OOP Class Relationships

Type Based Relationships

Type based relationships are characterized by how two classes are related to each other through ontological hierarchy.

Type Based Relationships

There are two type based relationships (there can be an extra one which is a type relationship that is sort of a hybrid of the two).

- Realization
- Specialization
- (Abstract class)

Realization

A realization relationship is a one way relationship that describes how something abstract is REALized by something concrete.

Realization (other names)

- *Realization – Abstraction.* A realization realizes an abstraction
- *Implementation – Interface.* An implementation implements an interface

Abstraction

An `Abstraction` is a special type of class that does not contain any implementation.

This means that an `Abstraction` doesn't have code that controls the form and behavior of the class

Abstraction

An `Abstraction` is a special type of class that does not contain any implementation.

This means that an `Abstraction` doesn't have code that controls the form and behavior of the class

Abstraction

An abstraction can only be useful if some other class realizes this abstraction.

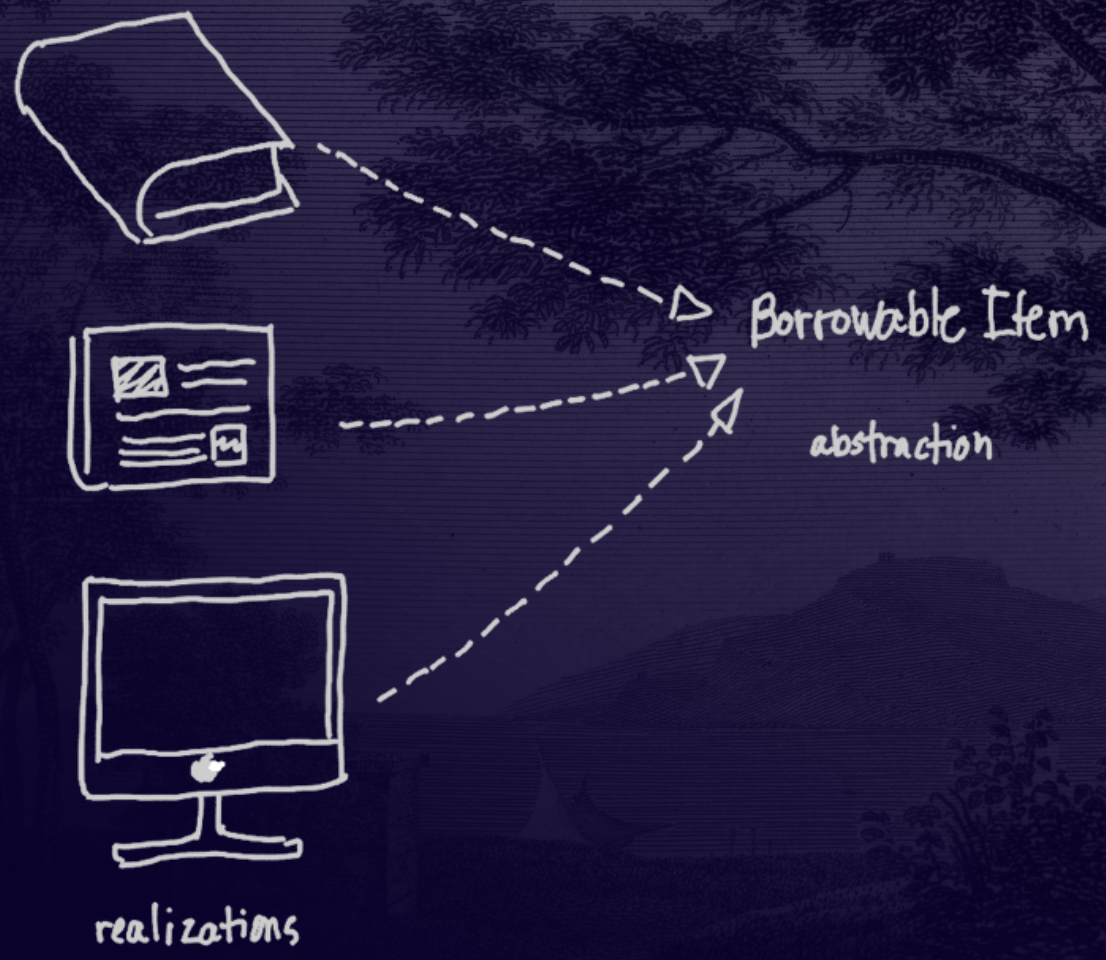
These `Realization` classes provide abstractions their form and behavior.

What's the point in maintaining some realization relationship between classes?

If abstractions can only be used through their realizations , then why create the abstraction at all?

Realization's importance

The importance of this seemingly pointless relationship lies in OOP's data hiding principle


Borrowable Item
abstraction
realizations

Realizing BorrowableItem

Any realization of `BorrowableItem`, such as `Book` or `Newspaper` will be forced to implement the `borrow()` and `return()` methods

Realization and Polymorphism

A `Book` is a `BorrowableItem`, allowing the library system to interact with it like any `BorrowableItem`.

But at the same time `Book` is a book so it behaves in the manner a book behaves.

Realization

By building all these relationships, the library system is able interact with resources without explicitly knowing which exact resource it is.

Realization

through the establishment of these relationships, OOP is able to uphold one of its core design principles, **data-hiding.**

Specialization

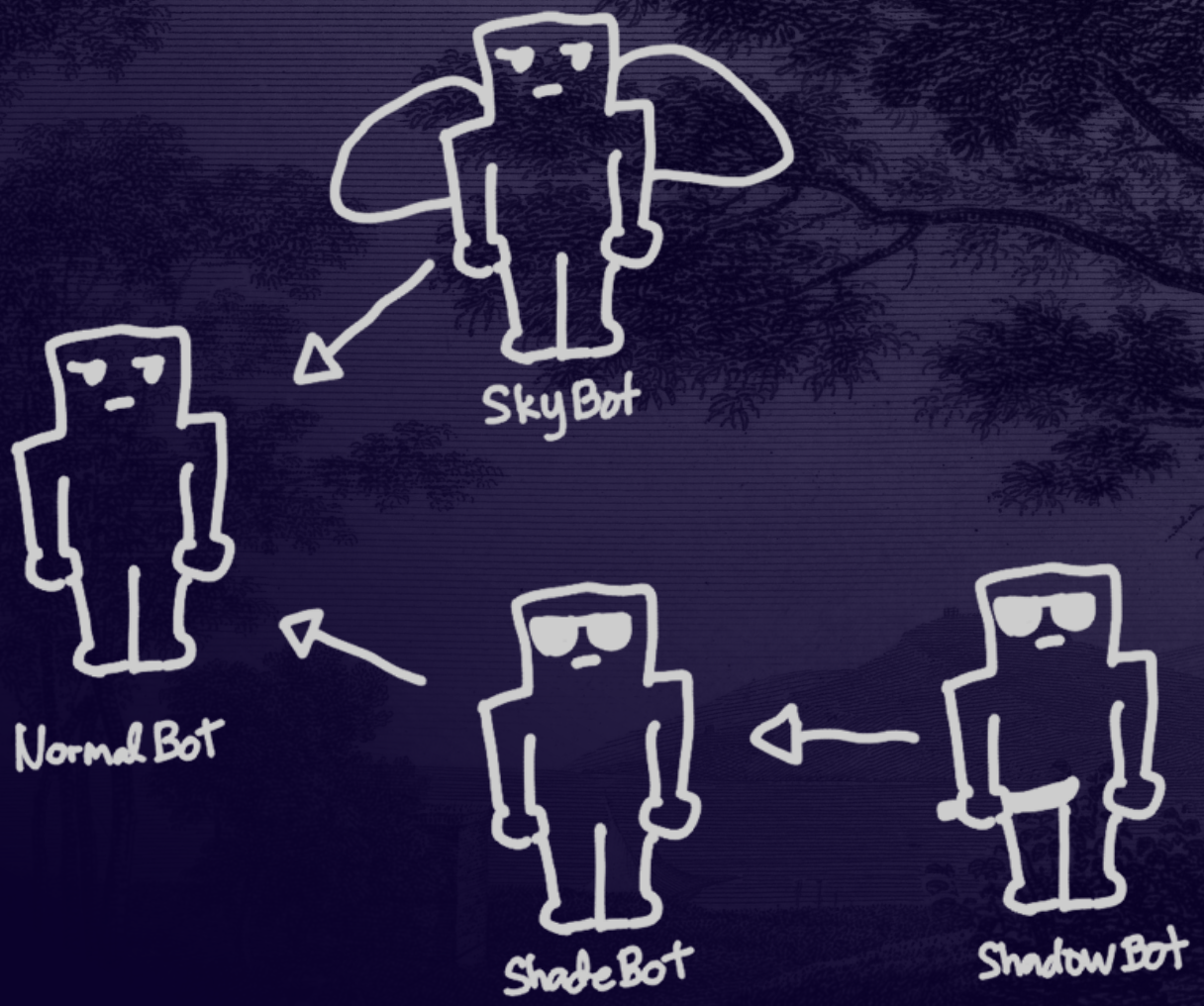
Specialization relationships are very similar to realization relationships.

You can think of these specialization relationships as realizations but between two real/concrete classes.

Specialization (other names)

Specialization relationships are sometimes called extension relationships

- *Child – Parent.* Child class inherits parent class
- *Sub Class – Super Class.* Sub class extends super class


SkyBot
NormalBot
ShadeBot
ShadowBot

Specialization inheritance

Specialization relationships enable one of OOP's core design principle, inheritance.

Specialization inheritance

When you write code for the general class, you do not need to rewrite it for specializations.

What you write inside specializations are the attributes and method that make it special

Specialization inheritance

Having one copy of code helps for maintainability.

When the recipe of all robot types need to change, the factory only needs to change the recipe of `NormalBot`, all of the special robots' recipes will change as well since they all use `NormalBot`'s recipe.

Specialization of Specialization

You can also specialize, specializations.

This is illustrated by `ShadowBot`, which is a special `ShadeBot` that has knife.

Specialization of Specialization

Since `ShadowBot` is a special `ShadeBot` and `ShadeBot` is a special `NormalBot`, `ShadowBot` is automatically a specialization of `NormalBot` as well.

Specialization and Polymorphism

Specializations also allow polymorphism in the same way realizations do.

A `ShadowBot` can be interacted with like any `NormalBot` or `ShadeBot` but since it is also a `ShadowBot` it will behave specifically like a `ShadowBot`.

Abstract Classes

An abstract class is something in between an abstraction and a generalization.

It contains attributes and methods with bodies but it also contains abstract methods as well.

Abstract Classes

An abstract class is something in between an abstraction and a generalization.

It contains attributes and methods with bodies but it also contains abstract methods as well.

Abstract Classes

When classes specialize/realize abstract classes, they inherit the attributes and methods with bodies but they are forced to implement the abstract methods as well.

Abstract Classes

These relationships are sometimes used if the system requires a mix of inheritance and implementation between classes.

Dependency Relationships

Dependency relationships, also known as **associations**, characterize how two classes interact with each other.

A class which is dependent on another class, needs to know how to interact with it.

Dependency Relationships

A dependency relationship is one way (but it is also possible for two objects to be dependent on each other).

A **client** class is dependent on some **dependency.**

Dependency Relationships

A dependency relationship is one way (but it is also possible for two objects to be dependent on each other).

A **client** class is dependent on some **dependency.**

Dependency Relationships

There are two types of dependencies

- Aggregation
- Composition

Dependency Relationships

There are two types of dependencies

- Aggregation
- Composition

Dragon Priest

Dragon Priest Mask
Dragon Priest Staff

Aggregation

Aggregation relationships are general usage and transactional dependencies.

When a dependency is an aggregate of some client, it means that the client merely **uses** the instances of this dependency.

Aggregation

These relationships are the looser forms of dependency, because the dependency instance can exist outside the lifetime of the client instance.

Composition

Composition relationships are ownership dependencies.

When a client is composed of some dependency, this means that the client **owns** the instances of this dependency.

Composition

These relationships are stronger forms of dependency since the existence of the dependency instance is tied to the client, meaning, the dependency ceases to exist outside the lifetime of the client instance.