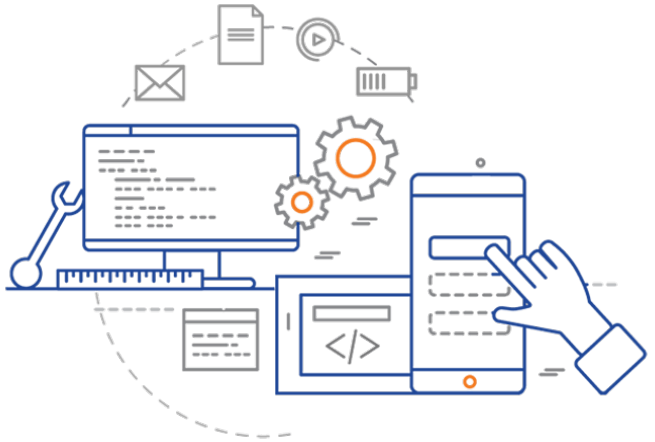


CS 398: Application Development

Week 04 Video: Analysis & Design 3

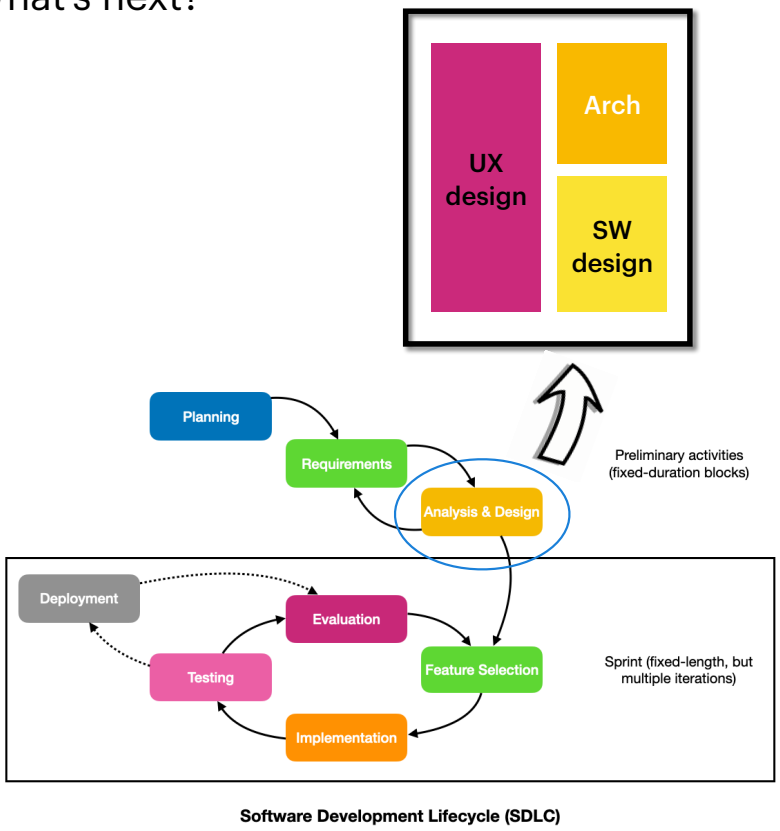
UX design; Design principles; SOLID principles

Phase 3: Analysis & Design

We've determined the high-level structure of our application. What's next?

1. We can consider the **UX design** of our system.
   - Mockups of critical screens.
   - Iterate on the user-interface design.
2. We can consider **software design** of our system:
   - What critical classes we might need.
   - How these classes should interrelate to one another.

UI prototypes are sometimes included in the requirements phase (i.e. iterating on user requirements). I prefer to think about the interface *after* making key technical decisions.



Software Development Lifecycle (SDLC)

What is Software Design?

The term “software design” is overused. The definition will vary depending on who you ask.

- A **UX designer** will treat design as the process of working with users to identify requirements, and iterating on the interaction and experience design with them to fine tune how they want the experience to work.
- A **software engineer** will want to consider ways of designing modules and source code that emphasize desirable characteristics like scalability, reliability and performance.
- A **software developer** may want to consider structure of their code, readability and maintainability, and correctness of the results (among other things).

In this course, we’ll define design as the set of **low-level implementation decisions that are made *prior to writing code***. We will include UX, software design and related discussions together.

UX Design & Prototyping

Designing for usability and user satisfaction.

User Experience Design (UX/UXD)

User experience design is about designing for people first: building a well-organized, cohesive and compelling user experience.

Building a consistent visual and behavioural experience provides a number of benefits.

- **Users will learn your application faster** if the user interface is familiar.
- **User can accomplish their tasks more quickly**, since the interface is consistent with other applications they've used.
- Users with disabilities will find your product **more accessible**.
- Your application will be **easier to document and explain**.

Apple's Human Interface Guidelines (1/2)

Since 1983, Apple has published their "Human Interface Guidelines", promoting their vision of a compelling interface. — https://developer.apple.com/design/human-interface-guidelines/

1. **Metaphor**. Take advantage of people's knowledge of the world by using metaphors to convey concepts and features of your application. e.g. the metaphor of file folders for storing documents.
2. **Reflect the User's Mental Model**. The user already has a mental model that describes the task your software is enabling, formed from real-world experiences, and experience with other software. An application that ignores the user's mental model would be difficult and even unpleasant to use; do what users expect!
3. **Explicit and Implied Actions**. Most operations involve the manipulation of an object using an action.
   - **Explicit**: The user sees the desired object onscreen, selects that object, and performs an exploit action to interact with it. e.g. using a menu command. Explicit actions are obvious and clear
   - **Implicit**: Implied actions arise from interaction with objects (e.g. dragging a graphic into a window to import it). If used, the interaction needs to be obvious (and probably leverage metaphor).

Apple's Human Interface Guidelines (2/2)

4. **Direct Manipulation**. Direct manipulation is an example of an implied action that allows users to feel that they are controlling the objects represented by the computer. The impact of the action should be immediately visible. e.g. selecting text with the mouse; dragging a graphic between windows to move it.
5. **User Control**. Allow the user to initiate and control actions. Provide users with capabilities, but let them remain in control.
6. **Feedback and Communication**. Keep users informed about what's happening by providing appropriate feedback and enabling communication with your application. Look for ways to communicate state!
7. **Consistency**. Consistency in the interface allows users to transfer their knowledge and skills from one application to another. Use the standard elements of the interface to ensure consistency within your application and to benefit from consistency across applications. When uncertain, look at what previous applications have done.
8. **Modelessness**. As much as possible, allow users to do whatever they want at all times. Avoid using modes that lock them into one operation and prevent them from working on anything else until that operation is completed.

Incremental Design

How do you build compelling interfaces? You iterate on your designs!

1. Build a low-effort **prototype** of your application — a sketch, or mockup of what it will look like. If you have multiple windows, or screens, you will want to build prototypes of each screen.
2. Show the prototypes to users, and ask for **feedback**. Assess how easy they find it to use (not just appearance).
   - Helpful: “Here’s the screen that is intended to do X. How would you interact with this?”
   - Not helpful: “Here’s what I built. Do you like it?”
3. Use the feedback to iterate and improve on your design. Circle back to (1) until you are satisfied.

Avoid the temptation to “just code it”. A prototype deliberately represents a lower commitment (time, cost), encouraging you to discard or modify it as needed.

It can often take multiple iterations to get to a design that works well.

Low-Fidelity Prototypes

Low-fidelity prototypes are deliberately simple, low-tech, and represent a minimal investment.

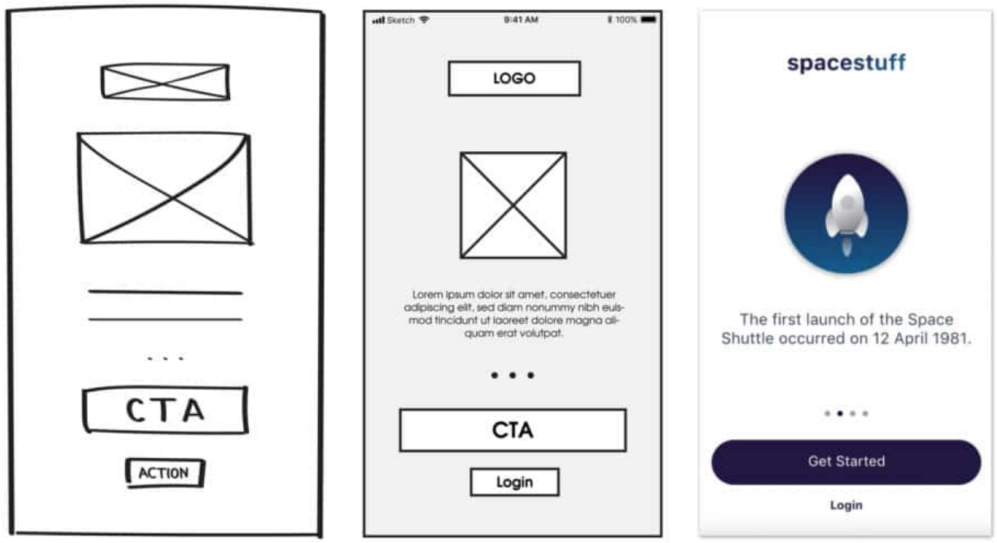
Building prototypes

- You can sketch something on paper.
- Many online tools help you build wireframe diagrams that you can show users.
- You can even make them semi-interactive to test progression through the interface.

What to prototype?

- Screens
- Interaction (through multiple screens)
- https://www.youtube.com/watch?v=yafaGNFu8Eg

These can get elaborate!



A low, medium and high-fidelity (final) version of a dialog.

Features of “Good Design”

What does it mean for software to be “well-designed”?

What is “Good Design”?

> It doesn’t take a huge amount of knowledge and skill to get a program working. Kids in high school do it all the time… The code they produce may not be pretty; but it works. It works because **getting something to work once just isn’t that hard.**
>
> **Getting software right is hard.** When software is done right, it requires a fraction of the human resources to create and maintain. Changes are simple and rapid. Defects are few and far between. Effort is minimized, and functionality and flexibility are maximized.
>
> – Robert C. Martin, Clean Architecture (2016).

We’re writing software in a hostile, changing environment.

How do we handle this? How do we “do software right”, in a way that “minimizes effort and maximized functionality and flexibility”?

Flexibility & Extensibility

Conditions will change over the lifetime of your software, and you need to design in a way that allows you to respond to those changes.

- The operating environment may change (e.g. a new version of Windows is released).
- The customer may want a new feature added (e.g. add support for a new payment system).
- You might decide to add more features as well (e.g. speech recognition!)

Flexibility implies that you can make changes to your code without breaking it (i.e. the opposite of "brittle code"). This also suggests simplicity in design, to enable non-breaking changes.

Extensibility implies the ability add new features, or drastically expand existing features. e.g. an image editor adding support for a new image type; a plain text editor adding support for code fences and syntax highlighting.

Simplicity & Readability

> We're programmers. Programmers are, in their hearts, architects, and the first thing they want to do when they get to a site is to bulldoze the place flat and build something grand. We're not excited by incremental renovation: tinkering, improving, planting flower beds.
>
> There's a subtle reason that programmers always want to throw away the code and start over. The reason is that they think the old code is a mess. And here is the interesting observation: they are probably wrong. The reason that they think the old code is a mess is because of a cardinal, fundamental law of programming:
>
> **It's harder to read code than to write it**.
>
> —Joel Spolsky (2000)

It's not enough to have code that works; your code should also be **clear** and understandable to everyone that will need work with it . Note that"other people" may include future-you. Will you understand this code one year from now?

This is also why experienced developers strive for clarity and (as much as possible) **simplicity** in their designs.

Support Code Reuse

Software is expensive and time-consuming to produce, so anything that reduces cost or time is welcome. Reusability, or code reuse is often positioned as the easiest way to accomplish this. It also reduces risk, since you are reusing tested code, instead of writing new, potentially defective code.

There are different levels of reuse:

- At the lowest level, you reuse classes: class libraries, containers, maybe some class “teams” like container/iterator.
- Frameworks are at the highest level. They really try to distill your design decisions. They identify the key abstractions for solving a problem, represent them by classes and define relationships between them.
- There also is a middle level. This is where I see patterns. Design patterns are both smaller and more abstract than frameworks. They’re really a description about how a couple of classes can relate to and interact with ”

— Distilled from Eric Gamma (2005)

Design Principles

How do we accomplish this?

Encapsulate What Varies

Identify the aspects of your application that vary and separate them from what stays the same.

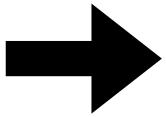
The main goal of this principle is to minimize the effect caused by changes.

You can do this by encapsulating classes, or functions. In both cases, your goal is separate and isolate the code that is likely to change from the rest of your code. This minimizes what you need to change over time.

```
fun getOrderTotal(order) {
    var total = 0
    for (item in order.lineItems) {
        total += item.price * item.quantity

        if (order.country == "US")
            total += total * 0.07 // US sales tax
        else if (order.country == "EU")
            total += total * 0.20 // European VAT
    }
    return total
}
```

Changing tax rates are likely.



```
fun getOrderTotal(order) {
  total = 0
  for (item in order.lineItems) {
    total += item.price * item.quantity
  }
  total += total * getTaxRate(order.country)
  return total
}

fun getTaxRate(country) {
    return when (country) {
        "US" -> 0.07 // US sales tax
        "EU" -> 0.20 // European VAT
        else -> 0
    }
}
```

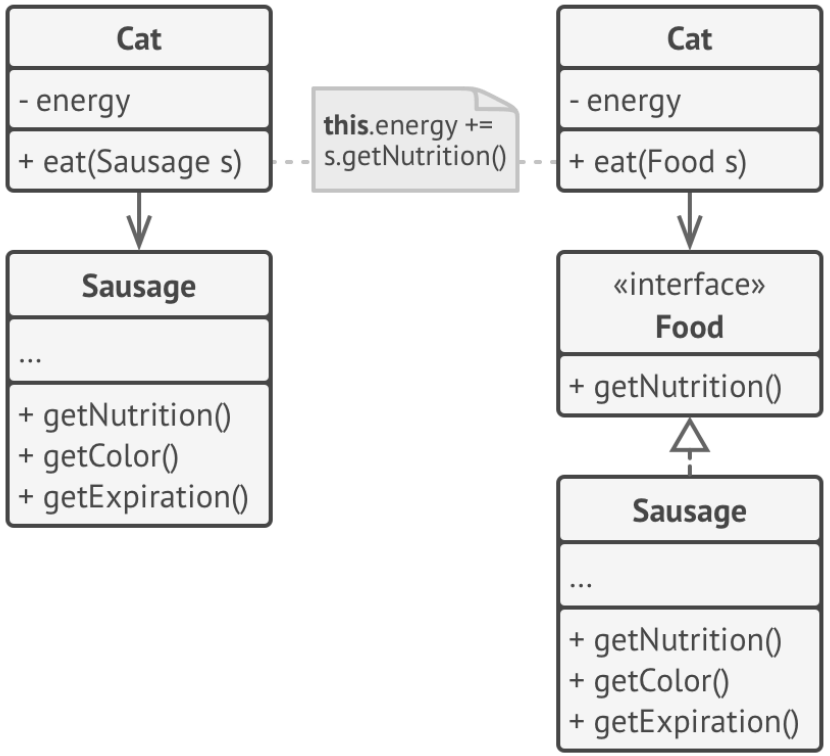
Program to an Interface, Not an Implementation

Program to an interface, not an implementation.

Dependencies between classes should be based on abstractions, not on concrete classes. This allows for maximum flexibility.

When classes rely on one another, you want to minimize the dependency - we say that you want *loose coupling* between the classes. Do do this, you extract an abstract interface, and use that to describe the desired behaviour between the classes.

e.g. our cat on the left can *only* eat sausage. The cat on the right can eat anything that provides nutrition, *including* sausage.



Favour Composition over Inheritance

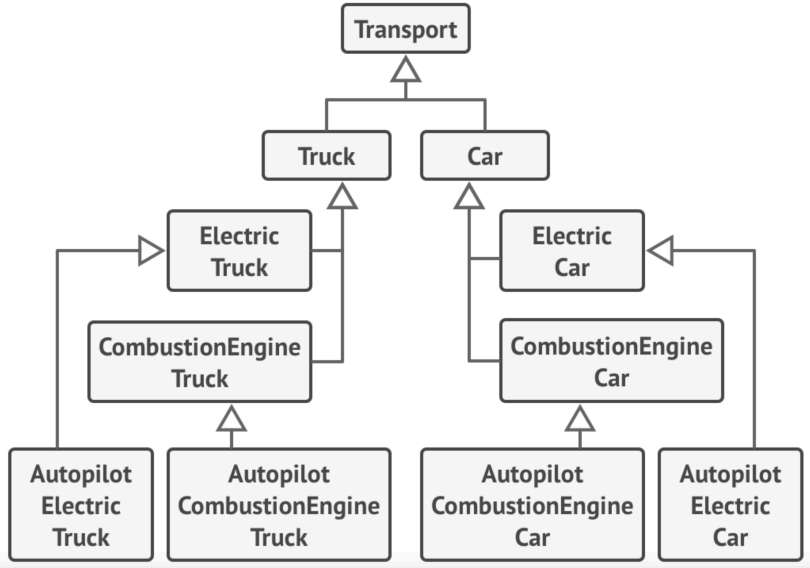
Inheritance is a useful tool for reusing code. In principle, it sounds great - derive from a base class, and you get all of it's behaviour for free!

Unfortunately it's rarely that simply. There are sometimes negative side effects of inheritance.

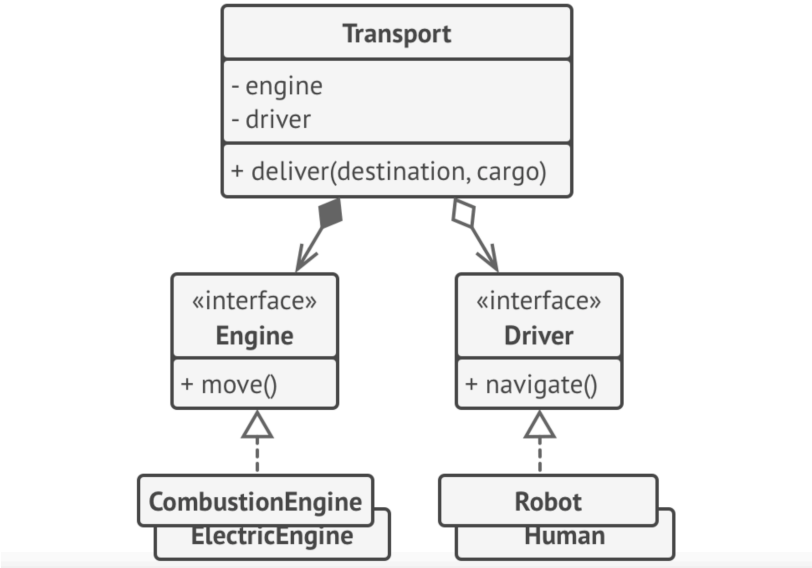
1. A subclass cannot reduce the interface of the base class. You have to implement all abstract methods, even if you don't need them.
2. When overriding methods, you need to make sure that your new behaviour is compatible with the old behaviour. In other words, the derived class needs to act like the base class.
3. Inheritance breaks encapsulation, because the details of the parent class are potentially exposed to the derived class.
4. Subclasses are tightly coupled to superclasses. A change in the superclass can break subclasses.
5. Reusing code through inheritance can lead to parallel inheritance hierarchies, and an explosion of classes. See below for an example.

A useful alternative to inheritance is composition. Where inheritance represents an **is-a** relationship (a car is a vehicle), composition represents a **has-a** relationship (a car has an engine).

Imagine a catalog application for cars and trucks.



Inheritance leads to class explosion, and unused intermediate classes.



Composition (aggregation) greatly reduces the complexity, and models based on supported behaviours.

SOLID Principles

How to design classes to behave well together.

SOLID Principles

SOLID was introduced by Robert ("Uncle Bob") Martin around 2002.

The SOLID Principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected ("class" meaning "a grouping of functions and data")

The goal of the principles is the creation of mid-level software structures that:

- Tolerate change (extensibility),
- Are easy to understand (readability), and
- Are the basis of components that can be used in many software systems (reusability).

There are five SOLID principles, and we'll walk through them.

Diagrams are from Ugonna Thelma: The S.O.L.I.D. Principles in Pictures.

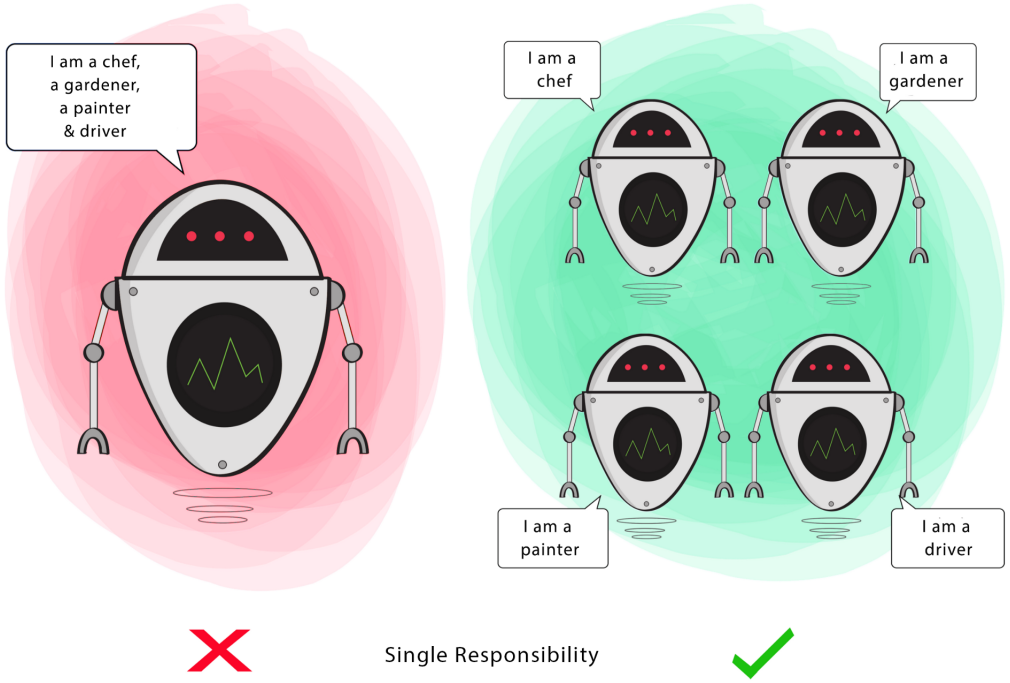
SOLID Principles

1. Single Responsibility

The Single Responsibility Principle (SRP) states that we want classes to do a single thing.

This is meant to ensure that are classes are focused, but also to reduce pressure to expand or change that class.

- A class has responsibility over a single block of functionality.
- There is only one reason for a class to change.



SOLID Principles

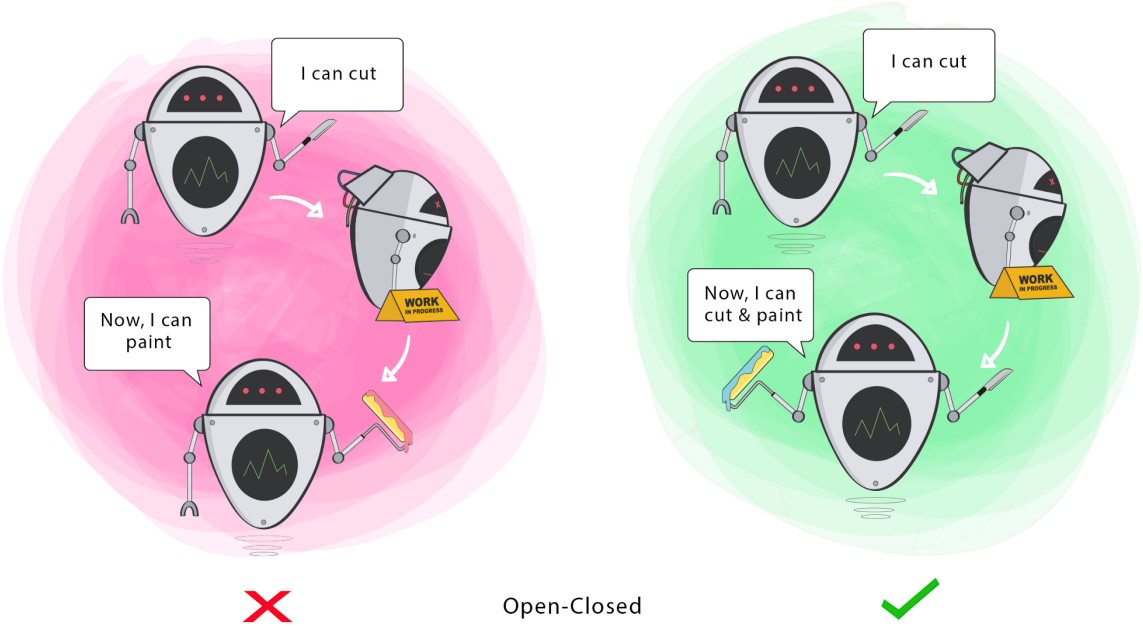
2. Open-Closed Principle

This principle champions subclassing as the primary form of code reuse.

A particular module (or class) should be reusable without needing to change its implementation.

"A software artifact should be open for extension but closed for modification. In other words, the behaviour of a software artifact ought to be extendible, without having to modify that artifact. "

– Bertrand Meyers (1988)



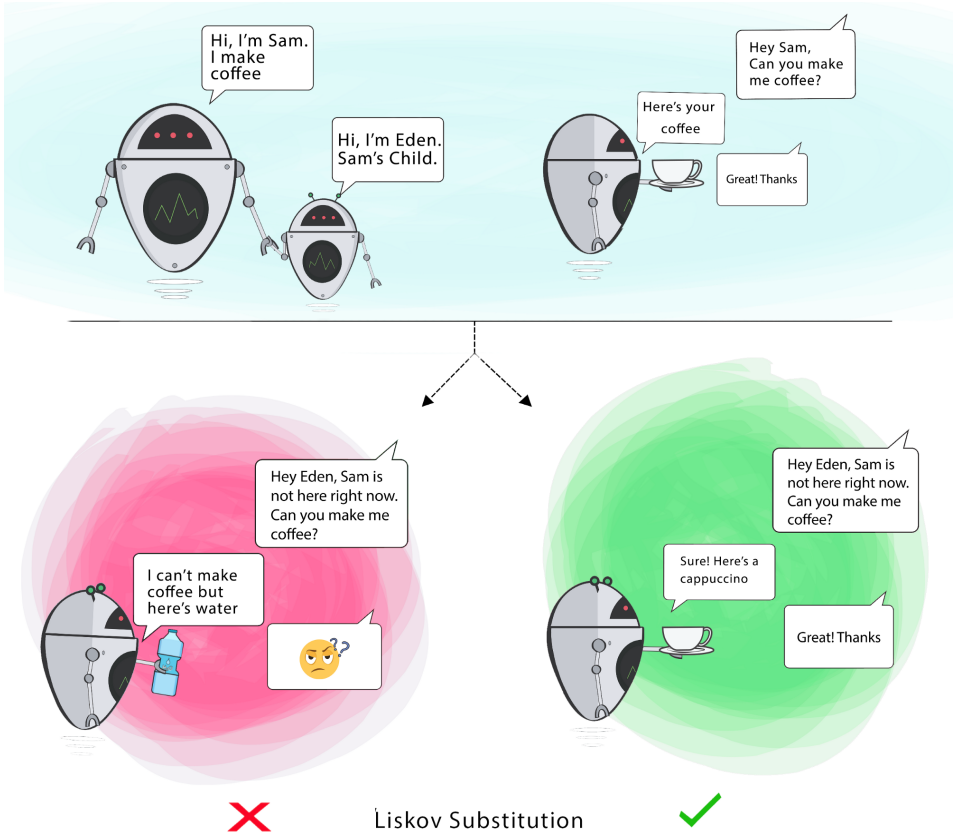
SOLID Principles

3. Liskov-Substitution Principle

It should be possible to substitute a derived class for a base class, since the derived class should retain the base class behaviour.

In other words, a child should always be able to substitute for its parent.

> “If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2, then S is a subtype of T”.
>
> -– Barbara Liskov (1988)



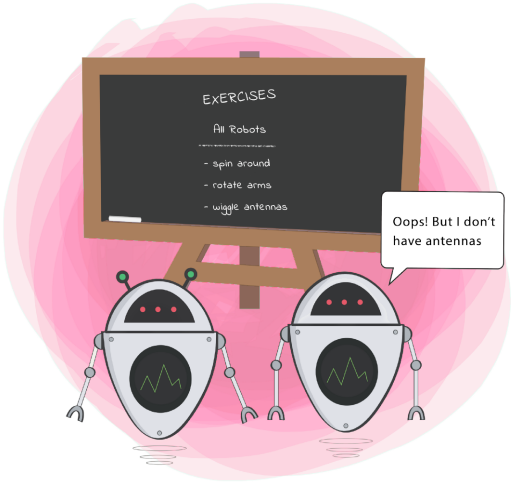
SOLID Principles

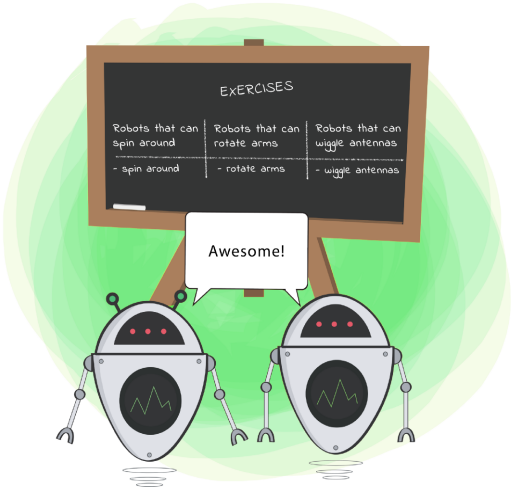
4. Interface Substitution

It should be possible to change classes independently from the classes on which they depend.

Also described as “program to an interface, not an implementation”. This means focusing your design on what the code is doing, not how it does it.

Never make assumptions about what the underlying code is doing – if you code to the interface, it allows flexibility, and the ability to substitute other valid implementations.





Interface Segregation

SOLID Principles

5.Dependency Inversion

The most flexible systems are those in which source code dependencies refer to abstractions (interfaces) rather than concretions (implementations). This reduces the dependency between these two classes.

- High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.