Why every single element of Solid is wrong?

A couple of years ago, Dan North, an experienced software developer gave a talk about whether the Solid principles where outdated. He realized that the idea of principles itself was problematic since principles are rules that you either follow or not, black, or white. Instead, he proposes thinking about properties. Properties should define a goal, rather than rules to follow, to move towards. You may be farther or closer to the goal so you can use it a filter to assess your code and decide what to do next. All cupid properties are interrelated, so an improvement on one of them will probably result in a positive effect on some others. CUPID properties try to make coding more joyful and pleasant to work with.

SOLID Principles

- Single Responsibility Principle

A class only needs one reason to change. Look at the example we have here (in the slides). We have three classes which are the geometry class, the AppGraphic class and the Rectangle class. The graphic class only uses the draw method while the geometry uses the one for calculating the area. The problem is that this design is violating the principle as the Rectangle class has two responsibilities. If any of the classes whether AppGraphic or Geometry requires a change it would affect the other one.

Dan thinks that this principle is outdated because he thinks that all the classes might be lazy and he also thinks that is like a procedure Extract-Transform and Load. Taking the responsibilities and transforming them into new classes. Any non-trivial code can have any number of reasons to change which may or may not include the one you got in mind.

- Open-Closed Principle

Classes should be opened for the extension but closed for the modification. Whenever a simple change in a program result in many other modifications in other dependent modules then your design is not rigid. The main idea of this is that the changes are done by adding new code not modifying the one that worked.

Look at this example Client and Service. Client uses service. What if we want to change Server? Then we would need to change the implementation of client. With just adding an interface we would solve the problem. Now client is open and closed

Dan thinks that the code is not something fragile or debt. If I write a simpler code, then I am winning the trade. Write a simple code and easy to change and you will have a code easy to change and to modify.

- Liskov Substitution Principle

This is a very basic one. What it states is that the subtypes must be able to substitute their base type. In other words, the objects of a program should be replaced for the instances of their subtypes without affecting the behavior of the program. Base types wouldn’t need to know what their subtypes are and also, we shouldn’t do anything special regarding the type

Dan says that most of the time this principle is misunderstood because people use to think that subtypes are like “acts like” or “sometimes is” and this is not good.

- Interface Segregation Principle

It is better to have several specific interfaces than a general one. Let me put an example. We have the Database interface with the method add. All the clients of the interface will be able to use this method. Now imagine that some clients also want to read. What should we do? Add a new method in the Database? NO That’s wrong because you are restricting to clients that only wanted to add to implement a method that they weren’t asking for. So, the solution is to divide this into two interfaces, one that can add and other that can do both.

- Dependency Inversion Principle

Top level modules must not depend on low level ones; both must depend on abstractions. Abstractions must not depend on details; it is the opposite way. We have to depend on abstractions, not of concrete implementations. We must program for an interface not for an implementation. Look at the example.

Properties of properties

Every property should have 3 properties

- Practical: To be practical properties need to be easy to articulate (describe), assess (use them as lens for code) and adopt (can start small and evolve)
- Human: Properties need to read from the perspective of people, not code. CUPID is about what it feels like to work with code, not an abstract description of code itself.
- Layered: properties should offer *guidance* for beginners and *nuance* for more experienced

CUPID Principles

- Composable: plays well with others
- Unix philosophy: does one thing well
- Predictable: does what you expect
- Idiomatic: feels natural
- Domain-based: the solution domain models the problem domain in language and structure

Composable

Software that is easy to use gets used, and used, and used again.

- Small surface area: For example, when using APIs, coding with a narrow one has less for you to learn, less to go wrong and less chance of conflict with other code. But be careful not to use a too narrow one, there is a sweet spot of just right cohesion between fragmented and bloated

- Intention revealing: Intention revealing code is easy to discover and easy to assess. You should be able to read code and quickly know what it is for.
- Minimal dependencies: Code with minimal dependencies gives you less to worry about and reduces library incompatibilities

Unix philosophy

- A simple, consistent model: The Unix philosophy says to write components that work together well, and that *do one thing and do it well*. For instance, the ls command lists details about files and directories, but it does not *know anything* about files or directories! There is a system command called stat that provides the information.
- Single purpose vs Single responsibility: At first it can look the same but "doing one thing well" is an outside-in perspective; it is the property of having a specific, well-defined, and comprehensive purpose. SRP is an inside-out perspective: it is about the organization of code. For example, when using an UI component, with SRP business and rendering should be separated while they could be grouped in one class since it is doing one thing well (control the UI component)

Predictable

Code should do what it looks like it does, consistently and reliably.

- Behave as expected: The first of Kent Beck's 4 rules of simple design is that code passes all the test. This should be true even where there are no tests, the behavior should be obvious from its structure and naming.
- Deterministic: Software should do the same thing every time. Deterministic code should be
    - Robust: In the breadth and completeness of situations we cover
    - Reliable: Acts as expected in situations we cover
    - Resilient: How well we handle uncovered situations
- Observable: Could should be observable in the control theory sense, that is we can infer its internal state from its outputs. Four stage model: Instrumentation, Telemetry, Monitoring and Alerting.

Idiomatic

- Everyone has their own coding style, different one from another. Idiomatic code is code that humans can understand, writing code for someone else, users, future developers, support people or even you, making it easy to understand, and then, to work on it.
- Language idioms: Depending on the language you are using you may find different ways to execute the same action. Some languages have strong opinions about how code should look but others offer more possibilities to do one action, which increases your cognitive load, impacting your capacity to think about the problem

Domain-based

We write software to meet a need. Whatever its purpose, code should convey what it is doing in the language of the problem domain to minimize cognitive distance between what you write and what it does.

- Domain-based language: You can declare a surname as *string[30]* but defining a Surname type will be more intention revealing.
- Domain-based structure: Many frameworks offer a skeleton project with a directory layout to get started quickly. This imposes an a priori structure on your code that probably has nothing to do with the problem you are solving. This increases cognitive load when trying to solve errors (since related files are in different directories) and reduces cohesion adding effort when making changes. Using a domain-based structure eases the understand and navigation of the code
- Domain-based boundaries: When we structure the code the way we want and name it the way we want, module boundaries become domain boundaries, and deployment becomes straightforward.