Software design principles

Foundational concepts

Information hiding

It refers to hiding the parts of the code that are likely to change behind stable interfaces for modules. Other modules should depend only on each other's stable, public-facing interfaces. Hidden implementation details can then change without affecting other modules.

Abstraction

It refers to the process of modeling and exposing only the important aspects of objects, while leaving details out of the picture. Abstraction is also the process of identifying which abstractions can be composed to achieve the desired functionality in entities.

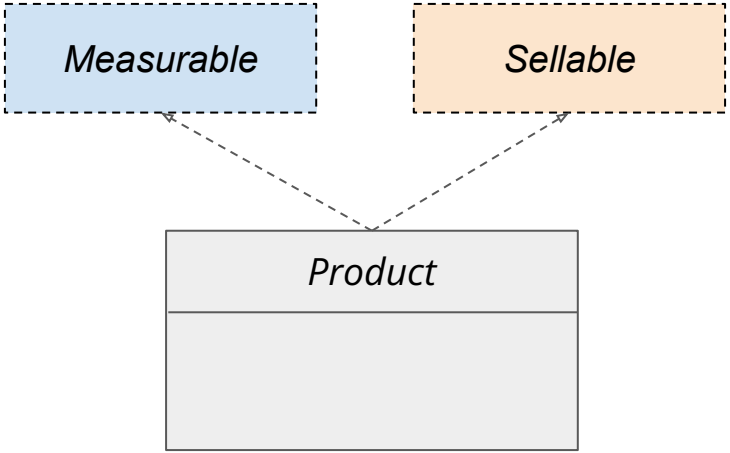
Encapsulation

Closely related to information hiding, it refers to the bundling of data with the methods to allow operations on that data, thus preventing direct access to some of an object's components.

Polymorphism

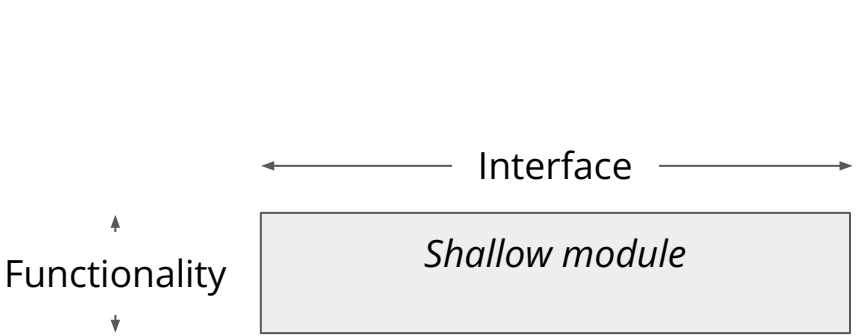
Closely related to abstraction, polymorphism refers to a pattern where classes have **different functionality while still complying with and sharing a common, stable interface**.

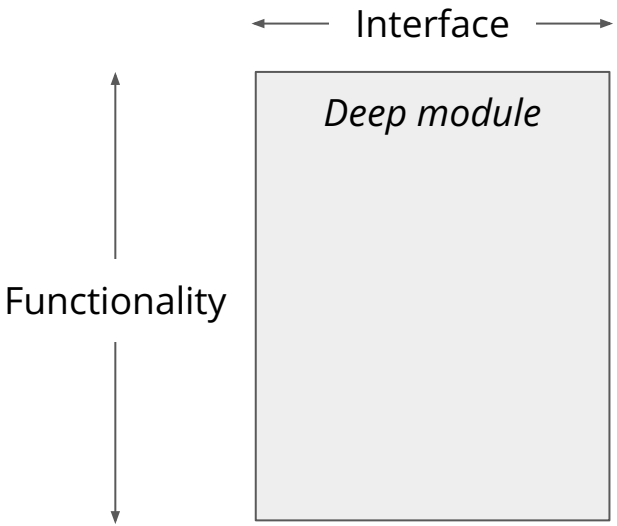
Foundational concepts



Foundational concepts: Deep modules

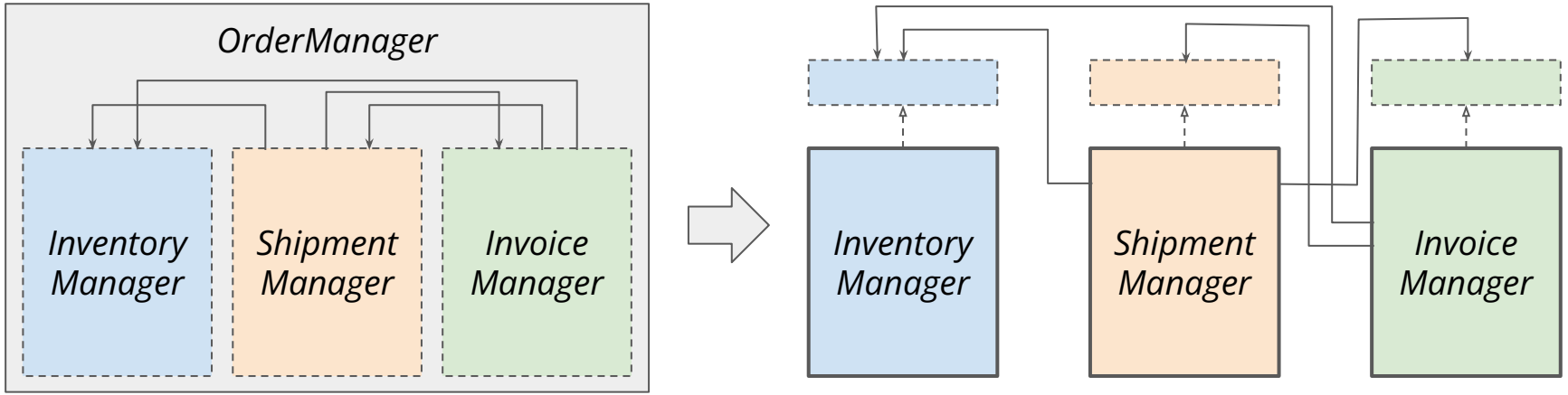
Deep modules are those that have a large amount of functionality exposed via a relatively simple interface. They are easier to work with, easier to maintain, and add more functionality than complexity to the code.





SOLID: Single responsibility principle

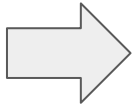
An entity should have only one reason to change. In other words, it should **encode only one single part of the domain knowledge**. This leads to code that is easier to maintain, easier to extend, and easier to understand. Violating this principle can quickly lead to high coupling between multiple responsibilities, and code that is hard to maintain.

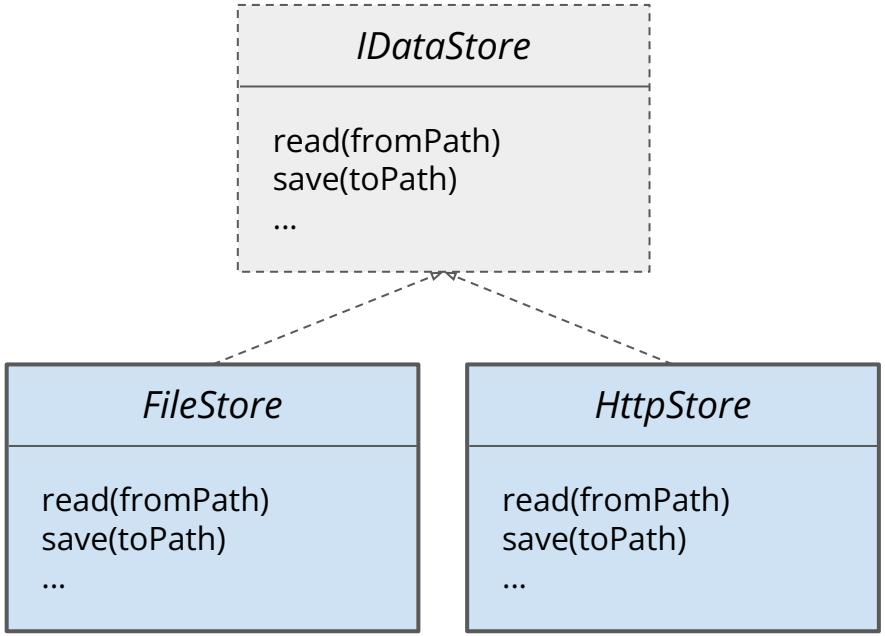


When grouping everything under the same class, it's very easy to introduce direct dependencies between the different responsibilities. This can lead to cases where a change in one responsibility requires updating other, unrelated parts of the code.

SOLID: Open/closed principle

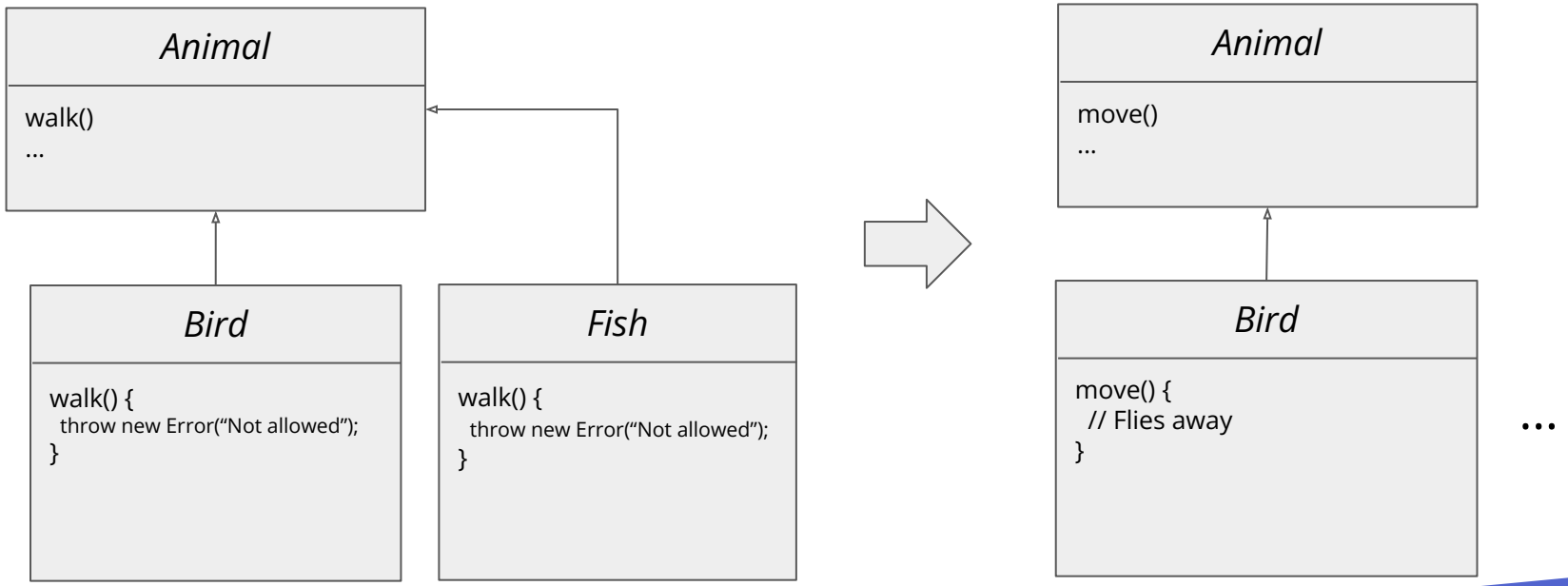
Entities should be open for extension, but closed for modification. In other words, we should be able to add functionality by writing new entities and code rather than having to modify existing code.





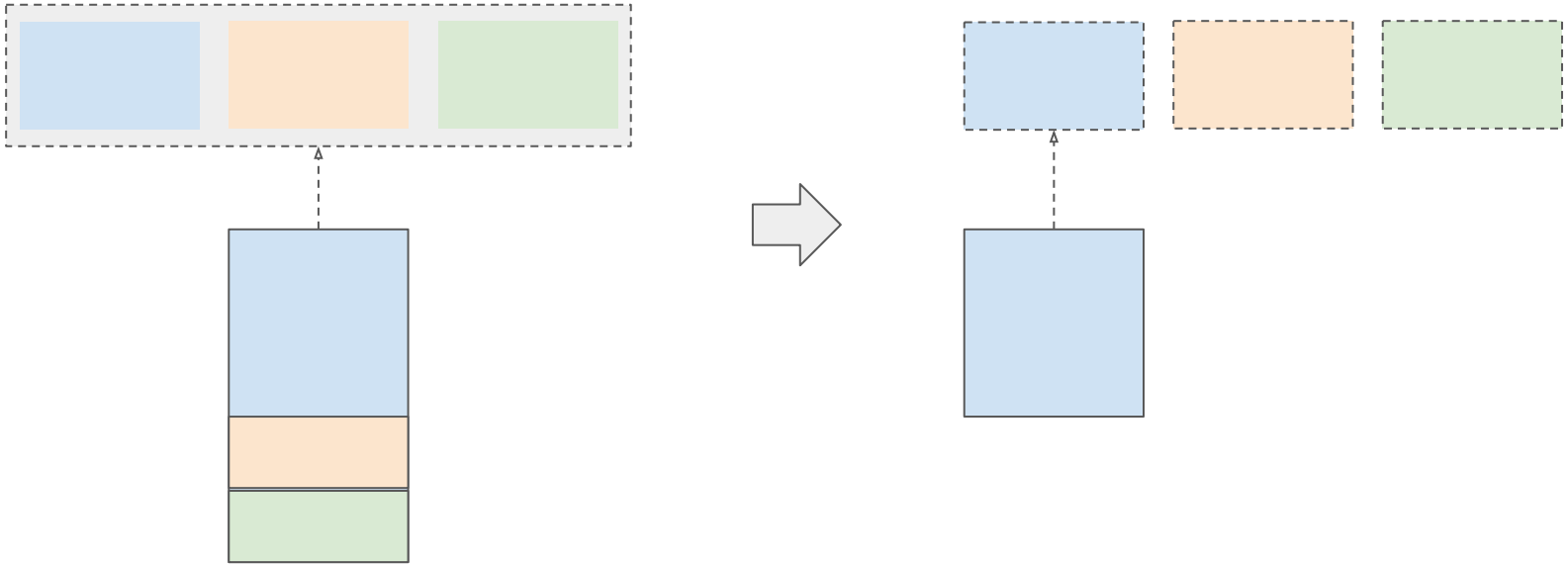
SOLID: Liskov substitution principle

If an object of type *T* is a subtype of another object of type *S*, then any object of type *S* should be substitutable by objects of type *T* without breaking the application. In simpler terms, subclasses do not implement any behavior that contradicts or restricts the intended behavior of their superclasses.



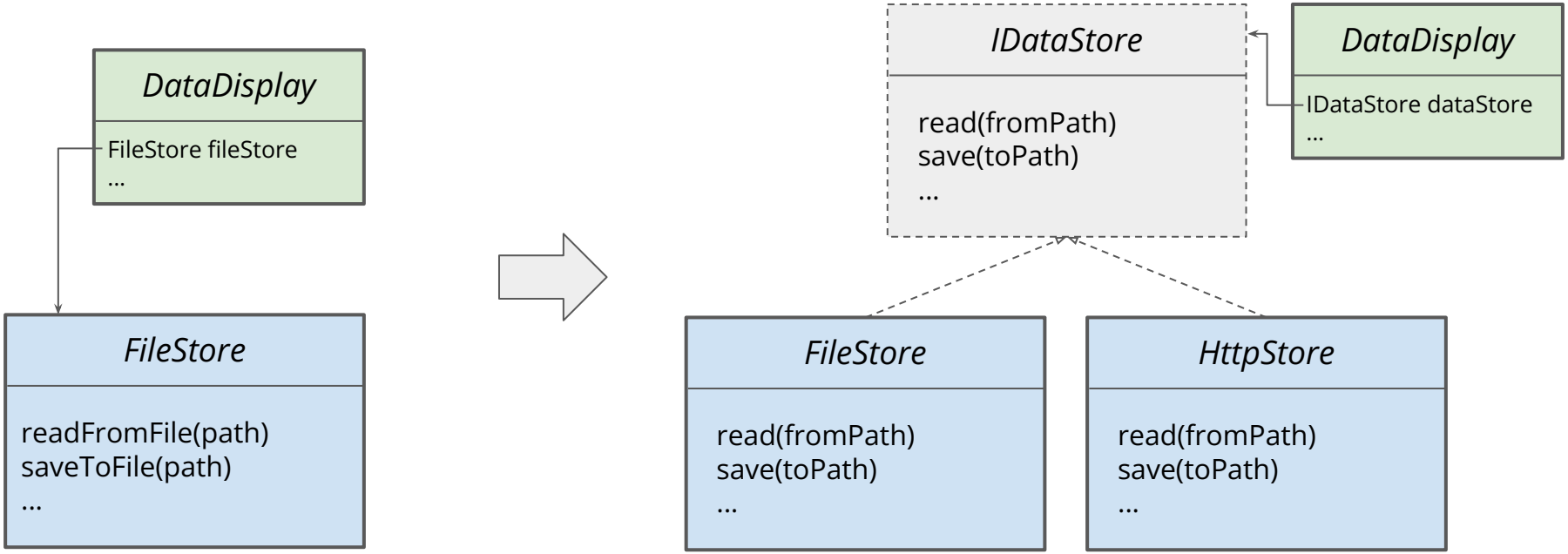
SOLID: Interface segregation principle

No code should depend on methods that it does not use. For example, if a certain class C implements a certain interface I, then the class should not have to provide empty implementations for methods it doesn't need. It is recommended to split large interfaces into smaller ones, as to allow classes to implement only those methods that they really need.



SOLID: Dependency inversion principle

High-level modules should not depend on low-level modules; instead, both should depend on abstractions. Additionally, abstractions should not depend on details; instead, details should depend on abstractions.



Code smells

Loops

Adding too many loops to our code can quickly make it **confusing to read and hard to maintain**. Iterating over entities is a fundamental concept in programming, but **hard-coding loops** can very quickly break several fundamental rules of **information hiding** and **encapsulation**.

Why is it bad?

- Loops obfuscate logic by requiring us to **explicitly code the iteration logic** in the main body of the code block.
- Loops often require us to **hard-code implementation details**, hurting **encapsulation**.
- Loops make our code considerably **more verbose** than it has to be.

How to fix it?

- Use a pipeline structure rather than hard-coding loops. Pipeline structures (`.map`, `.filter`, `.reduce`, among others), abstract the iteration logic and enable us to focus on how the elements of a collection are being filtered, mapped, or summarized.

Uninformative comments

Good comments are not a code smell. We want good comments. Bad comments are a code smell. We don't want bad comments. **Bad comments do not add any information that cannot be derived from the code itself.** Good comments capture the reasoning behind a complex / unclear piece of code. They also should capture the intended usage of the code.

Why is it bad?

- Uninformative comments make the code more polluted and harder to read.
- Comments generally add overhead to code maintenance. Good comments deliver value in return for the overhead, bad comments don't deliver any additional value.
- Uninformative comments are often used to try to clarify unclear code that is the result of other code smells.

How to fix it?

- First make sure that other smells are already addressed.
- Once the code is clean, remove any comment that duplicates information from the code.
- Add comments to capture any thought process that is not clear from the code, as well as any intended usage of the module.

Shotgun surgery

Happens when developers **have to change code in multiple places when making a change in somewhere else.** This is a result of high coupling, often to internal data structures that are directly exposed without a layer of abstraction. This coupling can happen at different levels, with semantic coupling being especially hard to treat.

Why is it bad?

- It leads to lower code maintainability, since we need to change multiple places when a single change is made in a component.
- We can easily forget to change one of the parts of the code, thus unintentionally introducing bugs.
- It is often a result of poor encapsulation or poor code modularity.

How to fix it?

- Add layers of abstraction between the components to reduce coupling to internal data structures.
- Prevent information leakage by encapsulating knowledge regarding the meaning of an internal variable in its respective class and exposing it via methods.

Knowledge duplication

Happens when **the same codified knowledge is found in multiple places**. The focus is on the knowledge, rather than only on code. It will be inevitable to have similar code, and it's not always the best strategy to unify them. If they represent different pieces of knowledge, it's not bad to keep them separate. If, however, they represent the same unit of knowledge (for example, code to read a file or to parse HTTP responses is copied in multiple places), then duplication is harmful.

Why is it bad?

- When a certain piece of duplicated knowledge changes, we need to look throughout the code to update every duplication.
- Different instances of the duplicated knowledge in the codebase may be doing different things while being expected to do the same thing (over time, these duplications tend to diverge in functionality and be "overfit" to the places where they are being used).

How to fix it?

- Extract the duplicated knowledge into helper functions or classes.
- Use the helper functions or classes throughout the code, but make sure not to overgrow them. They should still follow the SOLID principles.
- Avoid extracting similar code that represents different knowledge. This increases coupling between unrelated parts of the domain.

Alternative classes with different interfaces

When we have classes that should be substitutable by each other but have different interfaces. This often happens when different classes have methods which perform the same functionality but vary slightly on their signature, and usually indicates that the hierarchy and abstraction between classes is not properly defined.

Why is it bad?

- This violates the principle of Abstraction, since different classes with a common abstraction but different interfaces cannot be substituted by each other.
- This can also easily lead to more code than necessary, and once again, duplication of knowledge due to having to handle slight variations in the interfaces.

How to fix it?

- Standardize abstractions and their respective interfaces.
- If the different classes are proper subtypes of a supertype and the functionality is needed by all the subclasses, consider placing it in a superclass (but be careful! Poorly used inheritance can quickly lead to many other code smells).
- Favor composition over inheritance for sharing behavior between classes.

Unclear / confusing names

When the name of the entity does not clearly reflect what it does. If we cannot understand what the variable represents without looking at what's assigned to it, we are not naming it correctly. Poor naming also happens when the name does not precisely reflect the information being stored by variables (for example, what is a "customer" variable of type string storing? Is it the customer name? Or the customer ID?)

Why is it bad?

- Code becomes **unclear** and **confusing to understand**.
- The further an entity's usage is from its declaration, the harder it is to understand its **meaning** if the name is unclear.
- Makes it **harder for others** (including our future selves) to work with the code, since we must always look for the declaration and context of an entity with unclear name.

How to fix it?

- Change the name of the entity to be meaningful and represent what it does.
- If the name becomes too long, maybe it's a sign that the entity is doing or storing too much information. It's time to split the code into smaller portions.
- Write informative comments about behavior that cannot be captured in the name. Most IDEs provide an embedded way of showing comments.
- Avoid using names that mirror the variable's type.

Primitive obsession

When developers avoid creating their own fundamental types to deal with domain behavior or complex logic (for example, currency, telephone numbers, addresses), and rely too much on primitives. This does not leverage any of the principles of information hiding and encapsulation, and can result in very poor code maintainability.

Why is it bad?

- Leads to poor encapsulation, since knowledge about how to handle such domain behavior gets fragmented throughout the code.
- Can very easily lead to duplication of knowledge when having to deal with validation and other operations.
- It's easy to forget to update duplicated parts of the knowledge when the intended behavior changes.

How to fix it?

- Model domain behavior and non-primitive knowledge through custom types.
- Beware of overfitting types to their use-cases. This can lead to "type explosion", which also leads to hard-to-maintain code. "Type explosion" happens when we create too many fundamental types which differ from each other only slightly, indicating we are missing the correct underlying abstractions.
- Encapsulate validation logic in the custom types' classes.

Multitaskers / long functions

Long functions are not bad per se. **Functions and classes that do more than one thing (and are, as a consequence, long) are bad.** This smell is about **jamming different responsibilities into the same entity**; therefore the “multitasker” name rather than just “long function”. A long function that encapsulates, for example, the complexity of interacting with the kernel and exposes a simple interface is a good, very desirable function.

Why is it bad?

- Multitaskers are harder to understand and to change, since they require knowledge from different contexts and functionalities.
- Pieces of knowledge from within multitaskers cannot be reused if it’s not extracted.
- Longer functions are naturally more complex to work with.

How to fix it?

- Identify what is the core purpose of a function / class. Everything that doesn’t directly belong to that core purpose should be extracted into other entities.
- Use good naming strategies for functions. A function with a good name often transmits the intent of the code without the developer having to go and check the function body.

Divergent change

Happens when the same entity (a function, a module, a class) needs to change due to different reasons. For example, if a function to sort products by price needs to change when a new product type is added, when discounts are introduced, and when a new way of handling out-of-stock products is implemented.

Why is it bad?

- When an object or class needs to change for more than one reason, it indicates that we are violating the Single Responsibility principle.
- It is often a sign of poor modularity, since different pieces of knowledge are grouped together.

How to fix it?

- Split the object or class into multiple entities, each one with its own responsibility.
- Call each of the resulting entities whenever needed.
- Avoid de-duplicating code that doesn't represent the same piece of knowledge. This increases coupling without any additional benefit, and makes it harder to change code afterwards.

Large interfaces

Instead of only large classes being seen as a code smell, I prefer to say large interfaces. **If classes have many private modules to support only a few exposed public methods, these classes follow good practices of information hiding and encapsulation.** If, however, there are too many public methods grouped in a single interface, it's a sign that the interface is trying to do too much and could be better off broken down into multiple modules.

Why is it bad?

- Large interfaces are a sign that the modules or classes are leaking too much information about their internal structures to the external world. This violates the principle of Information hiding.
- Introduces coupling between different parts of the code that depend on this "central" interface.
- Large interfaces tend to grow larger and larger if untreated, leading to "big ball of mud" modules extremely hard to maintain.

How to fix it?

- Avoid exposing low-level details of a functionality to the external world.
- Split interfaces which have more than one responsibility.
- When defining the boundaries of an interface, focus on the behavior and the domain knowledge rather than on purely the number of public methods exposed.

Long parameter list

Happens when method signatures require long lists of parameters. They may include, among other things, flag parameters, which are a specially bad pattern. Flag parameters are used solely to determine which conditional path the method must execute, and make it much harder to reason about the method's overall behavior and intention.

Why is it bad?

- Long parameter lists are harder to maintain. For example, making intermediary parameters optional is a specially difficult task to achieve in most languages.
- If we pass parameters in the wrong order, we may very likely break the function. It becomes increasingly difficult to reason about a function's invocation the longer its parameter list gets.

How to fix it?

- Substitute data parameters by a single data object parameter.
- If it's possible to obtain information about another parameter from a parameter, remove the additional parameter.
- Avoid passing flag parameters.

Mutable data

Mutable data happens happens when components modify entities in-place instead of returning new instances of these entities with the modified properties. A particularly harmful type of mutable data is global data, since it's considerably harder to properly track changes to global objects.

Why is it bad?

- Updating data used by other parts of the code can easily introduce bugs and side-effects.
- Bugs due to mutable data are particularly hard to spot and fix.
- Mutable data creates hidden dependencies and coupling between multiple parts of the code.

How to fix it?

- Encapsulate data used by multiple methods with proper setter and getter methods, so that its usage can be better tracked throughout the code.
- When modifying an object, opt for returning a copy of the object which incorporates the changes in the data and leave the original object intact.
- Avoid modifying function parameters **as much as possible**.

Middle man

The middle man is a class that has several methods which act as passthrough methods (they simply delegate the call to a method of one of the class' dependencies), without adding any meaningful logic on top of it.

Why is it bad?

- This pattern adds virtually no value to the code, but it introduces additional sources of coupling since we are calling methods between multiple interfaces, and often fooling ourselves that we are following the information hiding principle (if the methods of a class are just calling other methods of another class, is this really information hiding?).
- Generally, encapsulation and information hiding are highly encouraged, but when methods become too shallow it's a sign that we are not properly structuring our code.

How to fix it?

- If there is no additional logic, remove as many intermediaries as possible. Keep in mind the separation between layers (if, however, layers are existing just too pass methods to lower-level layers, the system architecture probably needs some rework).
- Combine methods and simplify the class' interface to provide more functionality through fewer methods.

Data clumps

This happens when several pieces of data are always used together, but they are not encapsulated in an object or class. It can happen in method bodies, as well as parameters and return values.

Why is it bad?

- Leads to duplication of knowledge, since we always have to make sure that all the fields are present when a field is being used.
- Leads to more complex and less maintainable code, since data clumps tend to lead to long lists of parameters for methods that need the data.
- Data clumps normally reflect the internal data structure of objects, thus violating the encapsulation principle.

How to fix it?

- Move the closely related fields to an object or class.
- Refactor the method signatures to receive the whole object via the parameter.
- Encapsulate the behavior for managing and accessing the data within the object.

Repeated switches

Switch statements are not inherently bad (sometimes we need them), but this code smell happens when we have the same switch statements happening in different parts of the code. Repeated switches often happen when behavior must be customized in different places but based on the same set of conditions.

Why is it bad?

- This is again about duplicating knowledge in the code base. When we have the same conditional logic that needs to be executed in multiple places and we simply copy the code blocks, we are introducing multiple touchpoints that need to be changed in case a change in the conditional logic is necessary.
- It is very easy to miss adding / updating individual conditional clauses when logic changes, so repeated switches or chains of if-else statements is a common place for bugs to breed.

How to fix it?

- Encapsulate the conditional logic.
- Use polymorphism whenever relevant and applicable to remove conditional statements.

Refused bequest

When subclasses are inheriting methods and fields from the parent that they don't need. This happens when too much functionality is placed in the superclass, or when inheritance in used where composition would provide a better solution. This becomes a much worse smell when the subclass' interface does not align with the inherited methods from the superclass.

Why is it bad?

- This points towards an incorrect inheritance hierarchy. We are putting too much information at the superclasses, thus leading to unnecessary information leaking through to the subclasses.
- This is usually motivated by trying to avoid code duplication the incorrect way: by adding an inheritance relationship between the entities.

How to fix it?

- Remove unnecessary methods from the superclass.
- Use composition to inject behavior into classes instead of relying on inheritance too much.
- If there is interface misalignment, revisit the super-subclass relationship and consider extracting methods from the superclass.

Speculative generality

This code smell happens when we add functionality that isn't really needed but we believe will be needed in the future. It often takes place in the form of unused data (parameters, fields in objects, columns in the database schema, among others), unused methods, and unnecessary classes.

Why is it bad?

- This leads to code that is more complex than it has to be, and is harder to maintain over the long run.
- Unnecessary functionality is wasted time and resources.

How to fix it?

- Remove unused parameters.
- Remove unused method signatures when overloading.
- Remove unnecessary classes and functions.
- Keep the structure of the code simple: avoid over-engineering the solution and always keep the use-case in mind.

Unnecessary exceptions

Throwing exceptions is easy, handling exceptions properly is hard. Unnecessary exceptions happen when the errors could have been handled in a cleaner way by using sensible behavior defaults instead of throwing exceptions, and they make the code harder to maintain and to reason about.

Why is it bad?

- Exceptions disrupt the normal flow of the program, and are harder to reason about.
- Boilerplate code for handling exceptions can be painstakingly long and complicated.
- Testing exceptions can be particularly hard and often an overseen part of development.
- Exceptions are part of a module's interface: too many exceptions lead to shallower modules because they introduce more complexity to its interface without adding any functionality.

How to fix it?

- Add default behaviors or values for when entities are not found or values fall outside of expected ranges.
- Mask exceptions by handling them as closely as possible to the place where they are thrown.
- Aggregate exception handling by handling as many exceptions as possible in a single piece of code, rather than using a distinct piece of code to handle each individual exception.

Feature envy

Happens when a certain function within a module needs to call functions from other modules too often to complete its functionality. This often happens when functions of related functionality are spread across multiple modules, indicating that the code has low cohesion.

Why is it bad?

- Low cohesion leads to high coupling between modules, which makes the code considerably harder to maintain over the long run. Low cohesion also indicates that the responsibilities of the modules are not well defined and respected.
- It may be easy to forget to call a certain method or to call methods in the incorrect order, leading to many hard-to-find and fix bugs.

How to fix it?

- Move the function that is continuously interacting with another module to that module. More generally, consider improving the design of the system by better setting the boundaries between the domains.
- “Put things together that change together.”

Shallow modules

Shallow modules happens when the number of public methods exposed via that module's interface is large when compared to the depth of functionality that the module provides. In other words, shallow modules do not add any benefit in reducing the complexity of a module because of their complex interfaces and simple functionality.

Why is it bad?

- Shallow modules have a large surface area and little added functionality. More surface area means more places to coupling, as well as less information hiding.
- Shallow modules do not encapsulate any complexity: whatever benefit we could earn from not having to learn the module's internals is lost because we need to learn a complex interface.

How to fix it?

- Aim for deep modules: complex, meaningful features exposed via simple interfaces.
- Question the existence of shallow modules: if they are not doing much, can their functionality be absorbed by another module in a similar domain? Can they be reorganized as to provide more functionality with a simpler interface?