisfied), so the interesting form is "does this specification represent that implementation *usefully*?" A useful spec is one that is short, abstract, and reusable, not a transcript of the code.

The **specificand set** is the lens through which we will compare specifications. A specification is *strong* if its specificand set is small (few implementations satisfy it, the spec is demanding) and *weak* if its specificand set is large (many implementations satisfy it, the spec is permissive). Two specifications are equivalent if and only if their specificand sets are equal.

Specifying a function, therefore, amounts to choosing the right specificand set, neither so small that we accidentally rule out reasonable implementations, nor so large that bad implementations slip through. The next section makes this concrete.

Comparing Specifications

This is the chapter's centrepiece worked example. We have three implementations of a function find, and five candidate specifications. The exercise is to work out which implementations satisfy which specifications.

Three Implementations

First, the three implementations:

```
// Implementation 1
int find(const std::vector<int>& vec, int val) {
    for (int i = 0; ; i++)
        if (vec[i] == val) return i;
}

// Implementation 2
int find(const std::vector<int>& vec, int val) {
    for (int i = 0; i < vec.size(); i++)
        if (vec[i] == val) return i;
    return -1;
}

// Implementation 3
int find(const std::vector<int>& vec, int val) {
    for (int i = vec.size() - 1; i >= 0; i--)
        if (vec[i] == val) return i;
    return vec.size();
}
```

Five Specifications for find

And the candidate specifications:

```
Spec 1:
    // requires: exists i such that vect[i] == val
    // returns:  some j such that vect[j] == val
    //   (allows for a faster implementation, no falling-off guarantee)


Spec 2:
    // returns:  (exists j such that vect[j] == val) ? j : -1


Spec 3:
    // returns:  (exists j such that vect[j] == val) ? j : vect.size()
```

```
Spec 4:
    // returns:  (exists j such that vect[j] == val) ? j
    //               : k where k < 0 or k >= vect.size()


Spec 5:
    // returns:  (exists i such that vect[i] == val)
    //              ? (smallest j such that vect[j] == val)
    //              : -1
```

Which Satisfies Which

The conformance question is a careful case analysis.

Spec / Impl	Impl 1	Impl 2	Impl 3
Spec 1: precond val in vec; return any matching index	✓	✗ (returns -1, not a matching index)	✗ (returns vec.size(), not a matching index)
Spec 2: return matching index or -1	✗ (UB if val not in vec)	✓	✗ (returns vec.size(), not -1)
Spec 3: return matching index or vec.size()	✗ (UB if val not in vec)	✗ (returns -1, not vec.size())	✓
Spec 4: return matching index or any out-of-range k	✗ (UB)	✓ (-1 is out of range)	✓ (vec.size() is out of range)
Spec 5: return SMALLEST matching index or -1	✗ (UB)	✓ (forward scan returns smallest)	✗ (backward scan returns largest)

Reading this matrix illuminates several things:

- Implementation 1 satisfies only Spec 1, because Spec 1 is the only spec that requires val in vec, and that precondition is exactly what protects Implementation 1 from its undefined behaviour.
- Implementation 2 satisfies Spec 2 (by direct match), Spec 4 (because -1 is one of the allowed out-of-range sentinels), and Spec 5 (because the forward scan does return the smallest match).
- Implementation 3 satisfies Spec 3 and Spec 4 for analogous reasons.
- Spec 4 is *weaker* than Spec 2 (every impl that satisfies Spec 2 also satisfies Spec 4) and weaker than Spec 3. It accepts the largest set of implementations, because it leaves the sentinel value unconstrained. Geometrically, Spec 4's specificand set *contains* both Spec 2's and Spec 3's specificand sets.
- Spec 5 is *stricter* than Spec 2: it commits to returning the smallest matching index, where Spec 2 was free to return any. Every implementation satisfying Spec 5 also satisfies Spec 2; the converse is not generally true.

❓ **Question:** Which spec is "best"?

"Best" depends on what clients need. Spec 4 maximizes implementer freedom but offers the client almost no guarantee about the sentinel value, forcing clients to write code that handles *any* out-of-range k. Spec 5 maximizes client predictability, the client can rely on "smallest matching index", but rules out reasonable implementations like Impl 3 that scan backward (which might be faster in some access patterns). The trade-off is what we will formalize in the next section.

How Precise Should a Specification Be?

Two principles, dual to each other, govern the right size of a specification's specificand set:

- **Sufficiently restrictive.** A specification is sufficiently restrictive as long as it rules out all *unacceptable* implementations to the module's clients. If clients are unhappy with an implementation that satisfies the spec, the spec is too loose; tighten it.
- **Sufficiently general.** A specification is sufficiently general as long as it does not rule out *desirable* implementations. If a reasonable, efficient implementation does not satisfy the spec, the spec is too tight; loosen it.

💡 **Tip:** When in doubt about whether to tighten or loosen, think about *the next version of the implementation*. If you commit a too-tight spec now, you will not be able to optimize the implementation later without breaking clients (or quietly violating the contract). If you commit a too-loose spec now, clients will write code that relies on observed behaviour rather than promised behaviour, and that code will silently break when the implementation changes.

Checking Preconditions

A specification with a requires clause places an **obligation** on the client: ensure the precondition holds before calling. If the client breaches the obligation, the contract is silent as to what the function may do.

💡 **Tip:** *If the preconditions are not met, our code may do anything and still comply with the contract: return without executing, throw an exception, terminate, or process the call as best it can.*

This is the contract talking. It is *not* advice. The contract permits anything, and the implementer is free to take advantage of that freedom for speed. But there is a separate, pragmatic question: what should a well-engineered implementation actually do?

When to Check and When Not To

The pragmatic guideline is twofold: (1) check the precondition if it is easy to check, and throw an exception if it is violated; (2) if checking is impractical, design the procedure to detect indications of the problem during execution and report them. A program that detects and reports errors is more reliable and more robust than one that does not.

Two examples from the slides make the trade-off concrete:

```
// requires: list is not empty     ← cheap to check (a single comparison)
// requires: list is sorted        ← expensive to check (linear scan)
```

Checking the first is essentially free: a single boundary comparison. Add the check; throw an exception on violation. The contract still says "undefined if the list is empty," but the implementation goes the extra mile and reports the bug instead of producing nonsense.

Checking the second is *expensive*. Verifying that a list is sorted is an O(n) scan. If the rest of the function is an O(log n) binary search, the check would turn an O(log n) operation into an O(n) one, a categorical performance regression. Worse, the check is essentially redoing the work the client must have already done to satisfy the precondition in the first place. Do not check it.

> ⚠️ **Warning:** Preconditions are most often used precisely *because* the property they require is not easy to check. A function that wanted to verify everything it assumed could simply not have preconditions at all. The whole point of requires is to shift expensive verification off the function's critical path, *onto* the client, where it can be amortized or hoisted out of inner loops. Defeat the design by checking anyway, and you have given up the speed advantage you were buying.

Detecting and Reporting Errors

When the precondition is too expensive to check, but the function still wants to be helpful when something is amiss, the second-best strategy is **opportunistic detection during execution**. The function does not check the precondition up front; it does its work, and if it stumbles across evidence that the precondition was violated, it stops and reports an error.

In the binary-search-on-sorted-list example, this might mean noticing during the search that an element is out of order relative to its neighbours, and throwing an UnsortedList exception at that point. The function still does its O(log n) work in the common (correct) case; only the pathological cases pay the detection cost.

To summarize the approach:

- Check for indicative properties during the procedure's execution.
- Return to the calling state and raise an exception when a problem is detected.

- Even when a precondition exists, the function can additionally throw an exception in case the precondition is violated, and detection happens to be cheap. A function may legitimately have both a precondition and a throws clause for the same situation.

The phrase "return to the calling state" is important. When the function detects an error and raises an exception, it must do so without modifying anything the client would not expect to be modified. This is a **strong exception-safety guarantee, which we will revisit** when we cover exception safety in detail.

Summary

Recognition

- A specification is a **contract** between a module's provider and its user, covering obligations, benefits, and consistency constraints.
- A specification serves two roles: documenting the design of a future module, and documenting the correct usage of an existing one.
- The **specificand set** of a specification is the set of implementations that satisfy it.

Comprehension

- Pros and cons of restrictive vs. general specifications: a tighter spec helps the client but constrains the implementer; a looser spec helps the implementer but burdens the client with more edge cases to handle.
- Why expressions in specifications must be written over *public* state, clients cannot reason about private state, and the implementation must remain free to change it.
- The relationship between the modifies clause and C++ const: one documents (including globals), the other enforces (only for the qualified entity).
- Why a precondition is *not* the right place to record conditions handled by exceptions.

Application

- Writing a specification for a C++ method using requires, modifies, throws, ensures, and returns clauses.
- Writing a specification for a class using **specification fields** and per-method clauses.
- Writing a specification for a derived class that overrides methods, with the full spec (not an extension).
- Determining whether a C++ implementation conforms to a specification by checking each clause against the implementation.
- Determining whether one specification is *stronger* than another by comparing their specificand sets.
- Choosing whether and how to check a precondition at runtime, based on cost and on the value of robustness in the use case.

Practice Problems

1.

Write a CS 247-style specification for each of the following function signatures. Be explicit about preconditions, modifies clauses, and edge cases (negative numbers, empty containers, repeated elements).

```
// (a)
double average(const std::vector<double>& v);

// (b)
void reverse_in_place(std::vector<int>& v);

// (c)
int max_index(const std::vector<int>& v);

// (d)
void swap(int& a, int& b);

// (e)
bool starts_with(const std::string& s, const std::string& prefix);
```

For each, also decide whether the function *should* have a precondition or instead handle the edge case via an exception, and justify your choice in one sentence.

2.

Below are two specifications for the same operation. Determine which is stronger (smaller specificand set), and give a concrete implementation that satisfies one but not the other.

```
// Spec A
int dedup(std::vector<int>& v);
// modifies: v
// ensures:  v contains no duplicate values
// returns:  the new size of v

// Spec B
int dedup(std::vector<int>& v);
// modifies: v
// ensures:  v contains exactly the unique values from v@pre,
//           in the order of their first appearance in v@pre
// returns:  the new size of v
```

3.

Given the specification below, decide for each implementation whether it conforms. If it does not, identify the failing clause.

```
int second_min(const std::vector<int>& v);
// requires: v has at least two distinct values
// returns:  the second-smallest value in v
//           (i.e., if the smallest value is m, returns the smallest
//            value in v that is strictly greater than m)
```

Three candidate implementations:

```
// Implementation A
int second_min(const std::vector<int>& v) {
    auto sorted = v;
    std::sort(sorted.begin(), sorted.end());
    return sorted[1];
}

// Implementation B
int second_min(const std::vector<int>& v) {
    int m1 = INT_MAX, m2 = INT_MAX;
    for (int x : v) {
        if (x < m1) { m2 = m1; m1 = x; }
        else if (x < m2) m2 = x;
    }
    return m2;
}

// Implementation C
int second_min(const std::vector<int>& v) {
    std::set<int> s(v.begin(), v.end());
    auto it = s.begin();
    ++it;
    return *it;
}
```

4.

Find every bug in the following specification. There are at least three. Rewrite the specification to be correct.

```
void normalize(std::vector<double>& v);
// requires: v is not empty, and not all elements of v are zero,
//           and the sum of v is not zero
// ensures:  every element of v is divided by the sum of v
// throws:   EmptyVector if v is empty,
//           DivByZero  if the sum of v is zero
```

Hint: think about (a) what "normalize" should mean for negative inputs, (b) the relationship between the requires clause and the throws clause, and (c) whether the ensures clause actually says what the implementer intended.

5.

Write the specification of a class Stack<T> that supports the standard operations push, pop, top, size, and empty. Your specification should include:

- A list of specification fields describing the abstract state of the stack.
- A constructor that creates an empty stack.
- Full pre- and postconditions for every operation, including what happens when pop or top is called on an empty stack (your choice: precondition or exception, with justification).
- Explicit modifies clauses where appropriate.

Then write a second specification, this time of a derived class BoundedStack<T> that has a maximum capacity. push should throw StackFull if the stack is at capacity. List all inherited and new specification fields, and write the complete spec for each method that the bound changes (i.e., the constructor and push).

6.

Below is a function that *works correctly,* but whose specification is too loose. Write a stronger specification that rules out implementations the client would find surprising, while still allowing the given implementation.

```
// Spec (as written):
// modifies: v
// ensures:  v contains the same elements as v@pre
void shuffle(std::vector<int>& v);

// Implementation:
void shuffle(std::vector<int>& v) {
    static std::mt19937 rng(std::random_device{}());
    for (int i = v.size() - 1; i > 0; --i) {
        std::uniform_int_distribution<int> dist(0, i);
        std::swap(v[i], v[dist(rng)]);
    }
}
```

Why is the original specification problematic? (Hint: consider what implementations it permits that a client of shuffle would consider unacceptable, for example, the identity function technically satisfies it.) What is the right level of precision to demand of *randomness* in the spec?

7.

⭐ **Stretch, specification archaeology.** You are inheriting a codebase. The following function has no specification, only an opaque name and an implementation. Reverse-engineer (a) the strongest specification the implementation satisfies, and (b) the weakest specification that captures what a reasonable client would want.

```
std::pair<int, int> partition(std::vector<int>& v, int pivot);
// (no specification was written)

std::pair<int, int> partition(std::vector<int>& v, int pivot) {
    int low = 0;
    int high = v.size();
    int i = 0;
    while (i < high) {
        if (v[i] < pivot)       std::swap(v[i++], v[low++]);
        else if (v[i] > pivot)  std::swap(v[i],   v[--high]);
        else                    ++i;
    }
    return {low, high};
}
```

(a) What invariant does the loop maintain? Express it in terms of *ranges within v* and the pivot value.

(b) Write the **strongest** specification this implementation satisfies. ("Strongest" = smallest specificand set; reading the postcondition should let me reconstruct exactly what the implementation does.) Use v@pre wherever needed.

(c) Write a **weaker** specification that still gives the client what they need: a partitioning of v into three regions (less than pivot / equal to pivot / greater than pivot), with the regions described by the returned indices. Your weaker spec should not commit to the order of equal elements within the middle region, nor to relative order within the outer regions, leaving the implementer free to switch to a different partitioning algorithm.

(d) Which of (b) or (c) would you actually publish? Defend your answer in terms of client predictability versus implementer flexibility.

8.

⭐ **Stretch, checking precondition cost.** Each of the following functions has a precondition. For each, decide whether to (i) leave the precondition unchecked, (ii) check it cheaply with an if + exception throw, (iii) detect violations opportunistically during execution, or (iv) redesign to remove the precondition entirely. Justify each choice in one or two sentences, citing the asymptotic cost of checking and the failure mode if it is not checked.

```
// (a)
int kth_smallest(const std::vector<int>& v, int k);
// requires: 0 <= k && k < v.size()

// (b)
double cdf_lookup(const std::vector<double>& sorted_values, double q);
// requires: sorted_values is non-empty and sorted in non-decreasing order
// requires: 0.0 <= q && q <= 1.0

// (c)
void apply_in_topo_order(Graph& g, std::function<void(Node*)> visit);
// requires: g is a directed acyclic graph

// (d)
void set_pixel(Image& img, int x, int y, Color c);
// requires: 0 <= x && x < img.width()
// requires: 0 <= y && y < img.height()

// (e)
Matrix inverse(const Matrix& m);
// requires: m is invertible (i.e., det(m) != 0)
```

Finally, redesign one of the five (your choice) to *eliminate* its precondition entirely, for instance by making the function return an std::optional or by widening the input type. Show the new signature and specification, and argue whether the redesign is or is not an improvement.